Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#01: Verilog HDL Gate-Level Modelling

Instructor: Dr. Nabeel Siddiqui

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Sukkur Institute of Business Administration University

Department of Electrical Engineering

ESE-412: Digital System Design Lab

Handout#01: Verilog HDL Gate-Level Modelling

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali
033-18-0037

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Report Rubrics						Total Marks
	Criterion	**0.5**		**0.25**	**0.125**	
1	Accuracy	ϒ Desired output		ϒ Minor mistakes	ϒ Critical mistakes	
2	Timing	ϒ Submitted within the given time		ϒ 1 day late	ϒ More than 1 day late	

Lab Learning Objectives:

After completing this session, student should be able to:

- Identify logic gate primitives provided in the Verilog HDL
- Implement basic combinational circuits using gate-level modelling

Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

Gate-level Modelling:

Verilog is both a structural and behavioural language. Internals of each module can be defined at four levels of abstraction, depending on the need of the design. There are four levels of abstraction which include switch-level, gate-level, data flow, and behavioural or algorithm level. The switch-level is the lowest abstraction level, where module can be implemented in terms of switches, storage nodes, and interconnections between them.

The gate-level modelling is implemented in terms of logic gates and interconnections between these gates. This design method is similar to describing a design in terms of a gate-level logic diagram. Verilog allows the designer to mix and match all four levels of design methodologies in design. The modules behave identically to the external world identically irrespective of the level of abstraction at which module is described. Therefore, internals of the module can be changed without any change in the environment.

In the digital design community, the term register transfer level (RTL) is used for Verilog description that uses a combination of behavioural and data flow modelling. Normally, the higher level of abstraction, design will be more flexible and technology-independent. Although, the lower level description provides high performance therefore as the design matures, high level modules are replaced with the gate-level modelling.

Verilog primitives:
Logic primitives such as **not**, **and**, or, **nand**, **nor**, **xor** and **xnor** gates are classified as multiple-input gates. The syntax for the Verilog built-in primitive is as follows:

gate_type inst1(output_1, input_1, input_2,…, input_n);

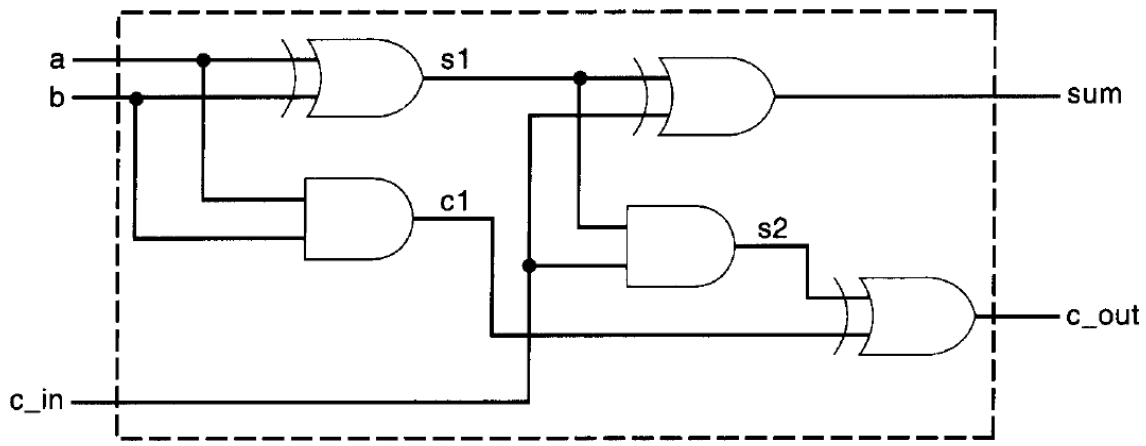
gate_type indicates type of built-in primitive, *inst1* indicates name of instantiation into the module which is optional, while in the brackets output is listed first, followed by the inputs.

Following gate primitives are used in the Verilog HDL. For respective truth tables, you may refer the book "Verilog – A guide to digital design and Synthesis", chapter #05.

#	Gate type
01	and
02	or
03	xor
04	nand
05	nor
06	xnor
07	buf
08	not
09	bufif1
10	bufif0
11	notif1
12	notif0

Lab Example#01: Full Adder

The logic diagram for the 1-bit Full Adder is converted to a Verilog description.



Listing 1.1: Full Adder

```
module fulladd(sum, c_out, a, b, c_in);
      output sum, c_out;
      input a, b, c_in;
      wire s1, c1, c2;
            xor (s1, a, b);
            and (c1, a, b);
            xor (sum, s1, c_in);
            and (c2, s1, c_in);
            or (c_out, c2, c1);
endmodule
```

Verilog code:

Lab Activities:

1. Design 1-bit Comparator with inputs A, B and outputs **A_gt_B**, **A_lt_B** and **A_eq_B**. Submit the results with comments.
2. Design 4 to 1 Multiplexer with inputs **S0**, **S1**, **D0**, **D1**, **D2**, **D3** and output **Y**. Submit the results with comments.
3. Design 1 to 4 Demultiplexer with inputs **Data**, **S0**, **S1** and outputs **Y0**, **Y1**, **Y2**, and **Y3**. Submit the results with comments.
4. Design Even Parity Generator with inputs **A**, **B**, **C**, **D** and output **EP**. Submit the results with comments.

Lab Activities: Demonstrate a blinking LED at 1 Hz using NEXYS4 DDR

```
module Blinky_1Hz(clock_in, clock_out);
        input clock_in;                 // System clock
      output clock_out;               // Low-frequency clock
      reg [31:0] counter_out;         // register for storing values
      reg clock_out;                 // register buffer for output port

   initial begin
      counter_out<=32'h00000000;     // 0 in Hexadecimal format
      clock_out<=0;
   end
    //this always block runs on the fast 100MHz clock
   always @(posedge clock_in)
         begin
            counter_out<=counter_out + 32'h00000001;  // Adding one at every clock pulse
            if (counter_out>32'h5F5E100)              // If count value reaches to 100000000 = 10^8
         begin
            counter_out<=32'h00000000;                // reset the counter
            clock_out <= ~clock_out;                 // and invert the clock pulse level
         end
         end
  endmodule
```

Hints:

FPGA Board system clock is 100 MHz

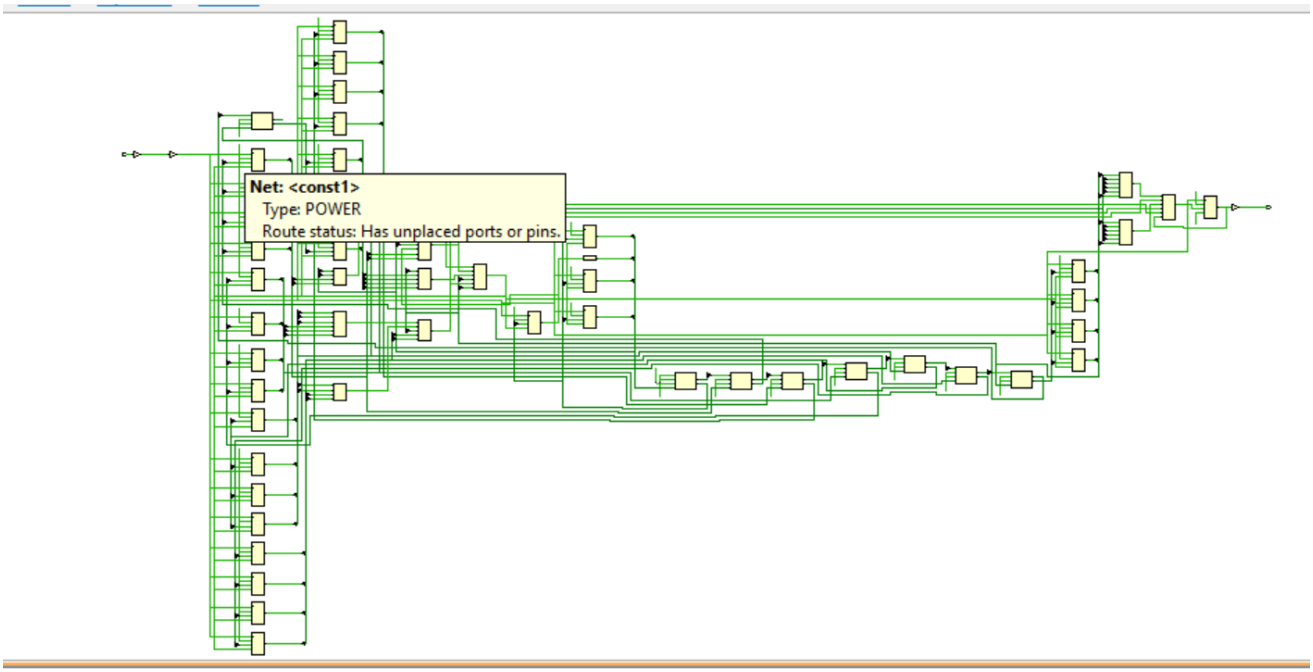
The system clock pin name is **E3** and **LED0** is labelled as **H17**.

Lab Exercise:

5. Amend the design problem to display LEDs blinking at 10 Hz. Submit the results with comments.

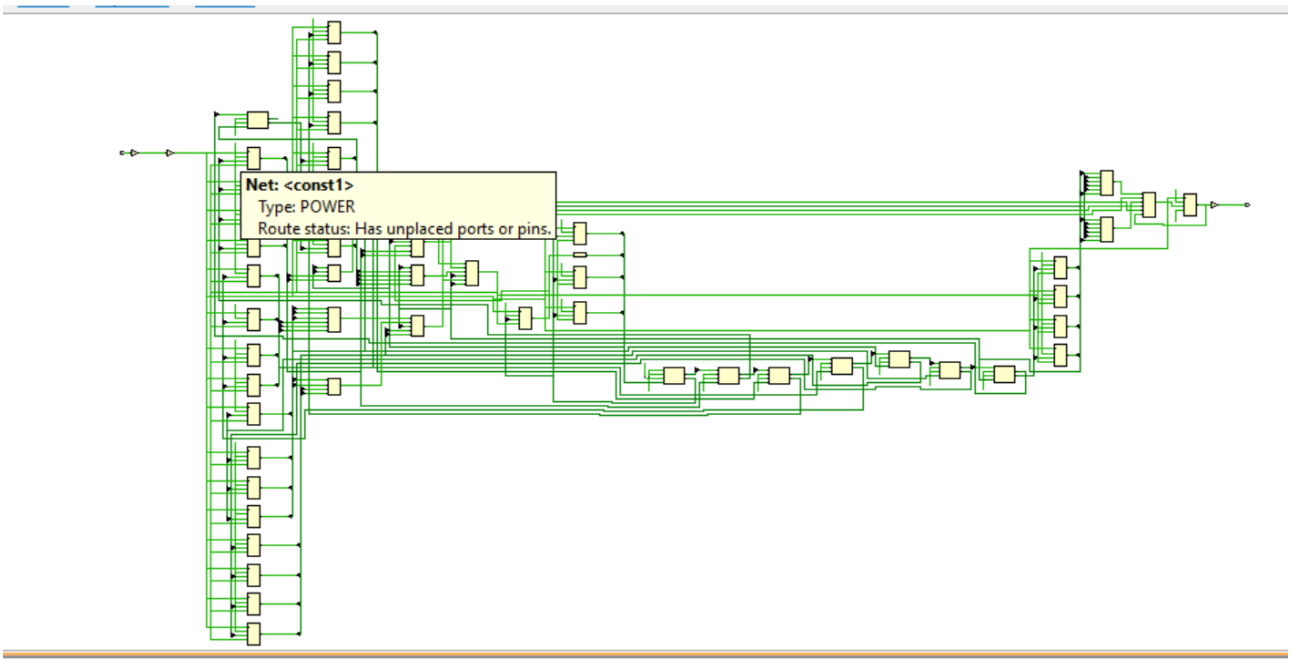
```
module Blinky_1Hz(clock_in, clock_out);

 input clock_in; // System clock

 output clock_out; // Low-frequency clock

 reg [31:0] counter_out; // register for storing values
```

```
reg clock_out; // register buffer for output port


initial begin
counter_out<=32'h00000000; // 0 in Hexadecimal format
clock_out<=0;
end
//this always block runs on the fast 100MHz clock
always @(posedge clock_in)
begin
counter_out<=counter_out + 32'h00000001; // Adding one at every clock pulse
if (counter_out>32'h3B9ACA00) // If count value reaches to 1000000000
begin
counter_out<=32'h00000000; // reset the counter
clock_out <= ~clock_out; // and invert the clock pulse level
end
end
    endmodule
```



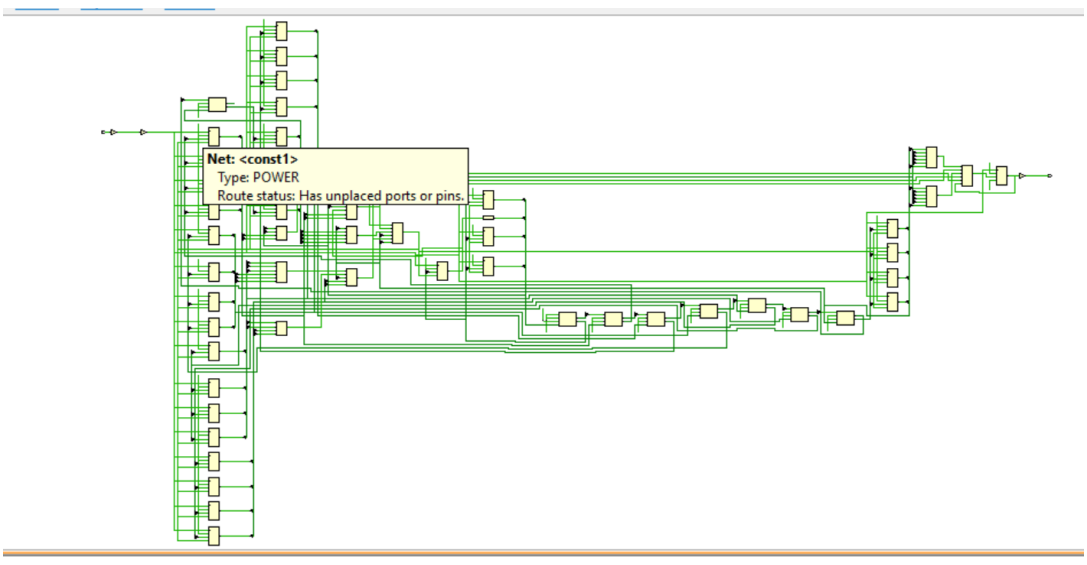
6. Amend the code so that all the 16 LEDs on the board blink at 10 Hz. Submit the results with comments.

```
module Blinky_1Hz(clock_in, clock_out);
 input clock_in; // System clock
 output clock_out; // Low-frequency clock
 reg [31:0] counter_out; // register for storing values
 reg [15:0]clock_out; // register buffer for output port

 initial begin
 counter_out<=32'h00000000; // 0 in Hexadecimal format
 clock_out<=0;
 end
 //this always block runs on the fast 100MHz clock
 always @(posedge clock_in)
 begin
 counter_out<=counter_out + 32'h00000001; // Adding one at every clock pulse
 if (counter_out>32'h3B9ACA00) // If count value reaches to 1000000000
 begin
 counter_out<=32'h00000000; // reset the counter
 clock_out <= ~clock_out; // and invert the clock pulse level
 end
 end
 endmodule
```



7. Amend the code so that all the 16 LEDs toggle at Even and Odd positions, alternatively. Submit the results with comments.

```
module Blinky_1Hz (clock_in, clock_out);

input clock_in;                 // System clock
output clock_out;                // Low-frequency clock
reg[31:0] counter_out;           // register for storing values
reg[15:0] clock_out = 16 'b0101010101010101; // register buffer for output port

 initial begin
 counter_out<=32' h00000000;     // 0 in Hexadecimal format
clock_out <= 0;

end
  //this always block runs on the fast 100MHz clock
  always @ (posedge clock_in)
 begin
 counter_out <= counter_out + 32 'h00000001; // Adding one at every clock pulse
 if (counter_out>32' h3B9ACA00)     // If count value reaches to 1000000000
  begin
 counter_out <= 32 'h00000000; // reset the counter
 clock_out <= ~clock_out; // and invert the clock pulse level
 end
 end
 endmodule
```



Detailed readings:

Verilog HDL A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition, chapter 5

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#02

Hierarchical Modelling Approach

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#02

Hierarchical Modelling Approach

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali 033-18-0037

Lab Learning Objectives:

After completing this session, student should be able to:

- Understand and apply hierarchical modelling concepts
- Explain difference between module and module instance in Verilog

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	Desired output	Minor mistakes	Critical mistakes	

2	Timing	☐ Submitted within the given time	☐ 1 day late	☐ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

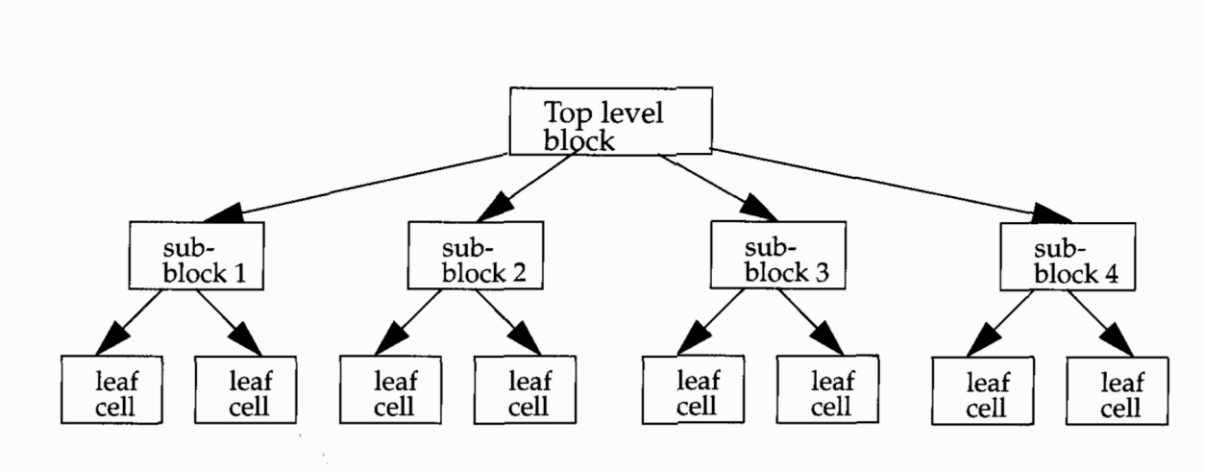
Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

Design methodologies:

There are basically two design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build-up the top-level block. We further subdivide the sub-blocks until we come to leaf cells, which cannot be further divided, as shown in figure below.



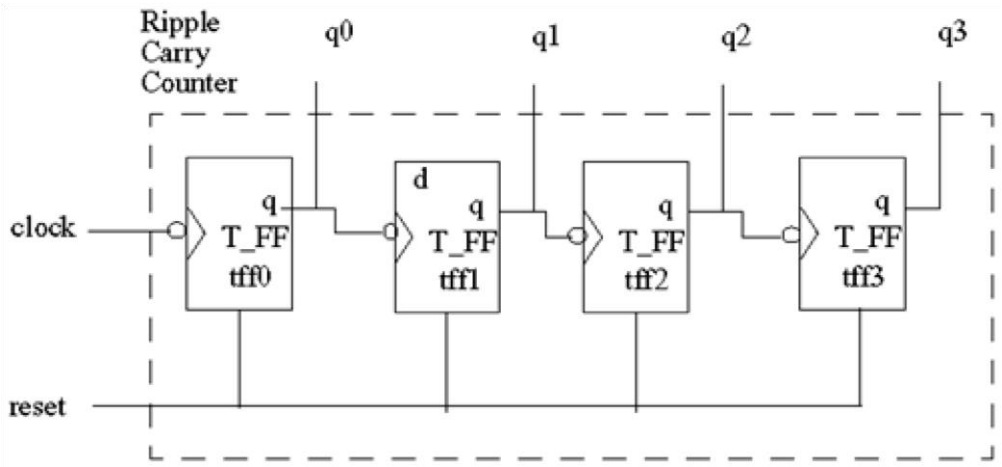
In the bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. Following figure shows the bottom-up design process. Typically, a combination of top-down and bottom-up design flow is used.

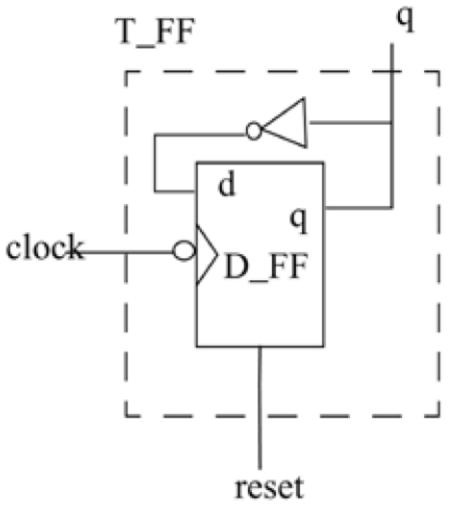
Hierarchical Design

Hierarchical Design (HD) flow enable you to partition a design into smaller, more manageable modules to be processed independently. Using a modular approach to the hierarchical design lets you analyse modules independent of the rest of the design, and reuse modules in the top-down design. A team of users can iterate on specific sections of a design, achieving timing closure and other design goals, and reuse the results.

4-bit Ripple Carry Counter

The ripple carry counter shown in figure below is made up of negative edge-triggered toggle flip flops (*T_FF*). Each of the *T_FFs* can be made up from negative edge-triggered D-flip flop (*D_FF*) and inverters (assuming *q_bar* output is not available on the *D_FF*).





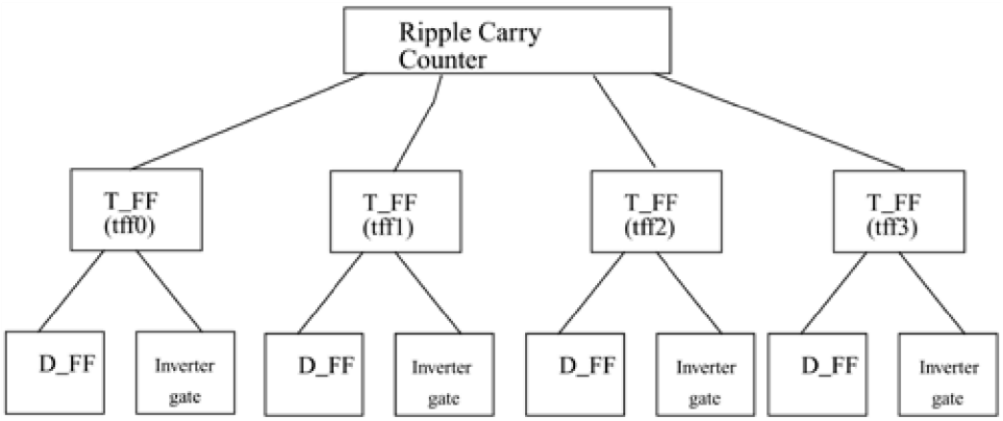
Thus, the ripple carry counter is built in a hierarchical fashion by using building blocks.

Difference between module and instance:

Modules

We now relate these hierarchical modelling concepts to Verilog. 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. Ripple carry counter, *T_FF*, *D_FF* are examples of modules. 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.



Instances

A module provides a template from which you can create actual objects. When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface. The process of creating objects from a module template is called instantiation, and the objects are called instances. In example below, the top-level block creates four instances from the T-flip flop (*T_FF*) template. Each *T_FF* instantiates a *D_FF* and an inverter gate. Each instance must be given a unique name. Note that // is used to denote singleline comments.

Lab Examples:

1. Implement 4-bit ripple carry counter using hierarchical modelling approach and test the design.

Listing 2.1: Ripple Carry Counter

```
module ripple_carry_counter(q, clk, reset);
      output [3:0] q; input clk, reset;
            T_FF tff0(q[0],clk, reset); T_FF
            tff1(q[1],q[0], reset);
            T_FF tff2(q[2],q[1], reset);
            T_FF tff3(q[3],q[2], reset);
endmodule
```

In the above module, four instances of the module *T_FF* (T-flip-flop) are used. Therefore, we must now define the internals of the module *T_FF*.

Listing 2.2: Toggle Flip Flop

```
module T_FF(q, clk, reset);
      output q; input clk,
      reset; wire d;
            D_FF dff0(q, d, clk, reset); not
            n1(d, q);
endmodule
```

The behavioural D flip-flop design is given below.

Listing 2.3: D Flip Flop

```
module D_FF(q, d, clk, reset);
      output q; input d, clk,
      reset; reg q;

always @(posedge reset or negedge clk)
      if (reset) q <= 1'b0;
      else q <= d;
endmodule
```

Lab Exercises:

1. Use the Blinky_1Hz design module from Lab#01 along with Ripple Carry Counter, and demonstrate the results using Nexys 4 DDR FPGA board. Submit the results of both simulation and hardware along with comments.

Exercise Code

```
module Blinky_1Hz(clock_in, clock_out);
input clock_in; // System clock
output clock_out; // Low-frequency clock
reg [31:0] counter_out; // register for storing values
reg clock_out; // register buffer for output port
initial begin
counter_out<=32'h00000000; // 0 in Hexadecimal format
clock_out<=0;
end
//this always block runs on the fast 100MHz clock
always @(posedge clock_in)
begin
```

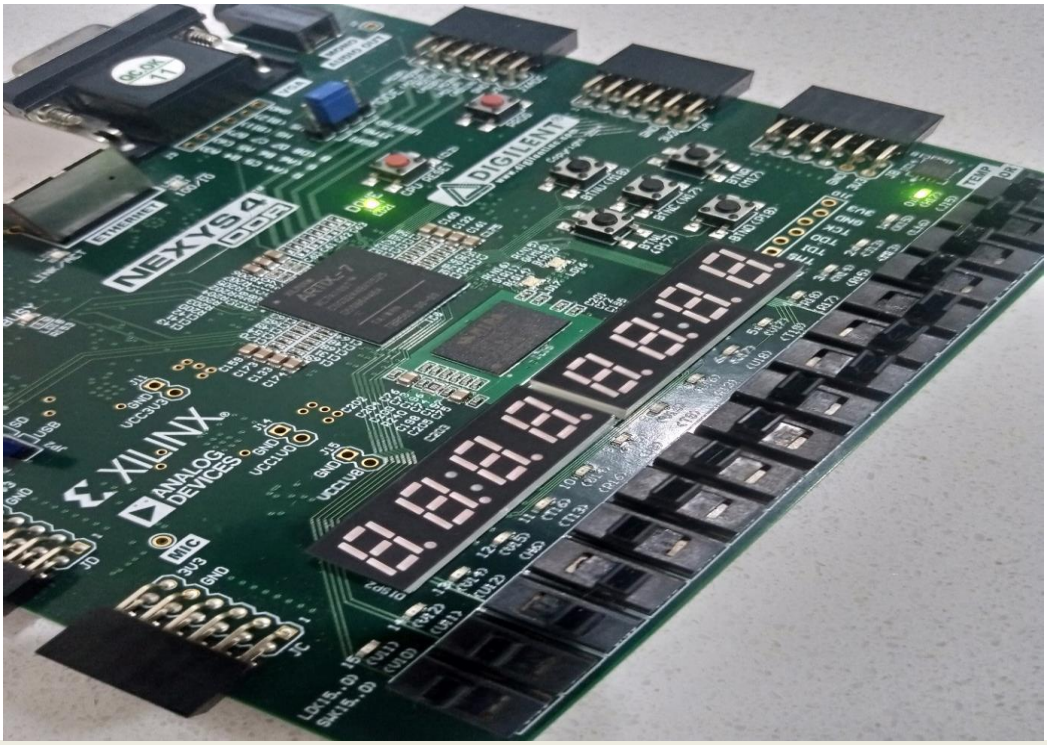
```
counter_out<=counter_out + 32'h00000001; // Adding one at every clock pulse
if (counter_out>32'h5F5E100) // If count value reaches to 100000000 = 10^8
begin
counter_out<=32'h00000000; // reset the counter
clock_out <= ~clock_out; // and invert the clock pulse level
end
end
endmodule


`timescale 1ns / 1ps
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;
T_FF tff0(q[0],clock_out, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

Blinky_1Hz b1(clk, clock_out);

endmodule


module T_FF(q1, clk, reset);
output q1;
input clk, reset;
wire d;
D_FF dff0(q1, d, clk, reset);
not n1(d, q1);
endmodule


module D_FF(q0, d, clk, reset);
output q0;
input d, clk, reset;
reg q0;
always @(posedge reset or negedge clk)
if (reset)
q0 <= 1'b0;
else
q0 <= d;
endmodule
```

2. Design 16-to-1 multiplexer by interconnecting five 4-to-1 multiplexers using hierarchical modelling approach. Test the design using both simulation and Nexys 4 DDR FPGA board. Submit the results of both simulation and hardware along with comments.

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

```
module Four_1_Mux(Out, S, D);

input [3:0]D;
input [1:0]S;
output Out;

not not0(S0, S[0]);
not not1(S1, S[1]);

and and0(W1, D[0], S0, S1);
and and1(W2, D[1], S0, S[1]);
and and2(W3, D[2], S[0], S1);
and and3(W4, D[3], S[0], S[1]);

or or1(Out, W1, W2, W3, W4);

endmodule
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date: 11/02/2021 01:23:16 AM
// Design Name:
// Module Name: Mux_16_1
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////


module Mux_16_1(Out_Main, SW, Data);

input [15:0]Data;
```

```
input [3:0]SW;
output  Out_Main;

wire [3:0]M;


Four_1_Mux inst0(M[0], SW[1:0], Data[3:0]);
Four_1_Mux inst1(M[1], SW[1:0], Data[7:4]);
Four_1_Mux inst2(M[2], SW[1:0], Data[11:8]);
Four_1_Mux inst3(M[3], SW[1:0], Data[15:12]);
Four_1_Mux inst4(Out_Main, SW[3:2], M[3:0]);


endmodule
```



Detailed readings:

Verilog HDL A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition, chapter 2

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#03

Introduction to Data Flow Modelling

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#03

Introduction to Data Flow Modelling

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali 033-18-0037

Lab Learning Objectives:

After completing this session, student should be able to:

- Describe the continuous assignment
- Incorporating delays in data flow modelling
- Using Verilog HDL operators

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	Desired output	Minor mistakes	Critical mistakes	
2	Timing	Submitted within the given time	1 day late	More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Learning Objectives:

After completing this session, student should be able to:

- Describe the continuous assignment
- Incorporating delays in data flow modelling
- Using Verilog HDL operators

Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

Data flow modelling

As we know that, Gate-level modelling is highly intuitive, however, it is preferable for small circuits as gate count is low and designer can connect and instantiate every gate easily. Data flow modelling provides higher abstraction for design and is effective for more complex designs. As on every coming day, gate density in digital circuits is increasing therefore higher level of design abstractions are beneficial in every aspect. Synthesis tools have been developed and integrated in the computer aided design tools to convert data flow modelling based designs into gate-level designs.

Lab Examples: Continuous Assignment

A continuous assignment is the most basic statement in data flow modelling used to drive a value onto the net. The assignment statement starts with the keyword assign.

```
Continuous_assignment_keyword #delay_units net_assignment = net_value
```

Key points regarding the continuous assignment.

- The left-hand side of an assignment must always be a scalar or vector net or a concatenation of scalar and vector nets. It cannot be scalar or vector nets.
- Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right-hand side value changes and the value is assigned to the left-hand side net.
- The operands on the right-hand side can be scalar or vector registers or nets or function calls.
- Delay values can be specified for assignments in terms of time units and delay value is optional.

Regular continous assignment:

Following example codes show how continous assignments are specified in Verilog. This method is called regular continuous assignment declaration.

```
// Continuous assign. out is a net. i1 and i2 are nets. assign
out = i1 & i2;
// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];
// Concatenation. Left-hand side is a concatenation of a scalar //
net and a vector net.
assign {c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Implicit continuous assignment:

Instead of declaring a net and then wiritng a continous assignment on the net, Verilog provides a shortcut by which a contnius assignment can be placed on a net when it is declared. This is called implicit continous assignment method.

```
//Regular continuous assignment
wire out;
assign out = in1 & in2;
//Same effect is achieved by an implicit continuous assignment
wire out = in1 & in2;
```

Implicit net declaration:

If a signal name is use to left of the continuous assignment, an implicit net declaration will be inferred for that signal. If the net is connected with the module port, net width of module port is used for net width.

```
// Continuous assign. out is a net.
wire i1, i2;
assign out = i1 & i2; //Note that out was not declared as a wire
//but an implicit wire declaration for out //is
done by the simulator
```

Delays:

Delay values control the time between the change in right-hand side operand and when the new value is assigned to the left-hand side. The delay value is specified after the keyword assign. Any change in the operands on right-hand side will result in delay of 10 time units. Write down the following codes in Vivado and verify the equivalence of all three types of specifying delays.

```
// Regular assignment delay assign #10 out = in1 & in2; //
Delay in a continuous assign
//implicit continuous assignment delay wire
#10 out = in1 & in2;
```

```
//Net Declaration Delays
wire # 10 out; assign out
= in1 & in2;
```

Expressions, Operators and Operands:

Dataflow modelling describes the design in terms of expressions instead of primitive gates. Expressions, operators, and operands form the basis of dataflow modelling.

Expressions: Expressions are constructs that combine operators and operands to produce a result.

```
// Examples of expressions. Combines operands and operators a
^ b
addr1[20:17] + addr2[20:17] in1
| in2
```

Operands can be any one of the data types. Some constructs will take only certain types of operands. Operands can be constants, integers, real numbers, nets, registers, times, bit-select (one bit of vector net or a vector register), part-select (selected bits of the vector net or register vector), and memories or function calls (functions are discussed later).

```
integer count, final_count;
final_count = count + 1;//count is an integer operand real
a, b, c;
c = a - b; //a and b are real operands
reg [15:0] reg1, reg2; reg [3:0]
reg_out;
reg_out = reg1[3:0] ^ reg2[3:0];//reg1[3:0] and reg2[3:0] are
//part-select register operands reg
ret_value;
ret_value = calculate_parity(A, B);//calculate_parity is a
//function type operand
```

Operators act on the operands to produce desired results. Verilog provides various types of operators.

```
d1 && d2 // && is an operator on operands d1 and d2 !a[0] //
! is an operator on operand a[0]
B >> 1 // >> is an operator on operands B and 1
```

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language.

Arithmetic operators:

There are two types of arithmetic operators which include binary and unary. Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.

```
A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers
```

```
A * B // Multiply A and B. Evaluates to 4'b1100
D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111
B - A // Subtract A from B. Evaluates to 4'b0001
F = E ** F; //E to the power F, yields 16
```

If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

```
in1 = 4'b101x;
in2 = 4'b1010;
sum = in1 + in2; // sum will be evaluated to the value 4'bx
```

Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

```
13 % 3 // Evaluates to 1
16 % 4 // Evaluates to 0
-7 % 2 // Evaluates to -1, takes sign of the first operand
7 % -2 // Evaluates to +1, takes sign of the first operand
```

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand.

```
-4 // Negative 4
+5 // Positive 5
```

Logical operators:

Logical operators are logical-and (&&), logical-or (||) and logical-not (!). Operators && and || are binary operators. Operator ! is a unary operator. Logical operators follow these conditions:

```
// Logical operations
A = 3; B = 0;
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
A || B // Evaluates to 1. Equivalent to (logical-1 || logical-0)
!A// Evaluates to 0. Equivalent to not(logical-1)
!B// Evaluates to 1. Equivalent to not(logical-0)
// Unknowns
A = 2'b0x; B = 2'b10;
A && B // Evaluates to x. Equivalent to (x && logical 1)
// Expressions
(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true. //
Evaluates to 0 if either is false.
```

1. Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).

2. If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is 01equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition.
3. Logical operators take variables or expressions as operands.
4. Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Relational operators:

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and lessthan-or-equal-to (<=). If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the expression takes a value x. These operators function exactly as the corresponding operators in the C programming language.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx
A <= B // Evaluates to a logical 0
A > B // Evaluates to a logical 1
Y >= X // Evaluates to a logical 1
Y < Z // Evaluates to an x
```

Equality operators:

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==). When used in an expression, equality operators return logical value 1 if true, 0 if false. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length.

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx
A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match) M
!== N // Results in logical 1
```

```
//examples of logical operators
module log_ops1 (a, b, z1, z2, z3);
input [3:0] a, b; output z1, z2, z3;
assign z1 = a && b; assign z2 = a ||
b; assign z3 = !a; endmodule

//test bench for logical operators
module log_ops1_tb; reg [3:0] a, b;
wire z1, z2, z3; initial

$monitor ("z1 = %d, z2 = %d, z3 = %d", z1, z2, z3);

//apply input vectors
initial begin

#0 a = 4'b0110; //nonzero vector
b = 4'b1100; //nonzero vector #5
a = 4'b0101; //nonzero vector b =
4'b0000; //zero vector #5 a =
4'b1000; b = 4'b1001;
```

```
#5 a = 4'b0000; b = 4'b0000;
#5 a = 4'b1111; b = 4'b1111;
#5 $stop; end
//instantiate the module into the test bench log_ops1
inst1 (
.a(a),
.b(b),
.z1(z1),
.z2(z2),
.z3(z3) );
endmodule
```

Bit-wise operator:

Bitwise operators are negation (~), and (&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand.

```
// X = 4'b1010, Y = 4'b1101
// Z = 4'b10x1
~X // Negation. Result is 4'b0101
X & Y // Bitwise and. Result is 4'b1000 X
| Y // Bitwise or. Result is 4'b1111
X ^ Y // Bitwise xor. Result is 4'b0111
X ^~ Y // Bitwise xnor. Result is 4'b1000
X & Z // Result is 4'b10x0
```

It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||. Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.

```
// X = 4'b1010, Y = 4'b0000
X | Y // bitwise operation. Result is 4'b1010
X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
```

```
//dataflow example of bitwise operators
module bitwise2 (a, b, and_rslt, or_rslt, neg_rslt, xor_rslt, xnor_rslt);
//define inputs and outputs input
[7:0] a, b;
output [7:0] and_rslt, or_rslt, neg_rslt, xor_rslt, xnor_rslt;
wire [7:0] a, b;
wire [7:0] and_rslt, or_rslt, neg_rslt, xor_rslt, xnor_rslt;
//define outputs using continuous assignment
assign and_rslt = a & b, //bitwise AND
or_rslt = a | b, //bitwise OR neg_rslt = ~a,
//bitwise negation xor_rslt = a ^ b,
//bitwise exclusive-OR xnor_rslt = a ^~ b;
//bitwise exclusive-NOR endmodule
```

Reduction operator:

Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~). Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.

```
// X = 4'b1010
&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
|X//Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
^X//Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
//A reduction xor or xnor can be used for even or odd parity
//generation of a vector.
```

```
//module to illustrate the use of reduction operators
module reduction2 (a, red_and, red_nand, red_or, red_nor,
red_xor, red_xnor); input [7:0] a;
output red_and, red_nand, red_or, red_nor, red_xor, red_xnor; wire
[7:0] a;
wire red_and, red_nand, red_or, red_nor, red_xor, red_xnor;
assign red_and = &a, //reduction AND red_nand = ~&a,
//reduction NAND red_or = |a, //reduction OR red_nor = ~|a,
//reduction NOR red_xor = ^a, //reduction exclusive-OR
red_xnor = ^~a; //reduction exclusive-NOR endmodule



//test bench for reduction2 module
module reduction2_tb; reg [7:0] a;
wire red_and, red_nand, red_or, red_nor, red_xor, red_xnor; initial
$monitor ("a=%b, red_and=%b, red_nand=%b, red_or=%b,
red_nor=%b, red_xor=%b, red_xnor=%b", a, red_and,
red_nand, red_or, red_nor, red_xor, red_xnor);
//apply input vectors
initial begin
#0 a = 8'b0011_0011;
#10 a = 8'b1101_0011;
#10 a = 8'b0000_0000;
#10 a = 8'b0100_1111;
#10 a = 8'b1111_1111;
#10 a = 8'b0111_1111;
#10 $stop;
end
//instantiate the module into the test bench
reduction2 inst1 ( .a(a),
.red_and(red_and),
.red_nand(red_nand),
.red_or(red_or),
.red_nor(red_nor),
.red_xor(red_xor),
.red_xnor(red_xnor)
```

```
);
endmodule
```

Shift operator:

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<). Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros.

```
// X = 4'b1100
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.
integer a, b, c; //Signed data types a
= 0;
b = -10; // 00111...10110 binary
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

```
//dataflow module to illustrate the shift operators module
shift2 (a, b, a_rslt, b_rslt);
//define inputs and outputs input
[7:0] a, b;
output [7:0] a_rslt, b_rslt; //define
inputs and outputs as wire wire a, b;
wire a_rslt, b_rslt;
//define outputs using continuous assignment
assign a_rslt = a << 2, //multiply by 4
b_rslt = b >> 3; //divide by 8 endmodule
//test bench for shift operators module
module shift2_tb; reg [7:0] a, b;
wire [7:0] a_rslt, b_rslt;
//display variables initial
$monitor ("a = %b, b = %b, a_rslt = %b, b_rslt = %b",
a, b, a_rslt, b_rslt); //apply input vectors initial
begin
#0 a = 8'b0000_0010; //2; a_rslt = 8 b =
8'b0000_1000; //8; b_rslt = 1 #10 a =
8'b0000_0110; //6; a_rslt = 24 b =
8'b0001_1000; //24; b_rslt = 3 #10 a =
8'b0000_1111; //15; a_rslt = 60 b =
8'b0011_1000; //56; b_rslt = 7
#10 a = 8'b1110_0000; //-32; a_rslt = -128 b
= 8'b0000_0011; //3; b_rslt = 0
#10 $stop;
end
```

```
//instantiate the module into the test bench shift2
inst1 (
.a(a),
.b(b),
.a_rslt(a_rslt),
.b_rslt(b_rslt)
);
endmodule
```

Concatenation operator:

The concatenation operator ({, }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result. Concatenations are expressed as operands within braces, with commas separating the operands.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

Replication operator:

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A; reg [1:0]
B, C; reg [2:0]
D;
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

Conditional operator:

The conditional operator(?:) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;

The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated. If the result is x (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same. The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
```

```
// Port declarations from the I/O diagram
output out; input i0, i1, i2, i3; input
s1, s0;
// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ; endmodule
```

Lab Activities:

1. Implement a 4-to-1 Multiplexer using data flow modelling by logical operators.
2. Implement data flow modelling based 4-bit Carry Look Ahead Adder.
3. Design 16-bit comparator using data-flow modelling. Make use of relational operators and continuous assignment statement. Also, write down the test bench to verify the results.

Lab Exercises:

1. A *magnitude comparator* checks if one number is greater than or equal to or less than another number. A 4-bit magnitude comparator takes two 4-bit numbers *A*, and *B* as input. Write the Verilog description of the module *magnitude_comparator*. Instantiate the magnitude comparator inside the stimulus modulus and try out a few combinations of *A* and *B*.

Note: See the corresponding Boolean equations from the following book, Chapter 6, Exercise 2

Detailed readings:

Verilog HDL A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition

Lab Exercise Code

```
`timescale 1ns / 1ps



module magnitude_comparator(Equal, Lesser, Greater, In1, In2 );

input [3:0]In1;
input [3:0]In2;

output reg Equal;
output reg Lesser;
output reg Greater;

always@(In1 or In2)
begin

    if(In1>In2)
    begin
    Greater = 1;
    Lesser = 0;
    Equal = 0;
    end

    else if (In1<In2)
    begin
    Greater = 0;
    Lesser = 1;
    Equal = 0;
    end

    else if(In1==In2)
    begin
    Greater = 0;
    Lesser = 0;
    Equal = 1;
    end

end
endmodule




## This file is a general .xdc for the Nexys4 DDR Rev. C
## To use it in a project:
```

```
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports)
according to the top level signal names in the project

## Clock signal
#set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 }
[get_ports { CLK100MHZ }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0
5} [get_ports {CLK100MHZ}];


##Switches

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 }
[get_ports { In1[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 }
[get_ports { In1[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13   IOSTANDARD LVCMOS33 }
[get_ports { In1[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 }
[get_ports { In1[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17   IOSTANDARD LVCMOS33 }
[get_ports { In2[0] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 }
[get_ports { In2[1] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 }
[get_ports { In2[2] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13   IOSTANDARD LVCMOS33 }
[get_ports { In2[3] }]; #IO_L5N_T0_D07_14 Sch=sw[7]
#set_property -dict { PACKAGE_PIN T8    IOSTANDARD LVCMOS18 }
[get_ports { SW[8] }]; #IO_L24N_T3_34 Sch=sw[8]
#set_property -dict { PACKAGE_PIN U8    IOSTANDARD LVCMOS18 }
[get_ports { SW[9] }]; #IO_25_34 Sch=sw[9]
#set_property -dict { PACKAGE_PIN R16   IOSTANDARD LVCMOS33 }
[get_ports { SW[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
#set_property -dict { PACKAGE_PIN T13   IOSTANDARD LVCMOS33 }
[get_ports { SW[11] }]; #IO_L23P_T3_A03_D19_14 Sch=sw[11]
#set_property -dict { PACKAGE_PIN H6    IOSTANDARD LVCMOS33 }
[get_ports { SW[12] }]; #IO_L24P_T3_35 Sch=sw[12]
#set_property -dict { PACKAGE_PIN U12   IOSTANDARD LVCMOS33 }
[get_ports { SW[13] }]; #IO_L20P_T3_A08_D24_14 Sch=sw[13]
#set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 }
[get_ports { SW[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
#set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 }
[get_ports { SW[15] }]; #IO_L21P_T3_DQS_14 Sch=sw[15]


## LEDs
```

```
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 }
[get_ports { Equal }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 }
[get_ports { Greater }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 }
[get_ports { Lesser }]; #IO_L17N_T2_A25_15 Sch=led[2]
#set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 }
[get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 }
[get_ports { LED[4] }]; #IO_L7P_T1_D09_14 Sch=led[4]
#set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 }
[get_ports { LED[5] }]; #IO_L18N_T2_A11_D27_14 Sch=led[5]
#set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 }
[get_ports { LED[6] }]; #IO_L17P_T2_A14_D30_14 Sch=led[6]
#set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 }
[get_ports { LED[7] }]; #IO_L18P_T2_A12_D28_14 Sch=led[7]
#set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 }
[get_ports { LED[8] }]; #IO_L16N_T2_A15_D31_14 Sch=led[8]
#set_property -dict { PACKAGE_PIN T15   IOSTANDARD LVCMOS33 }
[get_ports { LED[9] }]; #IO_L14N_T2_SRCC_14 Sch=led[9]
#set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 }
[get_ports { LED[10] }]; #IO_L22P_T3_A05_D21_14 Sch=led[10]
#set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 }
[get_ports { LED[11] }]; #IO_L15N_T2_DQS_DOUT_CSO_B_14
Sch=led[11]
#set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 }
[get_ports { LED[12] }]; #IO_L16P_T2_CSI_B_14 Sch=led[12]
#set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 }
[get_ports { LED[13] }]; #IO_L22N_T3_A04_D20_14 Sch=led[13]
#set_property -dict { PACKAGE_PIN V12   IOSTANDARD LVCMOS33 }
[get_ports { LED[14] }]; #IO_L20N_T3_A07_D23_14 Sch=led[14]
#set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 }
[get_ports { LED[15] }]; #IO_L21N_T3_DQS_A06_D22_14 Sch=led[15]

#set_property -dict { PACKAGE_PIN R12   IOSTANDARD LVCMOS33 }
[get_ports { LED16_B }]; #IO_L5P_T0_D06_14 Sch=led16_b
#set_property -dict { PACKAGE_PIN M16   IOSTANDARD LVCMOS33 }
[get_ports { LED16_G }]; #IO_L10P_T1_D14_14 Sch=led16_g
#set_property -dict { PACKAGE_PIN N15   IOSTANDARD LVCMOS33 }
[get_ports { LED16_R }]; #IO_L11P_T1_SRCC_14 Sch=led16_r
#set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 }
[get_ports { LED17_B }]; #IO_L15N_T2_DQS_ADV_B_15 Sch=led17_b
#set_property -dict { PACKAGE_PIN R11   IOSTANDARD LVCMOS33 }
[get_ports { LED17_G }]; #IO_0_14 Sch=led17_g
#set_property -dict { PACKAGE_PIN N16   IOSTANDARD LVCMOS33 }
[get_ports { LED17_R }]; #IO_L11N_T1_SRCC_14 Sch=led17_r


##7 segment display
```

```
#set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 }
[get_ports { CA }]; #IO_L24N_T3_A00_D16_14 Sch=ca
#set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 }
[get_ports { CB }]; #IO_25_14 Sch=cb
#set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 }
[get_ports { CC }]; #IO_25_15 Sch=cc
#set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 }
[get_ports { CD }]; #IO_L17P_T2_A26_15 Sch=cd
#set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 }
[get_ports { CE }]; #IO_L13P_T2_MRCC_14 Sch=ce
#set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 }
[get_ports { CF }]; #IO_L19P_T3_A10_D26_14 Sch=cf
#set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 }
[get_ports { CG }]; #IO_L4P_T0_D04_14 Sch=cg

#set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 }
[get_ports { DP }]; #IO_L19N_T3_A21_VREF_15 Sch=dp

#set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 }
[get_ports { AN[0] }]; #IO_L23P_T3_FOE_B_15 Sch=an[0]
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 }
[get_ports { AN[1] }]; #IO_L23N_T3_FWE_B_15 Sch=an[1]
#set_property -dict { PACKAGE_PIN T9    IOSTANDARD LVCMOS33 }
[get_ports { AN[2] }]; #IO_L24P_T3_A01_D17_14 Sch=an[2]
#set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 }
[get_ports { AN[3] }]; #IO_L19P_T3_A22_15 Sch=an[3]
#set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 }
[get_ports { AN[4] }]; #IO_L8N_T1_D12_14 Sch=an[4]
#set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 }
[get_ports { AN[5] }]; #IO_L14P_T2_SRCC_14 Sch=an[5]
#set_property -dict { PACKAGE_PIN K2    IOSTANDARD LVCMOS33 }
[get_ports { AN[6] }]; #IO_L23P_T3_35 Sch=an[6]
#set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 }
[get_ports { AN[7] }]; #IO_L23N_T3_A02_D18_14 Sch=an[7]


##Buttons

#set_property -dict { PACKAGE_PIN C12   IOSTANDARD LVCMOS33 }
[get_ports { CPU_RESETN }]; #IO_L3P_T0_DQS_AD1P_15 Sch=cpu_resetn

#set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 }
[get_ports { BTNC }]; #IO_L9P_T1_DQS_14 Sch=btnc
#set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 }
[get_ports { BTNU }]; #IO_L4N_T0_D05_14 Sch=btnu
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 }
[get_ports { BTNL }]; #IO_L12P_T1_MRCC_14 Sch=btnl
```

```
#set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 }
[get_ports { BTNR }]; #IO_L10N_T1_D15_14 Sch=btnr
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 }
[get_ports { BTND }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd


##Pmod Headers


##Pmod Header JA

#set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33 }
[get_ports { JA[1] }]; #IO_L20N_T3_A19_15 Sch=ja[1]
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 }
[get_ports { JA[2] }]; #IO_L21N_T3_DQS_A18_15 Sch=ja[2]
#set_property -dict { PACKAGE_PIN E18   IOSTANDARD LVCMOS33 }
[get_ports { JA[3] }]; #IO_L21P_T3_DQS_15 Sch=ja[3]
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 }
[get_ports { JA[4] }]; #IO_L18N_T2_A23_15 Sch=ja[4]
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 }
[get_ports { JA[7] }]; #IO_L16N_T2_A27_15 Sch=ja[7]
#set_property -dict { PACKAGE_PIN E17   IOSTANDARD LVCMOS33 }
[get_ports { JA[8] }]; #IO_L16P_T2_A28_15 Sch=ja[8]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 }
[get_ports { JA[9] }]; #IO_L22N_T3_A16_15 Sch=ja[9]
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 }
[get_ports { JA[10] }]; #IO_L22P_T3_A17_15 Sch=ja[10]


##Pmod Header JB

#set_property -dict { PACKAGE_PIN D14   IOSTANDARD LVCMOS33 }
[get_ports { JB[1] }]; #IO_L1P_T0_AD0P_15 Sch=jb[1]
#set_property -dict { PACKAGE_PIN F16   IOSTANDARD LVCMOS33 }
[get_ports { JB[2] }]; #IO_L14N_T2_SRCC_15 Sch=jb[2]
#set_property -dict { PACKAGE_PIN G16   IOSTANDARD LVCMOS33 }
[get_ports { JB[3] }]; #IO_L13N_T2_MRCC_15 Sch=jb[3]
#set_property -dict { PACKAGE_PIN H14   IOSTANDARD LVCMOS33 }
[get_ports { JB[4] }]; #IO_L15P_T2_DQS_15 Sch=jb[4]
#set_property -dict { PACKAGE_PIN E16   IOSTANDARD LVCMOS33 }
[get_ports { JB[7] }]; #IO_L11N_T1_SRCC_15 Sch=jb[7]
#set_property -dict { PACKAGE_PIN F13   IOSTANDARD LVCMOS33 }
[get_ports { JB[8] }]; #IO_L5P_T0_AD9P_15 Sch=jb[8]
#set_property -dict { PACKAGE_PIN G13   IOSTANDARD LVCMOS33 }
[get_ports { JB[9] }]; #IO_0_15 Sch=jb[9]
#set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 }
[get_ports { JB[10] }]; #IO_L13P_T2_MRCC_15 Sch=jb[10]
```

```
##Pmod Header JC

#set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 }
[get_ports { JC[1] }]; #IO_L23N_T3_35 Sch=jc[1]
#set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 }
[get_ports { JC[2] }]; #IO_L19N_T3_VREF_35 Sch=jc[2]
#set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 }
[get_ports { JC[3] }]; #IO_L22N_T3_35 Sch=jc[3]
#set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 }
[get_ports { JC[4] }]; #IO_L19P_T3_35 Sch=jc[4]
#set_property -dict { PACKAGE_PIN E7    IOSTANDARD LVCMOS33 }
[get_ports { JC[7] }]; #IO_L6P_T0_35 Sch=jc[7]
#set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 }
[get_ports { JC[8] }]; #IO_L22P_T3_35 Sch=jc[8]
#set_property -dict { PACKAGE_PIN J4    IOSTANDARD LVCMOS33 }
[get_ports { JC[9] }]; #IO_L21P_T3_DQS_35 Sch=jc[9]
#set_property -dict { PACKAGE_PIN E6    IOSTANDARD LVCMOS33 }
[get_ports { JC[10] }]; #IO_L5P_T0_AD13P_35 Sch=jc[10]


##Pmod Header JD

#set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 }
[get_ports { JD[1] }]; #IO_L21N_T3_DQS_35 Sch=jd[1]
#set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 }
[get_ports { JD[2] }]; #IO_L17P_T2_35 Sch=jd[2]
#set_property -dict { PACKAGE_PIN G1    IOSTANDARD LVCMOS33 }
[get_ports { JD[3] }]; #IO_L17N_T2_35 Sch=jd[3]
#set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 }
[get_ports { JD[4] }]; #IO_L20N_T3_35 Sch=jd[4]
#set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 }
[get_ports { JD[7] }]; #IO_L15P_T2_DQS_35 Sch=jd[7]
#set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 }
[get_ports { JD[8] }]; #IO_L20P_T3_35 Sch=jd[8]
#set_property -dict { PACKAGE_PIN G2    IOSTANDARD LVCMOS33 }
[get_ports { JD[9] }]; #IO_L15N_T2_DQS_35 Sch=jd[9]
#set_property -dict { PACKAGE_PIN F3    IOSTANDARD LVCMOS33 }
[get_ports { JD[10] }]; #IO_L13N_T2_MRCC_35 Sch=jd[10]


##Pmod Header JXADC

#set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVDS     }
[get_ports { XA_N[1] }]; #IO_L9N_T1_DQS_AD3N_15 Sch=xa_n[1]
#set_property -dict { PACKAGE_PIN A13   IOSTANDARD LVDS     }
[get_ports { XA_P[1] }]; #IO_L9P_T1_DQS_AD3P_15 Sch=xa_p[1]
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVDS     }
[get_ports { XA_N[2] }]; #IO_L8N_T1_AD10N_15 Sch=xa_n[2]
```

```
#set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVDS     }
[get_ports { XA_P[2] }]; #IO_L8P_T1_AD10P_15 Sch=xa_p[2]
#set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVDS     }
[get_ports { XA_N[3] }]; #IO_L7N_T1_AD2N_15 Sch=xa_n[3]
#set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVDS     }
[get_ports { XA_P[3] }]; #IO_L7P_T1_AD2P_15 Sch=xa_p[3]
#set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVDS     }
[get_ports { XA_N[4] }]; #IO_L10N_T1_AD11N_15 Sch=xa_n[4]
#set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVDS     }
[get_ports { XA_P[4] }]; #IO_L10P_T1_AD11P_15 Sch=xa_p[4]


##VGA Connector

#set_property -dict { PACKAGE_PIN A3    IOSTANDARD LVCMOS33 }
[get_ports { VGA_R[0] }]; #IO_L8N_T1_AD14N_35 Sch=vga_r[0]
#set_property -dict { PACKAGE_PIN B4    IOSTANDARD LVCMOS33 }
[get_ports { VGA_R[1] }]; #IO_L7N_T1_AD6N_35 Sch=vga_r[1]
#set_property -dict { PACKAGE_PIN C5    IOSTANDARD LVCMOS33 }
[get_ports { VGA_R[2] }]; #IO_L1N_T0_AD4N_35 Sch=vga_r[2]
#set_property -dict { PACKAGE_PIN A4    IOSTANDARD LVCMOS33 }
[get_ports { VGA_R[3] }]; #IO_L8P_T1_AD14P_35 Sch=vga_r[3]

#set_property -dict { PACKAGE_PIN C6    IOSTANDARD LVCMOS33 }
[get_ports { VGA_G[0] }]; #IO_L1P_T0_AD4P_35 Sch=vga_g[0]
#set_property -dict { PACKAGE_PIN A5    IOSTANDARD LVCMOS33 }
[get_ports { VGA_G[1] }]; #IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
#set_property -dict { PACKAGE_PIN B6    IOSTANDARD LVCMOS33 }
[get_ports { VGA_G[2] }]; #IO_L2N_T0_AD12N_35 Sch=vga_g[2]
#set_property -dict { PACKAGE_PIN A6    IOSTANDARD LVCMOS33 }
[get_ports { VGA_G[3] }]; #IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]

#set_property -dict { PACKAGE_PIN B7    IOSTANDARD LVCMOS33 }
[get_ports { VGA_B[0] }]; #IO_L2P_T0_AD12P_35 Sch=vga_b[0]
#set_property -dict { PACKAGE_PIN C7    IOSTANDARD LVCMOS33 }
[get_ports { VGA_B[1] }]; #IO_L4N_T0_35 Sch=vga_b[1]
#set_property -dict { PACKAGE_PIN D7    IOSTANDARD LVCMOS33 }
[get_ports { VGA_B[2] }]; #IO_L6N_T0_VREF_35 Sch=vga_b[2]
#set_property -dict { PACKAGE_PIN D8    IOSTANDARD LVCMOS33 }
[get_ports { VGA_B[3] }]; #IO_L4P_T0_35 Sch=vga_b[3]

#set_property -dict { PACKAGE_PIN B11   IOSTANDARD LVCMOS33 }
[get_ports { VGA_HS }]; #IO_L4P_T0_15 Sch=vga_hs
#set_property -dict { PACKAGE_PIN B12   IOSTANDARD LVCMOS33 }
[get_ports { VGA_VS }]; #IO_L3N_T0_DQS_AD1N_15 Sch=vga_vs


##Micro SD Connector
```

```
#set_property -dict { PACKAGE_PIN E2    IOSTANDARD LVCMOS33 }
[get_ports { SD_RESET }]; #IO_L14P_T2_SRCC_35 Sch=sd_reset
#set_property -dict { PACKAGE_PIN A1    IOSTANDARD LVCMOS33 }
[get_ports { SD_CD }]; #IO_L9N_T1_DQS_AD7N_35 Sch=sd_cd
#set_property -dict { PACKAGE_PIN B1    IOSTANDARD LVCMOS33 }
[get_ports { SD_SCK }]; #IO_L9P_T1_DQS_AD7P_35 Sch=sd_sck
#set_property -dict { PACKAGE_PIN C1    IOSTANDARD LVCMOS33 }
[get_ports { SD_CMD }]; #IO_L16N_T2_35 Sch=sd_cmd
#set_property -dict { PACKAGE_PIN C2    IOSTANDARD LVCMOS33 }
[get_ports { SD_DAT[0] }]; #IO_L16P_T2_35 Sch=sd_dat[0]
#set_property -dict { PACKAGE_PIN E1    IOSTANDARD LVCMOS33 }
[get_ports { SD_DAT[1] }]; #IO_L18N_T2_35 Sch=sd_dat[1]
#set_property -dict { PACKAGE_PIN F1    IOSTANDARD LVCMOS33 }
[get_ports { SD_DAT[2] }]; #IO_L18P_T2_35 Sch=sd_dat[2]
#set_property -dict { PACKAGE_PIN D2    IOSTANDARD LVCMOS33 }
[get_ports { SD_DAT[3] }]; #IO_L14N_T2_SRCC_35 Sch=sd_dat[3]


##Accelerometer

#set_property -dict { PACKAGE_PIN E15   IOSTANDARD LVCMOS33 }
[get_ports { ACL_MISO }]; #IO_L11P_T1_SRCC_15 Sch=acl_miso
#set_property -dict { PACKAGE_PIN F14   IOSTANDARD LVCMOS33 }
[get_ports { ACL_MOSI }]; #IO_L5N_T0_AD9N_15 Sch=acl_mosi
#set_property -dict { PACKAGE_PIN F15   IOSTANDARD LVCMOS33 }
[get_ports { ACL_SCLK }]; #IO_L14P_T2_SRCC_15 Sch=acl_sclk
#set_property -dict { PACKAGE_PIN D15   IOSTANDARD LVCMOS33 }
[get_ports { ACL_CSN }]; #IO_L12P_T1_MRCC_15 Sch=acl_csn
#set_property -dict { PACKAGE_PIN B13   IOSTANDARD LVCMOS33 }
[get_ports { ACL_INT[1] }]; #IO_L2P_T0_AD8P_15 Sch=acl_int[1]
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 }
[get_ports { ACL_INT[2] }]; #IO_L20P_T3_A20_15 Sch=acl_int[2]


##Temperature Sensor

#set_property -dict { PACKAGE_PIN C14   IOSTANDARD LVCMOS33 }
[get_ports { TMP_SCL }]; #IO_L1N_T0_AD0N_15 Sch=tmp_scl
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 }
[get_ports { TMP_SDA }]; #IO_L12N_T1_MRCC_15 Sch=tmp_sda
#set_property -dict { PACKAGE_PIN D13   IOSTANDARD LVCMOS33 }
[get_ports { TMP_INT }]; #IO_L6N_T0_VREF_15 Sch=tmp_int
#set_property -dict { PACKAGE_PIN B14   IOSTANDARD LVCMOS33 }
[get_ports { TMP_CT }]; #IO_L2N_T0_AD8N_15 Sch=tmp_ct

##Omnidirectional Microphone

#set_property -dict { PACKAGE_PIN J5    IOSTANDARD LVCMOS33 }
[get_ports { M_CLK }]; #IO_25_35 Sch=m_clk
```

```
#set_property -dict { PACKAGE_PIN H5    IOSTANDARD LVCMOS33 }
[get_ports { M_DATA }]; #IO_L24N_T3_35 Sch=m_data
#set_property -dict { PACKAGE_PIN F5    IOSTANDARD LVCMOS33 }
[get_ports { M_LRSEL }]; #IO_0_35 Sch=m_lrsel


##PWM Audio Amplifier

#set_property -dict { PACKAGE_PIN A11   IOSTANDARD LVCMOS33 }
[get_ports { AUD_PWM }]; #IO_L4N_T0_15 Sch=aud_pwm
#set_property -dict { PACKAGE_PIN D12   IOSTANDARD LVCMOS33 }
[get_ports { AUD_SD }]; #IO_L6P_T0_15 Sch=aud_sd


##USB-RS232 Interface

#set_property -dict { PACKAGE_PIN C4    IOSTANDARD LVCMOS33 }
[get_ports { UART_TXD_IN }]; #IO_L7P_T1_AD6P_35 Sch=uart_txd_in
#set_property -dict { PACKAGE_PIN D4    IOSTANDARD LVCMOS33 }
[get_ports { UART_RXD_OUT }]; #IO_L11N_T1_SRCC_35
Sch=uart_rxd_out
#set_property -dict { PACKAGE_PIN D3    IOSTANDARD LVCMOS33 }
[get_ports { UART_CTS }]; #IO_L12N_T1_MRCC_35 Sch=uart_cts
#set_property -dict { PACKAGE_PIN E5    IOSTANDARD LVCMOS33 }
[get_ports { UART_RTS }]; #IO_L5N_T0_AD13N_35 Sch=uart_rts

##USB HID (PS/2)

#set_property -dict { PACKAGE_PIN F4    IOSTANDARD LVCMOS33 }
[get_ports { PS2_CLK }]; #IO_L13P_T2_MRCC_35 Sch=ps2_clk
#set_property -dict { PACKAGE_PIN B2    IOSTANDARD LVCMOS33 }
[get_ports { PS2_DATA }]; #IO_L10N_T1_AD15N_35 Sch=ps2_data


##SMSC Ethernet PHY

#set_property -dict { PACKAGE_PIN C9    IOSTANDARD LVCMOS33 }
[get_ports { ETH_MDC }]; #IO_L11P_T1_SRCC_16 Sch=eth_mdc
#set_property -dict { PACKAGE_PIN A9    IOSTANDARD LVCMOS33 }
[get_ports { ETH_MDIO }]; #IO_L14N_T2_SRCC_16 Sch=eth_mdio
#set_property -dict { PACKAGE_PIN B3    IOSTANDARD LVCMOS33 }
[get_ports { ETH_RSTN }]; #IO_L10P_T1_AD15P_35 Sch=eth_rstn
#set_property -dict { PACKAGE_PIN D9    IOSTANDARD LVCMOS33 }
[get_ports { ETH_CRSDV }]; #IO_L6N_T0_VREF_16 Sch=eth_crsdv
#set_property -dict { PACKAGE_PIN C10   IOSTANDARD LVCMOS33 }
[get_ports { ETH_RXERR }]; #IO_L13N_T2_MRCC_16 Sch=eth_rxerr
#set_property -dict { PACKAGE_PIN C11   IOSTANDARD LVCMOS33 }
[get_ports { ETH_RXD[0] }]; #IO_L13P_T2_MRCC_16 Sch=eth_rxd[0]
```

```
#set_property -dict { PACKAGE_PIN D10   IOSTANDARD LVCMOS33 }
[get_ports { ETH_RXD[1] }]; #IO_L19N_T3_VREF_16 Sch=eth_rxd[1]
#set_property -dict { PACKAGE_PIN B9    IOSTANDARD LVCMOS33 }
[get_ports { ETH_TXEN }]; #IO_L11N_T1_SRCC_16 Sch=eth_txen
#set_property -dict { PACKAGE_PIN A10   IOSTANDARD LVCMOS33 }
[get_ports { ETH_TXD[0] }]; #IO_L14P_T2_SRCC_16 Sch=eth_txd[0]
#set_property -dict { PACKAGE_PIN A8    IOSTANDARD LVCMOS33 }
[get_ports { ETH_TXD[1] }]; #IO_L12N_T1_MRCC_16 Sch=eth_txd[1]
#set_property -dict { PACKAGE_PIN D5    IOSTANDARD LVCMOS33 }
[get_ports { ETH_REFCLK }]; #IO_L11P_T1_SRCC_35 Sch=eth_refclk
#set_property -dict { PACKAGE_PIN B8    IOSTANDARD LVCMOS33 }
[get_ports { ETH_INTN }]; #IO_L12P_T1_MRCC_16 Sch=eth_intn


##Quad SPI Flash

#set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 }
[get_ports { QSPI_DQ[0] }]; #IO_L1P_T0_D00_MOSI_14 Sch=qspi_dq[0]
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 }
[get_ports { QSPI_DQ[1] }]; #IO_L1N_T0_D01_DIN_14 Sch=qspi_dq[1]
#set_property -dict { PACKAGE_PIN L14   IOSTANDARD LVCMOS33 }
[get_ports { QSPI_DQ[2] }]; #IO_L2P_T0_D02_14 Sch=qspi_dq[2]
#set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 }
[get_ports { QSPI_DQ[3] }]; #IO_L2N_T0_D03_14 Sch=qspi_dq[3]
#set_property -dict { PACKAGE_PIN L13   IOSTANDARD LVCMOS33 }
[get_ports { QSPI_CSN }]; #IO_L6P_T0_FCS_B_14 Sch=qspi_csn
```

The End

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#04

Behavioural Modelling for Combinational Circuits

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#04

Behavioural Modelling for Combinational Circuits

Instructor: Dr. Nabeel Siddiqui

(Sajjad Ali 033-18-0037)

Lab Learning Objectives:

After completing this session, student should be able to:

- Understand and apply structured procedure statements such as **always** and **initial.**
- Understand and apply blocking and non-blocking procedural assignments.
- Understand Apply Conditional statements and Multiway branching

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	ϒ Desired output	ϒ Minor mistakes	ϒ Critical mistakes	
2	Timing	ϒ Submitted within the given time	ϒ 1 day late	ϒ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

What is Behavioural Modelling?

Verilog provides designers the ability to describe design functionality in an algorithmic manner. In other words, the designer describes the behaviour of the circuit. Thus, behavioural modelling represents the circuit at a very high level of abstraction. Behavioural Verilog constructs are similar to C language constructs in many ways.

Why we need Behavioural Modelling?

With the increasing complexity of digital design, it has become vitally important to make wise design decisions early in a project. Designers need to be able to evaluate the trade-offs of various architectures and algorithms before they decide on the optimum architecture and algorithm to implement in hardware. In this regard, behavioural modelling helps us to implement and verify the design in the fastest way to make early important decisions.

What are Procedures?

In computer language, the procedure is set of codes used to perform a task or set of tasks, contrary to function which produces an information or returns a value.

If this were a computer code, can be called a procedure. The procedure is a set of codes which are set of instructions to perform certain tasks, such as drawing a square shape in the following example:

```
Repeat the next two steps four times:
Draw a line of length n.
Turn right by 90 degrees.
```

While a function will look like this:

```
VAT equals = value_of_goods_sold * 0.2
Return VAT
```

Functions always return some value when they are called.

Lab Examples: Always and Initial procedures

There are two structured procedure statements in Verilog: always and initial. These statements are the two most basic statements in behavioural modelling. All other behavioural statements can appear only inside these structured procedure statements. Verilog is a concurrent programming language unlike the C programming language, which is sequential in nature. Activity flows in Verilog run in parallel rather than in sequence.

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. Multiple behavioural statements must be grouped, typically using the keywords begin and end. The initial blocks are typically used for initialization, monitoring, waveforms and other processes that must be executed only once during the entire simulation run.

```
assignment ::= variable_lvalue = [ delay_or_event_control ] expression
```

Listing 4.1: initial statement

```
module stimulus;
reg x,y, a,b, m;

initial
       m = 1'b0;          //single statement; does not need to be grouped
initial
begin
       #5  a = 1'b1;      //multiple statements; need to be grouped
       #25 b = 1'b0;
end

initial
begin
       #10 x = 1'b0;
       #25 y = 1'b1;
end

initial
       #50 $finish;
endmodule
```

Always statement:

All behavioural statements inside an always statement constitute an always block. 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 the always statement, the simulation must be halted inside an initial statement. If there is no $stop or $finish statement to halt the simulation, the clock generator will run forever. C programmers might draw an analogy between the always block and an infinite loop. But hardware designers tend to view it as a continuously repeated activity in a digital circuit starting from power on. The activity is stopped only by power off ($finish) or by an interrupt ($stop).

Listing 4.2: always statement

```
module clock_gen (output reg clock);
                                        //Initialize clock at time zero
initial
       clock = 1'b0;
                                   //Toggle clock every half-cycle (time period = 20)
always
       #10 clock = ~clock;
initial
       #1000 $finish;
endmodule
```

Procedural Assignments:

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. These are unlike continuous assignments used in the Dataflow Modelling, where one assignment statement can cause the value of the right-hand-side expression to be continuously placed onto the left-hand-side net.

Blocking Procedural Assignments:

Blocking assignment statements are executed in the order they are specified in a sequential block. **=** operator is used to specify blocking assignments. In the following example, the statement y = 1 is executed only after x = 0 is executed. The behaviour in a particular block is sequential in a ***begin-end*** block if blocking statements are used, because the statements can execute only in sequence. The statement count = count + 1 is executed last.

Listing 4.3: Blocking procedural assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
       x = 0; y = 1; z = 1; //Scalar assignments
       count = 0; //Assignment to integer variables
       reg_a = 16'b0; reg_b = reg_a; //initialize vectors

       reg_a[2] = #15 1'b1; //Bit select assignment with delay
       reg_b[15:13] = #10 {x, y, z}; //Assign concatenation to part select

       count = count + 1; //Assignment to an integer (increment)
end
```

The simulation time at which statements are executed are as follows:

- All statements ***x = 0*** through ***reg_b = reg_a*** are executed at time 0
- Statement ***reg_a[2] = 0*** at time = 15
- Statement ***reg_b[15:13] = {X, y, z}*** at time = 25
- Statement ***count = count + 1*** at time = 25

Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units. Note that for procedural assignments to registers, if the right-hand side has more bits than the register variable, the right-hand side is truncated to match the width of the register variable. The least significant bits are selected and the most significant bits are discarded. If right-hand side has fewer bits, zeros are filled in the most significant bits of the register variable.

Non-blocking procedural 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. Note that this operator has the same symbol as a relational operator, ***less_than_equal_to***. The operator **<=** is interpreted as a relational operator in an expression and as an assignment operator in the context of a non-blocking assignment. In the following example, the statements ***x = 0*** through ***reg_b = reg_a*** are executed sequentially at time 0. Then the three non-blocking assignments are processed at the same simulation time.

Listing 4.4: Nonblocking procedural assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
       x = 0; y = 1; z = 1; //Scalar assignments
       count = 0; //Assignment to integer variables
       reg_a = 16'b0; reg_b = reg_a; //initialize vectors

       reg_a[2] <= #15 1'b1; //Bit select assignment with delay
       reg_b[15:13] <= #10 {x, y, z}; //Assign concatenation to part select

       count <= count + 1; //Assignment to an integer (increment)
end
```

In this example the statements ***x = 0*** through ***reg_b = reg_a*** are executed sequentially at time 0. Then, the three nonblocking assignments are processed at the same simulation time.

1. ***reg_a[2] = 0*** is scheduled to execute after 15 units (i.e., time = 15)
2. ***reg_b[15:13] = {x, y, z}*** is scheduled to execute after 10 time units (i.e., time = 10)
3. ***count = count + 1*** is scheduled to be executed without any delay (i.e., time = 0)

Thus, the simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution. Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.

Applications of non-blocking statements:

Having described the behavior of nonblocking assignments, it is important to understand why they are used in digital design. They are used as a method to model several concurrent data transfers that take place after a common event. Consider the following example where three concurrent data transfers take place at the positive edge of clock.

Listing 4.5: Applications of Nonblocking assignments

```
module Blocking_statements();

reg in1 = 1;
reg in2 = 0;
reg in3 = 1;
reg reg1, reg2, reg3, clock;

initial
clock = 0;
always
#10 clock = ~ clock;

always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1; // The old value of reg1
end

initial
#40 $finish;

endmodule
```

At each positive edge of clock, the following sequence takes place for the nonblocking assignments.

1. A ***read*** operation is performed on each right-hand-side variable, ***in1***, ***in2***, ***in3***, and ***reg1***, at the positive edge of clock. The right-hand-side expressions are evaluated, and the results are stored internally in the simulator.
2. The ***write*** operations to the left-hand-side variables are scheduled to be executed at the time specified by the intra-assignment delay in each assignment, that is, schedule "***write***" to ***reg1*** after 1 time unit, to ***reg2*** at the next negative edge of clock, and to ***reg3*** after 1 time unit.
3. The ***write*** operations are executed at the scheduled time steps. The order in which the write operations are executed is not important because the internally stored right-hand-side expression values are used to assign to the left-hand-side values. For example, note that ***reg3*** is assigned the old value of ***reg1*** that was stored after the read operation, even if the write operation wrote a new value to ***reg1*** before the write operation to ***reg3*** was executed.

Thus, the final values of ***reg1***, ***reg2***, and ***reg3*** are not dependent on the order in which the assignments are processed.

To understand the read and write operations further, consider Listing 4.6, which is intended to swap the values of registers ***a*** and ***b*** at each positive edge of clock, using two concurrent ***always*** blocks.

Listing 4.6: Blocking vs Nonblocking statements

```
module Race_condition();

reg a = 1, b = 0;
reg clock;

initial
clock = 0;
always
#10 clock = ~ clock;

//I11ustration 1: Two concurrent always blocks with blocking
//statements
always @(posedge clock)
a = b;
always @(posedge clock)
b = a;

//I11ustration 2: Two concurrent always blocks with nonblocking
//statements
always @(posedge clock)
a <= b;
always @(posedge clock)
b <= a;

initial
#100 $stop;

endmodule
```

In Listing 4.6, in ***illustration 1***, there is a race condition when blocking statements are used. Either ***a = b*** would be executed before ***b = a***, or vice versa, depending on the simulator implementation. Thus, values of registers ***a*** and ***b*** will not be swapped. Instead, both registers will get the same value (previous value of ***a*** or ***b***), based on the Verilog simulator implementation. However, nonblocking statements used in ***illustration 2*** eliminate the race condition. At the positive edge of clock, the values of all right-hand-side variables are ***"read",*** and the right-hand-side expressions are evaluated and stored in ***temporary*** variables. During the ***write*** operation, the values stored in the temporary variables are assigned to the left-hand-side variables. Separating the ***read*** and ***write*** operations ensures that the values of registers ***a*** and ***b*** are swapped correctly, regardless of the order in which the write operations are performed.

For digital design, use of nonblocking assignments in place of blocking assignments is highly recommended in places where concurrent data transfers take place after a common event. In such cases, blocking assignments can potentially cause race conditions because the final result depends on the order in which the assignments are evaluated. Nonblocking assignments can be used effectively to model concurrent data transfers because the final result is not dependent on the order in which the assignments are evaluated. Typical applications of nonblocking assignments include pipeline modeling and
modeling of several mutually exclusive data transfers.

Timing Controls:

Various behavioral timing control constructs are available in Verilog. In Verilog, if there are no timing control statements, the simulation time does not advance. Timing controls provide a way to specify the simulation time at which procedural statements will execute. There are three methods of timing control: ***delay-based timing control***, ***event-based timing control***, and ***level-sensitive timing control***.

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. We used delay-based timing control statements when writing few modules in the preceding chapters but did not explain them in detail. In this section we will discuss delay-based timing control statements. Delays are specified by the symbol **#**. Syntax for the delay-based timing control statement is shown below.

Listing 4.7: Regular delay control

```
module Regular_delay_control();

// define parameters
parameter latency = 20;
parameter delta = 2;
// define register variables
reg x, y, z, p, q;

initial
begin
        x = 0;                  // no delay control
    #10 y = 1;                  // delay control with a number.
    #latency z = 0;             // Delay control with identifier. Delay of 20 units
    #(latency + delta) p = 1;   // Delay control with expression
    #y x = x + 1;               // Delay control with identifier. Take value of y.
    #(4:5:6) q = 0;             // Minimum, typical and maximum delay values.
end

endmodule
```

Intra-assignment delay control Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator. Such delay specification alters the flow of activity in a different manner. Listing 4.8 shows the contrast between intra-assignment delays and regular delays.

Listing 4.8: Intra-statement delay

```
module Intra-statement_delay();


        // define register variables
reg x, y, z;
        // intra assignment delays
initial
begin
    x = 0; z = 0;
    y = #5 x + z;
        // Take value of x and z at the time=O
        // evalaute x + z and then wait 5 time units to assign value to y
end

endmodule
```

Note the difference between intra-assignment delays and regular delays. Regular delays defer the execution of the entire assignment. Intra-assignment delays compute the right-hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable. Intra-assignment delays are like using regular delays with a temporary variable to store the current
value of a right-hand-side expression.

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. There are four types of event-based timing control: ***regular event control***, ***named event control***, ***event OR control***, and ***level-sensitive timing control***.

Regular event control

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. The keyword **po.edge** is used for a negative transition, as shown in Listing 7.9.

Listing 4.9: Regular Event Control

```
//  d=q is executed whenever the signal clock changes value
@(clock) q = d;
// d=q is executed whenever the signal clock does positive transition
@(posedge clock) q = d;
// d=q is executed whenever the signal clock does positive transition
@(negedge clock) q = d;
// d is evaluated immediately and assinged to q at the postive edge of the clock
q = @(posedge clock) d;
```

Named event control
Verilog provides the capability to *declare* an event and then *trigger* and *recognize* the occurrence of that event (see Listing 4.10). The event does not hold any data. A named event is declared by the keyword **event.** An event is triggered by the symbol ->. The triggering of the event is recognized by the symbol @.

Listing 4.10: Named Event Control

```
// This is an example of a data buffer storing data after the
// last packet of data has arrived.
event received_data; // Define an event called received_data
always @(posedge clock) // check at each positive clock edge
begin
if (last_data_packet) // If this is the last data packet
->received_data; // trigger the event received_data
end
always @(received_data) // Await triggering of event received_data
// When event is triggered, store all four
// packets of received data in data buffer
// use concatenation operator { }
data_buf={data-pkt[O] , data-pkt[l] , data-pkt[2] , data-pkt[3]};

end
```

Event OR control
Sometimes a transition on anyone of multiple signals or events can trigger the execution of a statement or a block of statements. This is expressed as an OR of events or signals. The list of events or signals expressed as an ***OR*** is also known as a ***sensitivity*** list. The keyword **or** is used to specify multiple triggers, as shown in Listing 4.11.

Listing 4.11: Event OR Control

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
// Wait for reset or clock or d to change
begin
if (reset) // if reset signal is high, set q to O.
q = 1 'bO;
else if(clock) // if clock is high, latch input
q = d;
end
```

Level-Sensitive Timing Control
Event control discussed earlier waited for the change of a signal value or the triggering of an event. The symbol @ provided edge-sensitive control. Verilog also allows level-sensitive timing control, that is, the ability to wait for a certain condition to be true before a statement or a block of statements is executed. The keyword ***wait*** is used for level-sensitive constructs.

Listing 4.12: Event OR Control

```
always
wait (count_enable) #20 count count + 1;
```

In the above example, the value of ***count_enable*** is monitored continuously. If ***count_enable*** is ***0***, the statement is not entered. If it is logical 1, the statement ***count = count + 1*** is executed after 20 time units. If ***count_enable*** stays at ***1***, count will be incremented every 20 time units.

Conditional statements:

These statements make decisions based on certain conditions. So ***if*** condition is true, the statement is executed, if false no execution takes place. ***If-else*** conditional statements are available in the Verilog. For one condition, a single ***if*** statement is adequate. For two conditions, if-else combination is used as used in C programming. For multiple conditions, nested ***if-else-if*** constructs are used. The example on the next page showing multiple ways of conditional statements. It is important to note that, else expression is executed when either condition is false (1) or ambiguous (x, z).

Listing 4.13: Conditional statements

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if «expression» true_statement;

//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if «expression» true_statement; else false_statement


//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if «expressionl» true_statementl;
else if «expression2» true_statement2;
else if «expression3» true_statement3;
else default_statement;
```

Multiway branching:

In cases, where there are many conditions, using ***if-else-if*** can be unmanageable, therefore Verilog offers Multiway branching using case statement. The keywords ***case***, ***default*** and ***endcase*** are used in the case statement. Each of statement can be a single statement or block of statement grouped by the ***begin-end*** keywords. The ***case*** statement can be nested.

Listing 4.14: Case statement

```
case (expression)
      alternivel: statementl;
      alternative2: statement2;
      alternative3: statement3;
      default: default_statement; // default is optional
endcase
```

Listing 4.15: ALU control

```
//Execute statements based on the ALU control signal
reg [1:0] alu_control;
case (alu_control)
      2'dO y x + z;
      2'd1 : y = x - z;
      2'd2 : y = X * z;
      default: $display("Invalid ALU control signal");
endcase
```

The ***case*** statement can also act like a many-to-one multiplexer. To understand this, let's implement 4-to-1 multiplexer using case statement.

Listing 4.16: Four to One Multiplexer

```
module mux4_to_1 (out, iO, i1, i2, i3, sl, sO);
// Port declarations from the I/O diagram
      output out;
      input iO, i1, i2, i3;
      input s1. sO;
      reg out;
      always @(sl or sO or iO or i1 or i2 or i3)
      case ({sl, sO}) //Switch based on concatenation of control signals
            2'dO out iO;
            2'd1 out i1;
            2'd2 out i2;
            2'd3 out i3;
            default: $display("Invalid control signals");
      endcase
endmodule
```

Lab Exercises:

1. What is the difference between if-else if- else and case constructs in Verilog?
2. What is the difference between delay-based and event-based timing control?

Detailed readings:

Verilog HDL A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition, chapter 7

Digital Design and Verilog HDL Fundamentals by Joseph Cavanagh, chapter 4

Exercise

1. What is the difference between if-else if- else and case constructs in Verilog?

Ans: Since if-else infers priority, it should be used when more than one input condition could occur. Using case, one the other hand, is appropriate when the inputs are mutually exclusive.

2. What is the difference between delay-based and event-based timing control?

Ans: The **delay timing** control is a way of adding a delay between when the simulator encounters the statement and when it executes.

The **event timing** expression allows the statement to be delayed until the occurrence of some simulation event, which can change of value on a net or variable or an explicitly named event triggered in another procedure.

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#05

Behavioural Modelling for Sequential Circuits - I

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#05

Behavioural Modelling for Sequential Circuits - I

Instructor: Dr. Nabeel Siddiqui

Sajjad ALi

033-18-0037

Lab Learning Objectives:

After completing this session, student should be able to:

- Implement basic Flip Flops and Registers using behavioural modelling

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	ϒ Desired output	ϒ Minor mistakes	ϒ Critical mistakes	
2	Timing	ϒ Submitted within the given time	ϒ 1 day late	ϒ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Learning Objectives:

After completing this session, student should be able to:

- Implement basic Flip Flops and Registers using behavioural modelling

Lab Hardware and Software Required:

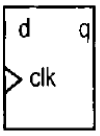
1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

A sequential circuit is a circuit with ***memory,*** which forms the ***internal state*** of the circuit. Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The ***synchronous design methodology*** is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. This methodology is the most important principle in developing a large, complex digital system and is the foundation of most synthesis, verification, and testing algorithms.

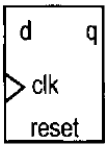
Lab Examples: D Flip Flop

D Flip Flop is the most basic storage component of the sequential circuit design.



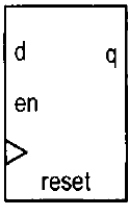
clk	q*
0	q
1	q
↓	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(b) D FF with asynchronous reset



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

(c) D FF with synchronous enable

We consider three types of D FFs:

- D FF without asynchronous reset
- D FF with asynchronous reset
- D FF with synchronous enable

Listing 5.1: Verilog Code for D FF without asynchronous reset

```
module d_ff
(
input wire clk,
input wire d ,
output reg q
);
// body
always @(posedge clk)
       q <= d;
endmodule
```

The rising edge is expressed by the **posedge clk** event in the sensitivity list. The **posedge** (for "positive edge") keyword specifies the direction of the clk signal changing toward 1. It indicates that the always block is activated only at the rising edge of the **clk** signal, a condition reflecting the characteristics of an edge-triggered FF. Note that the **d** signal is not included in the sensitive list. This is consistent with the fact that the **d** signal is sampled only at the rising edge of the **clk** signal, and a change in its value does not trigger any immediate response.

Listing 5.2: Verilog Code for D FF with asynchronous reset

```
module d_ff_reset
(
input wire clk, reset,
input wire d ,
output reg q
);
// body
always @(posedge clk, posedge reset)
       if (reset)
              q <= 1'b0;
       else
              q <= d;
endmodule
```

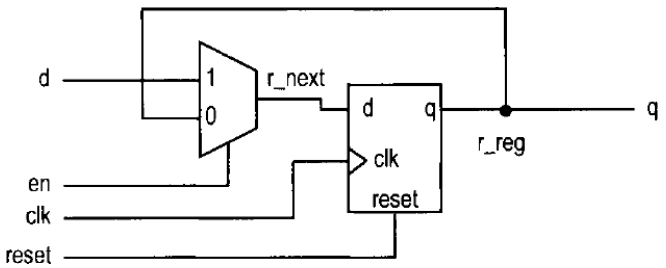
Note that the **posedge reset** event is also included in the sensitivity list and its value is checked first in the **if** statement. The **q** signal is cleared to 0 if it is asserted and its operation is independent of the **clk** signal. The signal clears the D FF to 0 any time and is not controlled by the clock signal. It actually has a higher priority than the regularly sampled input. Using an asynchronous **reset** signal violates the synchronous design methodology and thus should be avoided in normal operation. Its major application is to perform system initialization.

A D FF may include an additional control signal, **en,** to enable the FF to sample the input value. Note that the **en** signal is examined only at the rising edge of the clock and thus is synchronous. If it is not asserted, the FF keeps its previous value.

Listing 5.3: One-segment Verilog coding style for D FF with synchronous enable

```
module d_ff_en_1seg
(
input wire clk, reset,
input wire d ,
output reg q
);
// body
always @(posedge clk, posedge reset)
       if (reset)
              q <= 1'b0;
       if else (en)
              q <= d;
endmodule
```

Note that there is no **else** branch after the second **if** statement. According to Verilog definition, a variable keeps its previous value if it is not assigned. If **en** is 0, **q** keeps its previous value. Thus, omission of the else branch describes the desired behavior of this FF. Since the enable signal is synchronous, this circuit can be constructed by a regular D FF and simple next-state logic.



Listing 5.4: Two-segment Verilog coding style for D FF with synchronous enable

```
module d_ff_en_2seg
(
input wire clk, reset,
input wire d ,
output reg q
);

// signal declaration
reg r_reg, r_next;

// DFF
always @(posedge clk, posedge reset)
       if (reset)
              r_next <= 1'b0;
       else
              r_next <= r_reg;

// next_state logic
always@(*)
       if (en)
              r_next = d;
       else
              r_next = r_reg;
```

```
// output logic
always@(*)
q = r_reg;

endmodule
```

For clarity, we use suffixes **_next** and **_reg** to emphasize the next input value and the registered output of an FF. They are connected to the **d** and **q** signals of a D FF.

Lab Examples: Register

A register is a collection of D FFs that are controlled by the same clock and reset signals. Like a D FF, a register can have an optional asynchronous reset signal and a synchronous enable signal. The code is identical to that of a D FF except that the array data type is needed for the relevant input and output signals. For example, an 8-bit register with asynchronous reset.

Listing 5.5: Verilog Code for D FF with asynchronous reset

```
module reg_reset
(
input wire clk, reset,
input wire [7:0] d ,
output reg [7:0] q
);
// body
always @(posedge clk, posedge reset)
       if (reset)
              q <= 1'b0;
       else
              q <= d;
endmodule
```

Lab Examples: Parameterised Register

A register file is a collection of registers with one input port and one or more output ports. The write address signal, **w_addr**, specifies where to store data, and the read address signal, **r_addr**, specifies where to retrieve data. The register file is generally used as fast, temporary storage. The code for a parameterized 2^w -by-B register file is shown below. Two parameters are defined in this design: W specifies the number of address bits, which implies that there are 2^W words in the file, and B specifies the number of bits in a word.

Listing 5.6: Verilog Code for parameterized register

```
module para_reg
#(
parameter B = 8, // number of bits
          W = 2 // number of address bits
)
(
input wire clk,
input wire wr_en,
input wire [W-1:0] w_addr, r_addr,
input wire [B-1:0] w_data,
output wire [B-1:0] r_data
);
```

```
// Signal declaration
reg [B-1:0] array_reg [2**W-1:0];

// body
always @(posedge clk)
       if (wr_en)
              array_reg[w_addr] <= w_data;

// read operation
assign r_data = array_reg[r_addr];

endmodule
```

The code includes several new features. First, a two-dimensional array data type is defined, as in

reg [B-1:0] array_reg [2**W-1:0];

It indicates that the **array_reg** variable is an array of **[2**W-1:0]** elements and each element is with the data type of **reg** [B-1:0]. Second, a signal is used as an index to access an element in the array, as in **array_reg [w_addr]**. Although the description is very abstract, Xilinx software recognizes this language construct and can derive the correct implementation accordingly. The **array_reg** [….] = and = **array_reg** [….] statements infer decoding and multiplexing logic, respectively.

Some applications may need to retrieve multiple data words at the same time. This can be done by adding an additional read port:

r_data2 = **array_reg [r_addr_2];**

Lab Examples: Shift Registers

Free-running shift register: This register shifts its contents to the left or right by one position in each clock cycle. There is no other control signal.

Listing 5.6: Verilog Code for free running shift register

```
module free_run_shift_reg #(parameter N=8)
(
input wire clk, reset,
input wire s_in,
output wire s_out
);


// signal declaration
reg [N-1:0] r_reg;
wire [N-1:0] r_next;

// body or register
always@(posedge clk, posedge reset)
    if(reset)
        r_reg <= 0;
    else
        r_reg <= r_next;

// next_state logic
assign r_next= {s_in, r_next[N-1:1]};

// output logic
assign s_out = r_reg[0];

endmdule
```

The next-state logic is a 1-bit shifter, which shifts **r_reg** right one position and inserts the serial input, **s_in**, to the MSB. Since the 1-bit shifter involves only reconnection of the input and output signals, no real logic is needed.

Universal shift register: A universal shift register can load parallel data, shift its content left or right, or remain in the same state. It can perform parallel-to-serial operation (first loading parallel input and then shifting) or serial-to-parallel operation (first shifting and then retrieving parallel output). The desired operation is specified by a 2-bit control signal, **ctrl**.

Listing 5.7: Verilog Code for universal shift register

```
module univ_shift_reg #(parameter N=8)
(
input wire clk, reset,
input wire [1:0] ctrl,
input wire [N-1:0] d,
output wire [N-1:0] q
);

// signal declaration
reg [N-1:0] r_reg, r_next;

// body/register
always@(posedge clk, posedge reset)
    if(reset)
        r_reg <= 0;
    else
        r_reg <= r_next;


// next_state logic
always@(*)
    case(ctrl)
        2'b00 : r_next = r_reg; // no operation
        2'b00 : r_next = {r_reg[N-2:0], d[0]}; // shift left
        2'b00 : r_next = {d[N-1], r_reg[N-1:1]}; // shift right
        default: r_next = d; // load
    endcase

// output logic
assign q = r_reg;

endmodule
```

The next-state logic uses a 4-to-1 multiplexer to select the desired next value of the register. Note that the LSB and MSB of d (i.e., **d[0]** and **d[N-1]**) are used as serial input for the shift-left and shift-right operations, respectively.

Lab Exercises:

1. What is Linear Feedback Shift Register? Implement behavioural model of the LFSR and verify the design using test bench. Submit the Verilog codes and functional simulation waveforms.

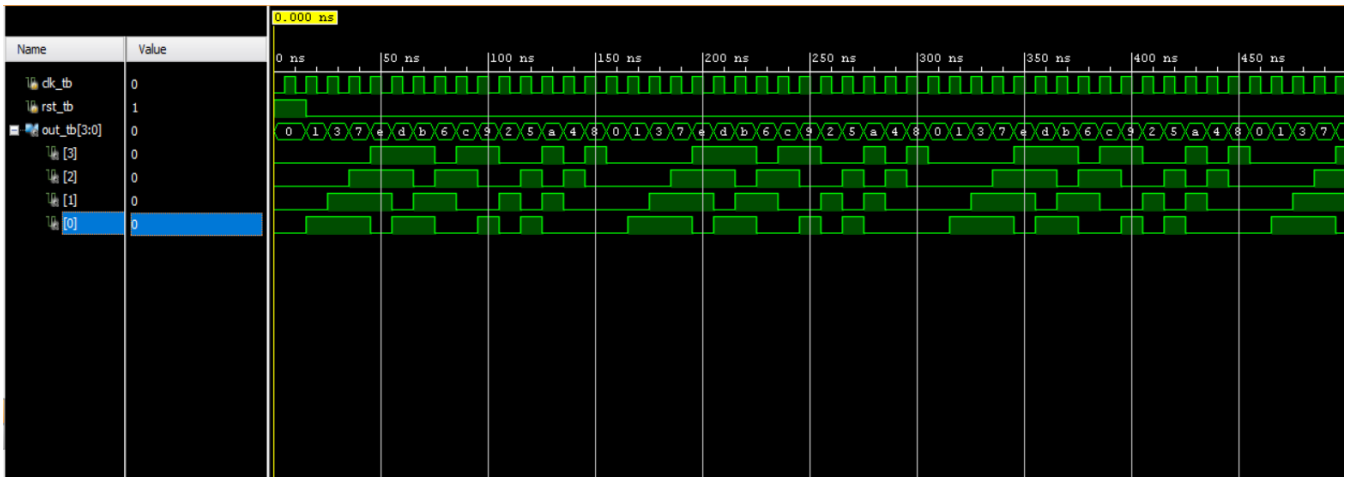
   `timescale 1ns / 1ps

   module lfsr (out, clk, rst);
   output reg [3:0] out; input
   clk, rst;
   wire feedback;
   assign feedback = ~(out[3] ^ out[2]);
   always @(posedge clk, posedge rst)
   begin
   if (rst)
   out = 4'b0; else
   out = {out[2:0],feedback}; end

endmodule

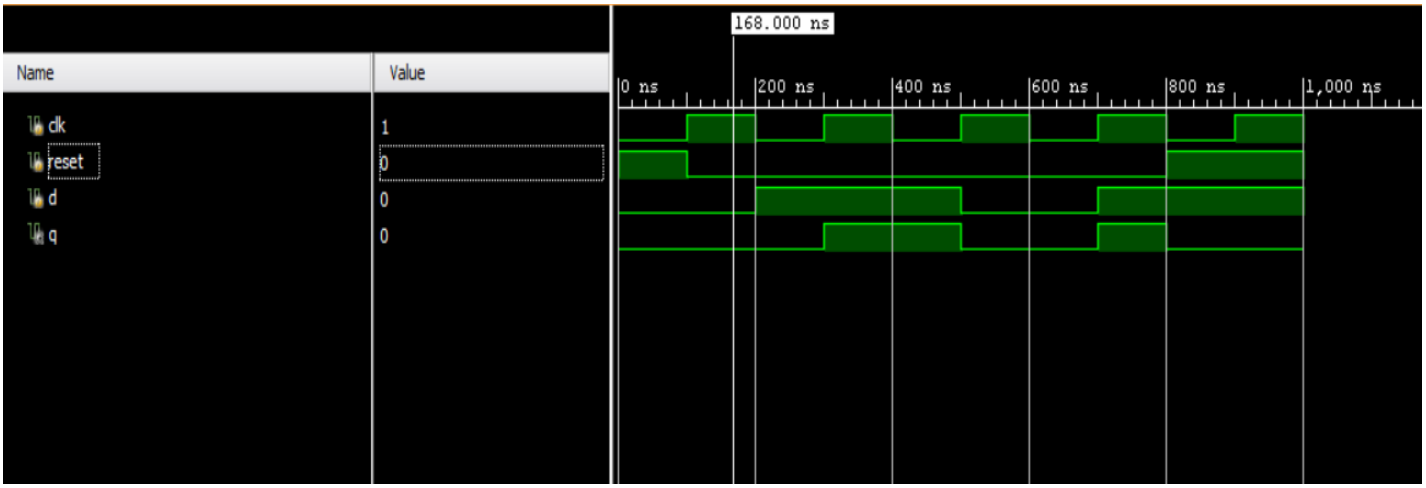
// Test bench File

```
`timescale 1ns / 1ps
module lfsr_tb(); reg
clk_tb;
reg rst_tb;
wire [3:0] out_tb;

lfsr DUT(out_tb,clk_tb,rst_tb); endmodule

initial
begin
clk_tb = 0;
rst_tb = 1;
#15;

rst_tb = 0;
#200;
end

always
begin
#5;
clk_tb = ~ clk_tb; end
end
endmodule
```



2. Verify the functionality of all example designs using Verilog testbench. Submit the Verilog codes and functional simulation waveforms.

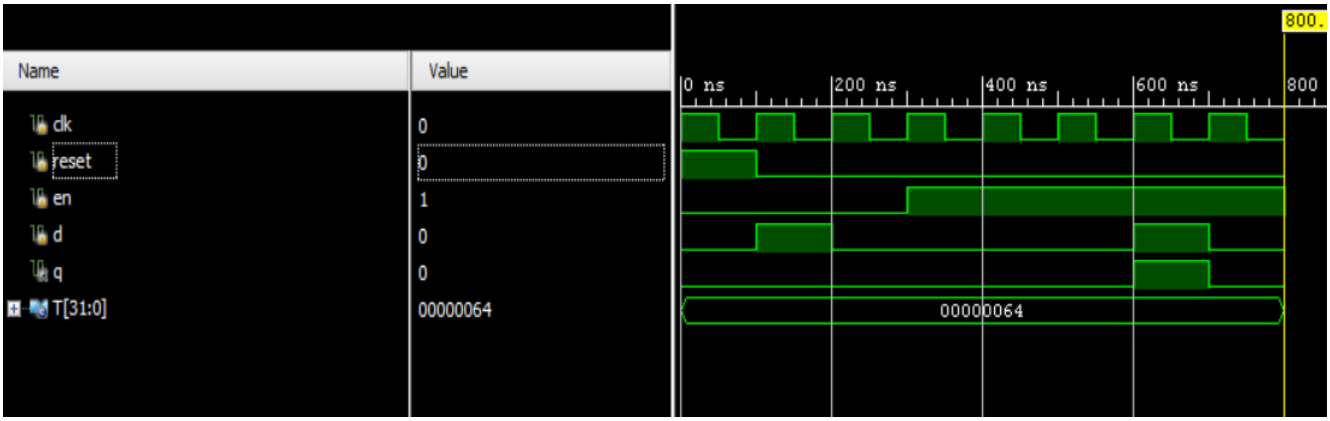
```
`timescale 1ns / 1ps
module
dff_without_async_reset_
tb(); reg clk;
reg d; wire q;
dff_without_async_reset
uut(.clk(clk), .d(d), .q(q));
initial
begin
d=0;
clk=0; end
initial
begin
#100 d=1'b0;
#100 d=1'b1;
#200 d=1'b0;
#300 d=1'b1;
#500 d=1'b0;
#600 d=1'b1;
#100 $stop; end
always #100 clk = ~ clk;

endmodule
```



D-FF Sync with enable – Test Bench

```
`timescale 1ns / 1ps
module
dff_with_sync_enable_tb(
);
localparam T=100; reg
clk,reset;
reg en; reg d; wire q;
dff_with_sync_enable
uut(.clk(clk), .reset(reset),
.en(en), .d(d), .q(q));
initial
```

```
begin
clk =0; d=0;
en=0; reset =1;
#100 reset = 0; end
initial
begin
#100 d=1;
#100 d =0;
#100 en =1;
#300 d=1;
#100 d=0;
#100 $stop; end
always begin
clk = 1; #(T/2);
clk = 0; #(T/2);
end endmodule
```



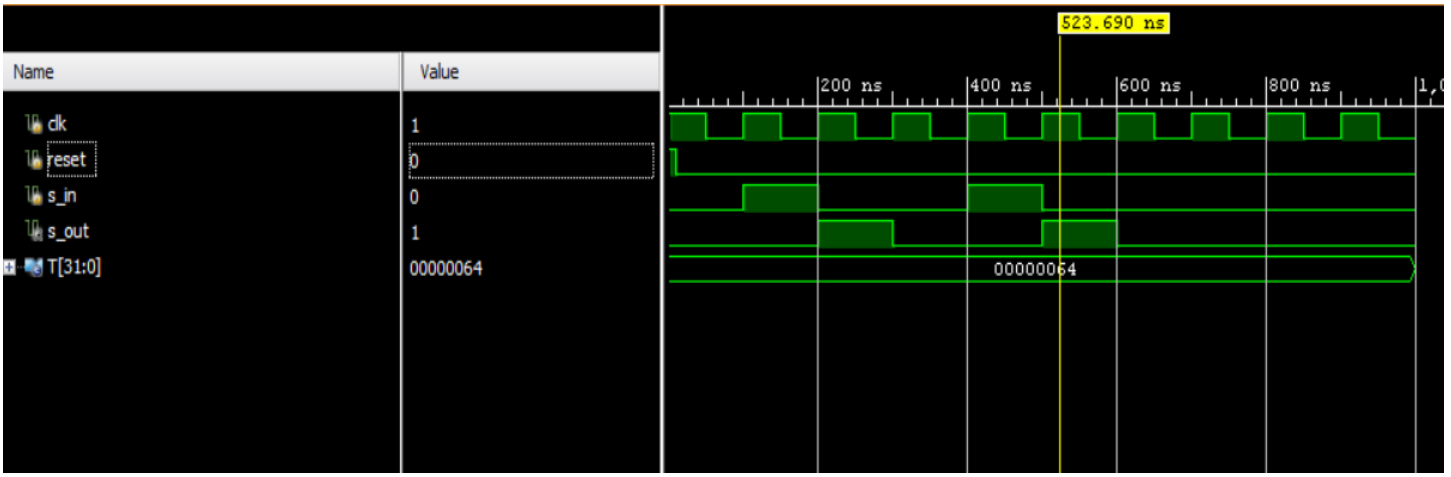
Register Reset – Test Bench

```
`timescale 1ns / 1ps
module reg_reset_tb();

    localparam T
    =100;reg clk,reset;
    reg [7:0] d;
    wire [7:0] q;

    reg_reset

    uut(clk,reset,d,q);initial
        begin
            clk =0;
            d=3'B000;
            reset = 1;
            #100 reset =
        0;end
    initial
        begin
        #100 d =3'b000;
        #100 d =3'b001;
        #100 d =3'b010;
        #100 d =3'b011;
```

```
#100 d =3'b100;
#100 d =3'b101;
#100 d =3'b110;
#100 d =3'b111;
    end
    always
    begin
    clk = 1; # (T/2) ;
    clk = 0; # (T/2) ;
    end
    endmodule
```

Free- running shift register – Test Bench

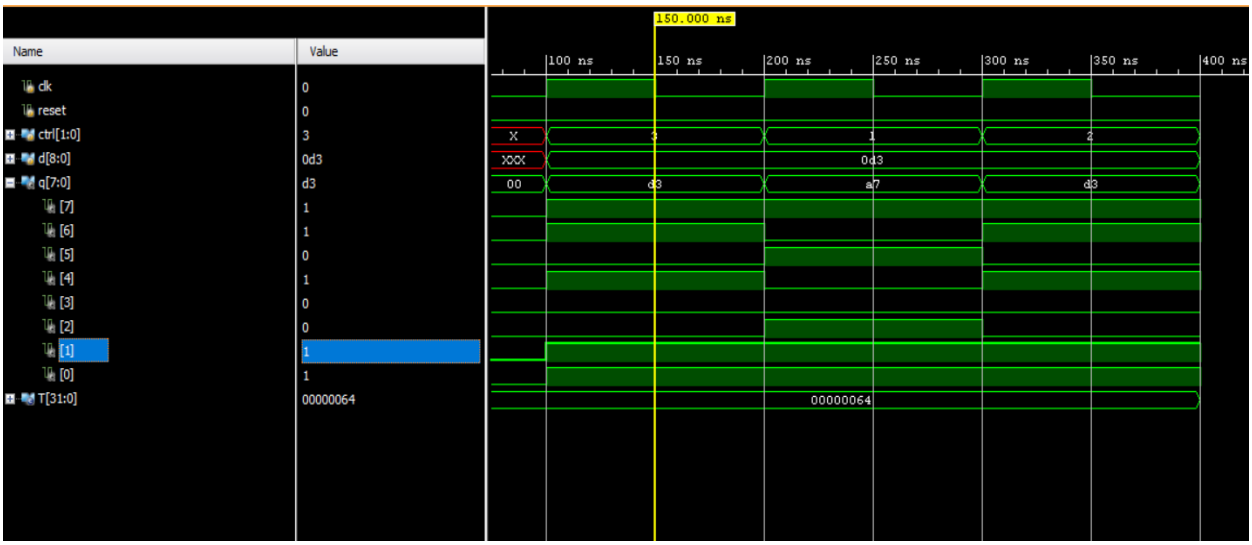
```
`timescale 1ns / 1ps
module free_run_shift_reg_tb();localparam T
   = 100;
reg clk, reset;reg s_in;
wire s_out;
free_run_shift_reg uut(.N(8),.clk(clk),.reset(reset),.s_in(s_in),.s_out(s_out)); initial
begin
clk =0; s_in=0; reset = 1;
#10 reset =0; end
initial
begin
#100 s_in = 1;
#100 s_in = 0;
#200 s_in = 1;
#100 s_in =0;
#500 $stop;

end always
begin
clk = 1; # (T/2);
clk =0; # (T/2);
end

endmodule
```



Universal Shift Register – Test Bench

```
`timescale 1ns / 1ps

    module

      univ_shift_reg_tb();

      localparam T = 100;
      reg clk,
      reset;reg
      [1:0] ctrl;
      reg [8:0] d;
      wire [7:0] q;

      univ_shift_reg

      uut(clk,reset,ctrl,d,q);initial
         begin
            clk = 0;
            reset = 1;
            # 10 reset = 0;
         end

      initial
         begin
            #100;
            d=8'b11010011
            ;
            ctrl = 2'b11;

            #100;
            ctrl=2'b01;

            #100;
            ctrl=2'b10;

            #100;
         $stop;
            end
```

```
always
   begin
      clk = 1;
      # (T/2);
      clk = 0;
      # (T/2);
      end
      endmodule
```



Detailed readings:

Verilog HDL: A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition

Digital Design and Verilog HDL Fundamentals by Joseph Cavanagh

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#06

Behavioural Modelling for Sequential Circuits - II

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#06

Behavioural Modelling for Sequential Circuits - II

Instructor: Dr. Nabeel Siddiqui

Sajjad ALi

033-18-0037

Lab Learning Objectives:

After completing this session, student should be able to:

- Implement binary counters using Verilog HDL behavioural modelling

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	Desired output	Minor mistakes	Critical mistakes	
2	Timing	Submitted within the given time	1 day late	More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

A sequential circuit is a circuit with **memory,** which forms the **internal state** of the circuit. Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The **synchronous design methodology** is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. This methodology is the most important principle in developing a large, complex digital system and is the foundation of most synthesis, verification, and testing algorithms.

Lab Examples: Binary Counters

Free-running binary counter: A free-running binary counter circulates through a binary sequence repeatedly. For example, a 4-bit binary counter counts from "**0000**", "**0001**''. . . to "**1111**'' and wraps around.

Listing 6.1: Verilog Code for free running binary counter

```
module free_run_bin_counter
#(parameter N=8)(       input
clk, reset,       output wire
max_tick,       output wire
[N-1:0] q);

// signal declaration
reg [N-1:0] r_reg; wire
[N-1:0] r_next;

// body or register
always@(posedge clk, posedge reset )
if(reset)          r_reg <= 0;
else
        r_reg <= r_next;

// next_state_logic assign
r_next = r_reg + 1;

// output logic assign
q = r_reg;
assign max_tick = (r_reg == 2**N-1) ? 1'b1 : 1'b0; endmodule
```

The next-state logic is an incrementor, which adds 1 to the register's current value. By definition of the **+** operator, the addition implicitly wraps around after the **r_reg** reaches **"1…1".** The circuit also consists of an output status signal, **max-tick,** which is asserted when the counter reaches the maximal value, **"1…1"** (which is equal to **2N -1**).

Universal binary counter A universal binary counter is more versatile. It can count up or down, pause, be loaded with a specific value, or be synchronously cleared. Note the difference between the reset and syn_clr signals. The former is asynchronous and should only be used for system initialization. The latter is sampled at the rising edge of the clock and can be used in normal synchronous design.

syn_clr	load	en	up	q*	Operation
1	–	–	–	00···00	synchronous clear
0	1	–	–	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q−1	count down
0	0	0	–	q	pause

Listing 6.2: Verilog Code for Universal Binary Counter

```
 module Uni_bin_counter#(parameter N=8)
( input wire clk, reset,          input
wire syn_clr, load, en, up,          input
wire [N-1:0] d,          output wire
max_tick, min_tick,          output wire
[N-1:0] q);
             // signal
declaration  reg [N-1:0]
r_reg, r_next;

// body or register
always@(posedge clk, posedge reset)
if(reset)         r_reg <= 0;
else
        r_reg <= r_next;

// next_state logic  always@*
if(syn_clr)         r_next =
0;     else if (load)
r_next = d;     else if (en &
up)         r_next = r_next +
1;     else if (en & ~up)
r_next = r_next - 1;     else
        r_next = r_reg;
         // output
logic assign q =
r_next;
assign min_tick = (r_reg == 0) ? 1'b1 : 1'b0; assign
max_tick = (r_reg == 2**N-1) ? 1'b1 : 1'b0;
 endmodule
```

The next-state logic follows the functional table and is described by an always block, which contains an **if** statement to prioritize the desired operations.

Mod-m counter A mod-m counter counts from 0 to m-1 and wraps around and is also called truncated counter. A parameterized mod-m counter is shown in Listing 6.3. It has two parameters: M, which specifies the limit, m; and N, which specifies the number of bits needed and should be equal to $\lceil \log_2 M \rceil$.

Listing 6.3: Verilog code for the Mode-m counter

```
 module mod_m_counter#(parameter N=4, M=10)
(input wire clk, reset,
input wire max_tick,
output wire [N-1:0] q
                    );

// signal declaration
reg [N-1:0] r_reg; wire
[N-1:0] r_next;

// body or register
always@(posedge clk, posedge reset)
if(reset)          r_reg <= 0;
else
        r_reg <= r_next;

// next_state logic
assign r_next = (r_reg == M-1) ? 1'b0 : (r_reg + 1);

// output logic assign
q = r_reg;
assign max_tick = (r_reg == 2**M-1) ? 1'b1 : 1'b0;
 endmodule
```

The next-state logic is constructed by a conditional operator. If the counter reaches **M-1**, the new value is cleared to **0**. Otherwise, it is incremented by **1**. Inclusion of the **N** parameter in the code is somewhat redundant since its value depends on **M**. A more elegant way is to define a function that calculates **N** from **M** automatically.

Lab Examples: Testbench for sequential circuits

A testbench is a program that mimics a physical lab bench. In this section, we illustrate the construction of a simple testbench for the previous universal binary counter. It can serve as a template for other sequential circuits.

Listing 6.4: Testbench for universal binary counter

```
`timescale 1ns / 1ps
// 'timescale directive specifies that
// simulation time unit is 1ns and
// simulator timestep is 10 ps

module Uni_bin_counter_tb();

// signal declaration  localparam T
= 20; // clock period  reg clk,
reset; reg syn_clr, load, en, up;
```

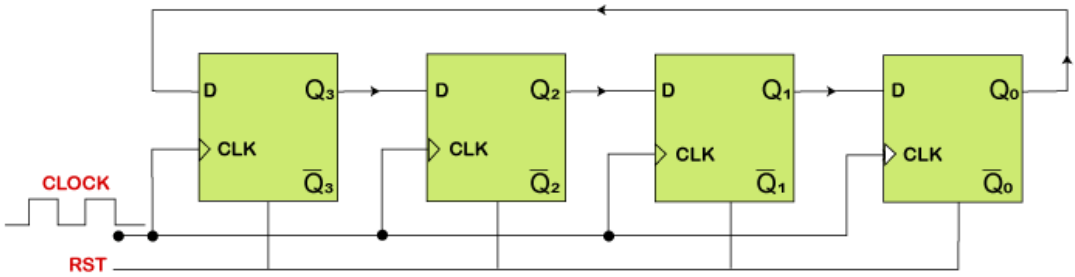
```
reg [2-1:0] d; wire max_tick,
min_tick; wire [2-1:0] q;

// dut instantiation
Uni_bin_counter uut (
                        .clk(clk),
                        .reset(reset),
                        .syn_clr(syn_clr),
                        .load(load),
                        .en(en),
                        .up(up),
                        .d(d),
                        .max_tick(max_tick),
                        .min_tick(min_tick),
                        .q(q)
                    );



// clock generation
initial     clk =
1'b0; always
    #(T/2) clk  = ~ clk;

// reset for the first half cycle
initial begin     reset = 1'b1;
#(T/2) reset = 1'b0; end

// other signals
initial begin
// ==Initial Input==
syn_clr = 1'b0;
load = 1'b0;     en =
1'b0;     up = 1'b1;
d = 3'b000;
    @(negedge reset); // wait reset to de-assert
    @(negedge clk);   // wait for one clock
     // ==test load==      load =
1'b1; // assert load      d =
3'b011;
    @(negedge clk); // wait for one clock
load = 1'b0;  // de-assert load
    repeat(2) @(negedge clk); // wait for two clocks



// ==test sync_clear==
    syn_clr = 1'b1;  // assert clear
@(negedge clk);   // wait for one clock
syn_clr = 1'b0;  // de-assert clear




// ==test up-counter and pause==
en = 1'b1;      up = 1'b1;
     repeat(10) @(negedge clk);   // wait for 10 clocks
en = 1'b0;    // pause
     repeat(10) @(negedge clk);   // wait for 10 clocks
en = 1'b1;
     repeat(2) @(negedge clk);   // wait for 10 clocks

// ==test down counter==
up = 1'b0;
    repeat(10) @(negedge clk);
```

```
 // ==wait statement==      wait(q==2);
// wait untill q =2          @(negedge
clk);      up = 1'b1;

// continue until min_tick becomes 1
@(negedge clk);
wait(min_tick);      @(negedge clk);
up = 1'b0;

// ==absolute delay==
#(4*T); // wait for 80 ns
en = 1'b0;
    #(4*T); // wait for 80 ns


// ==Stop simulation==
    $stop;
end
endmodule
```

Lab Exercises:

1. What is Ring Counter? Explain?

Ans:- A ring counter is a digital circuit with a series of flip flops connected in a feedback manner. Ring Counter is composed of Shift Registers. The data pattern will re-circulate as long as clock pulses are applied. The circuit is a special type of shift register where the last flip flop's output is fed back to the input of the first flip flop. When the circuit is reset, except one of the flip flop output, all others are made zero. For the n-flip flop ring counter, we have a *MOD-n* counter. That means the counter has n different states. For example, if we take a 4-bit Ring Counter, then the data pattern will repeat every four clock pulses. If the pattern is 1000, it will generate 0100, 0010, 0001, 1000, etc.



Detailed readings:

Verilog HDL: A Guide to Digital Design and Synthesis by Samir Palnitkar, 2nd Edition FPGA prototyping using Verilog examples by Pong Chu

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#07

Vivado Post-synthesis and Post-implementation timing simulation

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#07

Vivado Post-synthesis and Post-implementation timing simulation

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali

Lab Learning Objectives:

After completing this session, student should be able to:

- Create a Vivado project sourcing HDL model(s) and targeting a specific FPGA device located on the Nexys4 DDR.
- Use the provided Xilinx Design Constraint (XDC) file to constrain the pin locations
- Simulate the design using the Vivado simulator
- Synthesize and implement the design
- Perform post-synthesis timing simulation
- Perform post-implementation timing simulation
- Generate the bitstream
- Configure the FPGA using the generated bitstream and verify the functionality

Lab Report Rubrics					**Total Marks**
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	☐ Desired output	☐ Minor mistakes	☐ Critical mistakes	
2	Timing	☐ Submitted within the given time	☐ 1 day late	☐ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. Xilinx Vivado 2016.2
2. Digilent NEXYS 4 DDR FPGA Board
3. Desktop/Laptop Computer

Background Theory:

This lab guides you through the process of using Vivado IDE to create a simple HDL design targeting the Nexys4 DDR boards. You will simulate, synthesize, and implement the design with default settings. Finally, you will generate the bitstream and download it in to the hardware to verify the design functionality. This lab is broken into steps that consist of general overview statements providing information on the detailed instructions that follow. Follow these detailed instructions to progress through the lab.

Procedure

Design Description

The design consists of some inputs directly connected to the corresponding output LEDs. Other inputs are logically operated on before the results are output on the remaining LEDs as shown in **Figure 1**.

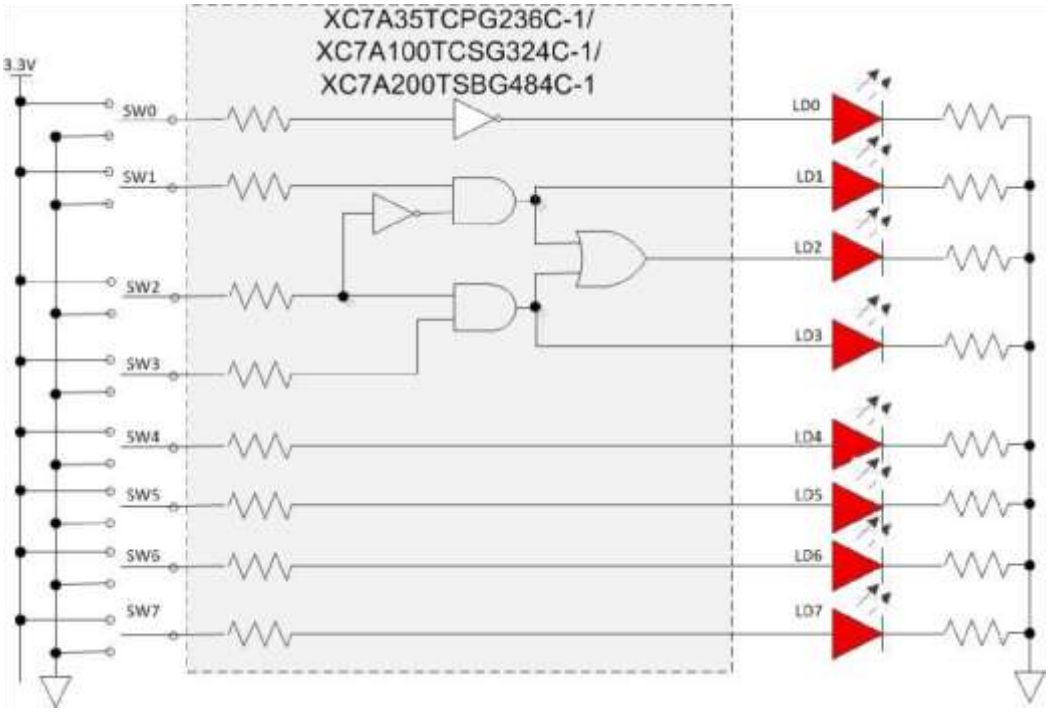
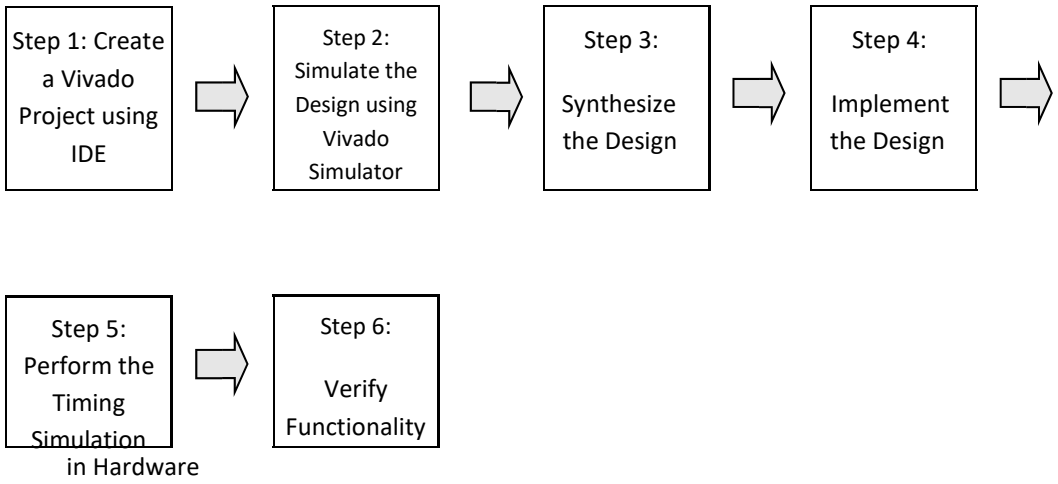


Figure 1. The Completed Design

General Flow:



Board support for the Basys3, Nexys4 DDR, Nexys Video is not included in Vivado 2016.2 by default. The relevant zip files need to be extracted and saved to: {Vivado installation}\data\boards\board_files\.

These files can be downloaded either from the Digilent, Inc. webpage (https://reference.digilentinc.com/vivado/boardfiles2015) or the XUP webpage (http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-fpga-design-flow.html) where this material is also hosted.

Create a Vivado Project using IDE Step 1

1-1. Launch Vivado and create a project targeting **XC7A100TCSG324-1 (Nexys4 DDR)** and using the Verilog HDL. Use the provided lab1.v and lab1_<board>.xdc files from the {sources}*lab1* directory.

1-1-1. Open Vivado by selecting **Start > All Programs > Xilinx Design Tools > Vivado 2016.2 > Vivado 2016.2**

1-1-2. Click **Create New Project** to start the wizard. You will see *Create A New Vivado Project* dialog box. Click **Next**.

1-1-3. Click the Browse button of the *Project location* field of the **New Project** form, browse to **{labs}**, and click **Select**.

1-1-4. Enter **lab1** in the *Project name* field. Make sure that the *Create Project Subdirectory* box is checked. Click **Next**.

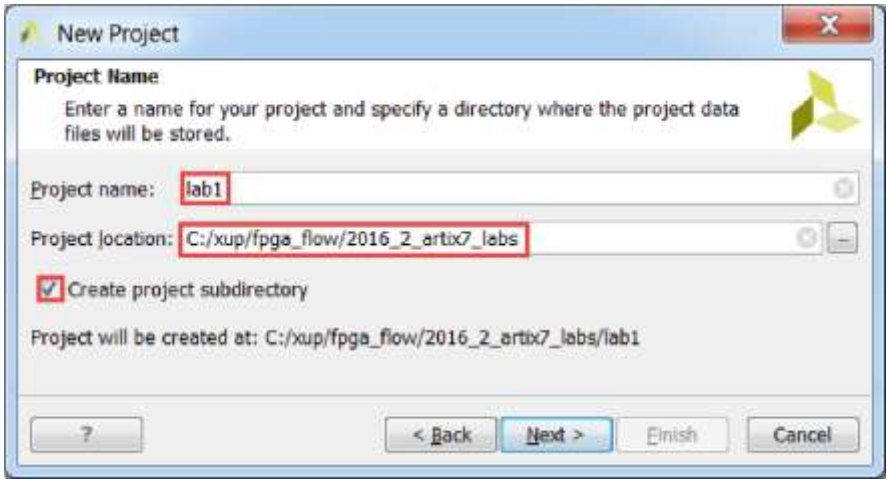


Figure 2. Project Name and Location entry

1-1-5. Select **RTL Project** option in the *Project Type* form, and click **Next**.

1-1-6. Using the drop-down buttons, select **Verilog** as the *Target Language* and *Simulator Language* in the *Add Sources* form.

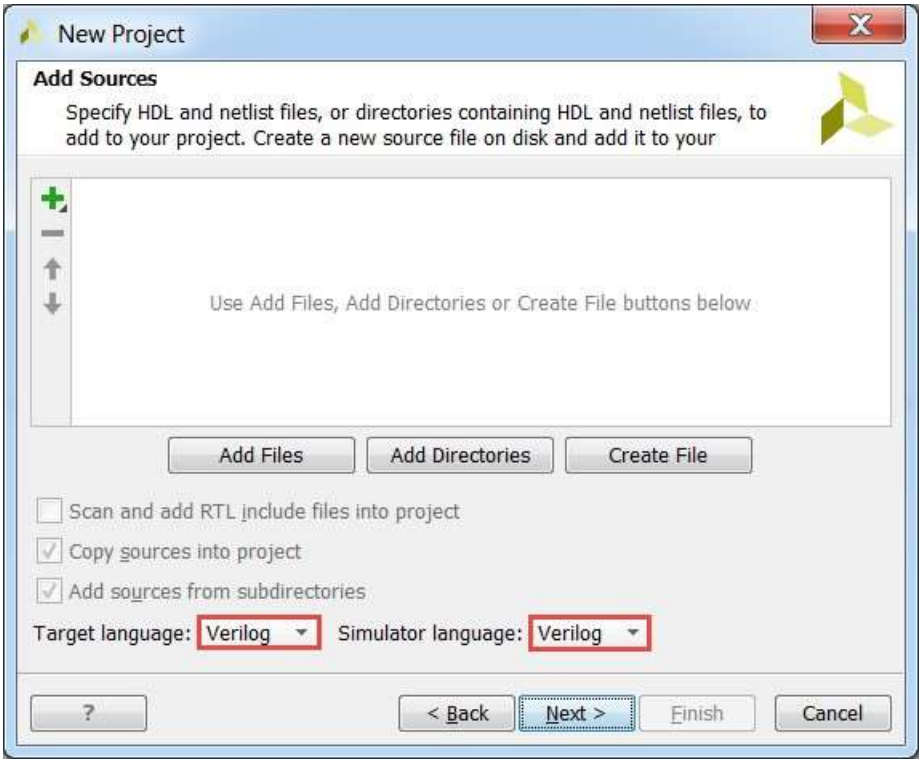


Figure 3. Selecting Target and Simulator language

1-1-7. Click on the **Green Plus** button, then **Add Files…** and browse to the **{sources}\lab1** directory, select *lab1.v,* click **OK**.

If it isn't already checked, check **Copy sources into project** and then click **Next** to get to the *Add Existing IP* form.

1-1-8. Since we do not have any IP to add, click **Next** to get to the *Add Constraints* form.

1-1-9. Click on the **Green Plus** button, then **Add Files…** and browse to the **{sources}\lab1** directory (if necessary), select *lab1_basys3.xdc, lab1_nexys4_ddr.xdc,* or *lab1_nexys_video.xdc* and click **OK** (if necessary), and then click **Next.**

This Xilinx Design Constraints file assigns the physical IO locations on FPGA to the switches and LEDs located on the board. This information can be obtained either through the board's schematic or the board's user guide.

1-1-10. In the *Default Part* form, use the **Parts** option and various drop-down fields of the **Filter** section. If using the *Nexys4 DDR* board, select the **XC7A100TCSG324-1** part. Select the **XC7A35TCPG236-1** for *Basys3* board, or **XC7A200tsbg484-1** for the *Nexys Video* board.

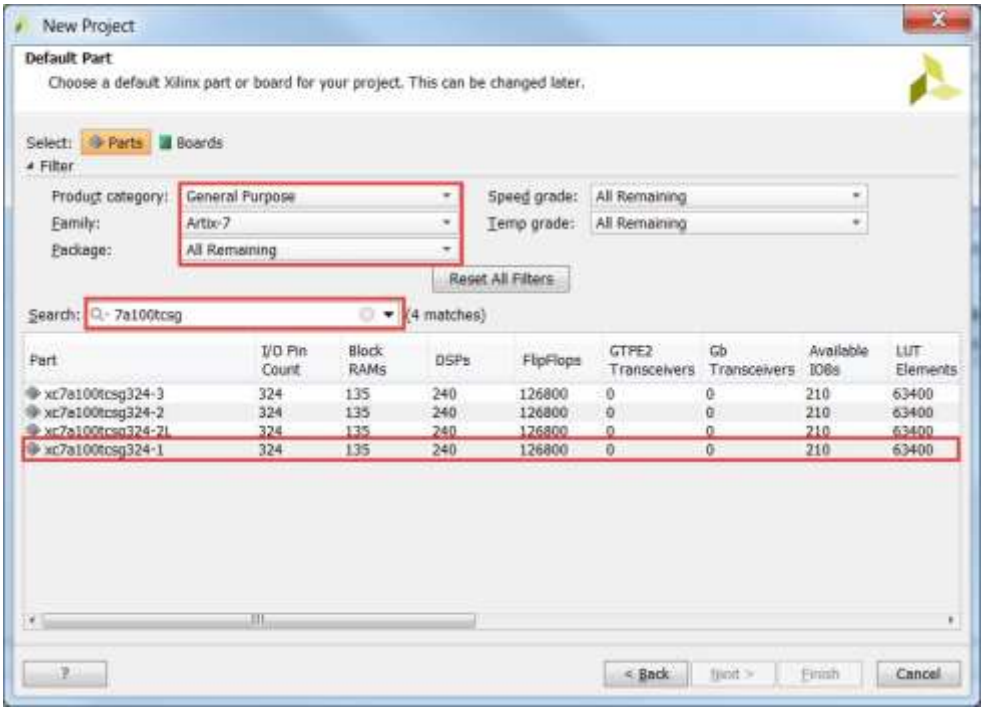


Figure 4. Part Selection for the Nexys4 DDR

You may also select the **Boards** option, select **digilentinc.com** under the Vendor filter and select the appropriate board. Notice that Nexys4 DDR, the Basys3, and the Nexys Video may not be listed as they are not in the tools database. If not listed then you can download the board files for the desired boards either from Digilent Inc website or from the XUP website's workshop material pages and install them in the Vivado installation. You must restart Vivado to see the boards.

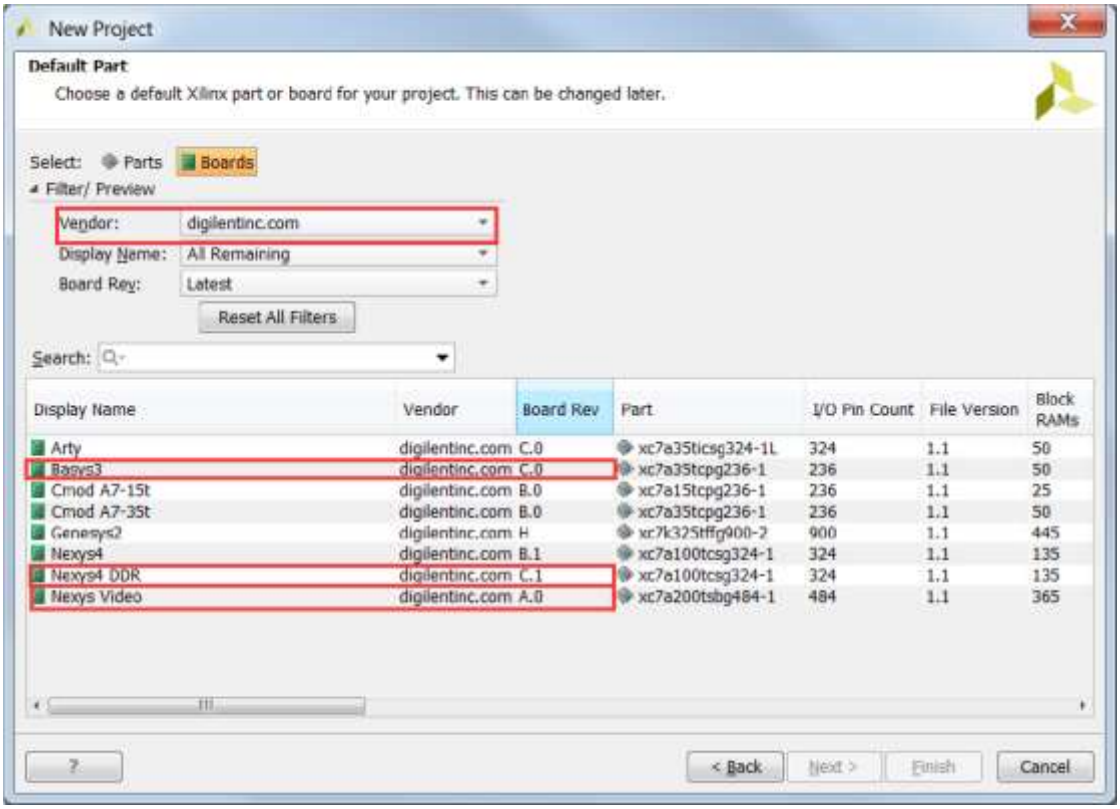


Figure 5. Selecting the target board 1-1-11. Click **Next**.

1-1-12. Click **Finish** to create the Vivado project.

Use the Windows Explorer and look at the {labs}\lab1 directory. You will find that the lab1.cache and lab1.srcs directories and the lab1.xpr (Vivado) project file have been created. The lab1.cache directory is a place holder for the Vivado program database. Two directories, constrs_1 and sources_1, are created under the lab1.srcs directory; deep down under them, the copied lab1_<board>.xdc (constraint) and lab1.v (source) files respectively are placed.

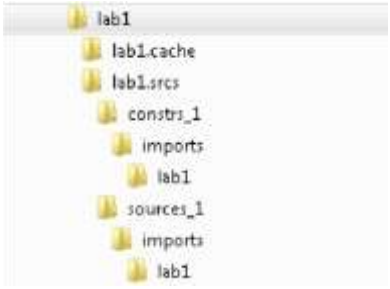


Figure 6. Generated directory structure

1-2. Open the lab1.v source and analyze the content.

1-2-1. In the *Sources* pane, double-click the **lab1.v** entry to open the file in text mode.

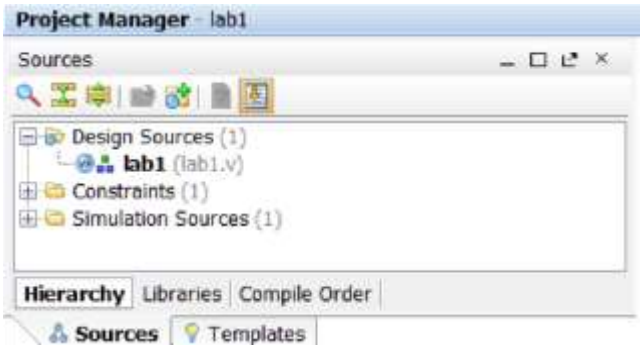


Figure 7. Opening the source file

1-2-2. Notice in the Verilog code that the first line defines the timescale directive for the simulator. Lines 2-4 are comment lines describing the module name and the purpose of the module.

1-2-3. Line 7 defines the beginning (marked with keyword **module**) and Line 19 defines the end of the module (marked with keyword **endmodule**).

1-2-4. Lines 8-9 defines the input and output ports whereas lines 12-17 defines the actual functionality.

1-3. Open the lab1_basys3.xdc, lab1_nexys4_ddr.xdc, or lab1_nexys_video.xdc source and analyze the content.

1-3-1. In the *Sources* pane, expand the *Constraints* folder and double-click the **lab1_<board>.xdc** entry to open the file in text mode.

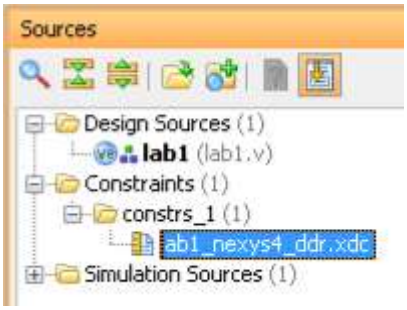


Figure 8. Opening the constraint file

1-3-2. For the lines 5-12 defines the pin locations of the input switches [7:0] and lines 17-24 defines the pin locations of the output LEDs [7:0].

1-4. Perform RTL analysis on the source file.

1-4-1. Expand the *Open Elaborated Design* entry under the *RTL Analysis* tasks of the *Flow Navigator* pane and click on **Schematic**.

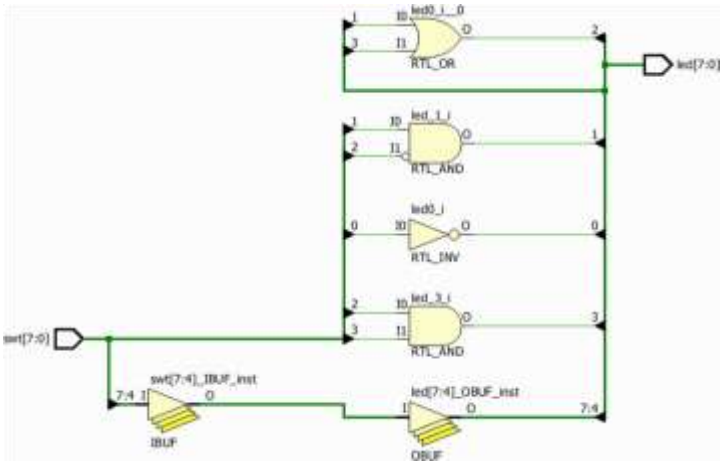


Figure 9. A logic view of the design

Notice that some of the switch inputs go through gates before being output to LEDs through output buffers and the rest go through input buffers and output buffers to LEDs as modeled in the file.

Simulate the Design using the Vivado Simulator Step 2

2-1. Add the lab1_tb.v testbench file.

2-1-1. Click **Add Sources** under the *Project Manager* tasks of the *Flow Navigator* pane.

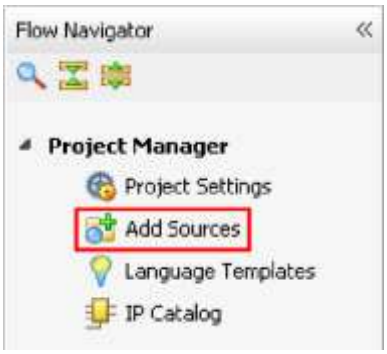


Figure 10. Add Sources

2-1-2. Select the *Add or Create Simulation Sources* option and click **Next**.

Figure 11. Selecting Simulation Sources option

2-1-3. In the *Add Sources Files* form, click the **Green Plus** button and then **Add Files…. 2-1-4.** Browse to the **{sources}\lab1** folder and select *lab1_tb.v* and click **OK**.

2-1-5. Click **Finish**.

2-1-6. Select the *Sources* tab and expand the *Simulation Sources* group.

The lab1_tb.v file is added under the *Simulation Sources* group, and **ab1.v** is automatically placed in its hierarchy as a dut (device under test) instance.

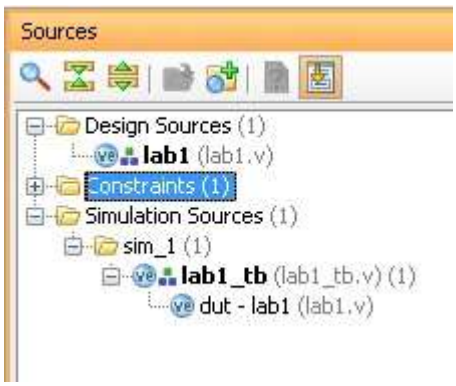


Figure 12. Simulation Sources hierarchy

2-1-7. Using the Windows Explorer, verify that the **sim_1** directory is created at the same level as constrs_1 and sources_1 directories under the lab1.srcs directory, and that a copy of lab1_tb.v is placed under **lab1.srcs > sim_1 > imports > lab1**.

2-1-8. Double-click on the **lab1_tb** in the *Sources* pane to view its contents.

```
1  `timescale 1ns / 1ps
2  //////////////////////////////////////////////////////////////////////////////////
3  // Module Name: lab1_tb
4  //////////////////////////////////////////////////////////////////////////////////
5  module lab1_tb(
6
7      );
8
9      reg [7:0] switches;
10     wire [7:0] leds;
11     reg [7:0] e_led;
12
13     integer i;
14
15     lab1 dut(.led(leds),.swt(switches));
16
17     function [7:0] expected_led;
18        input [7:0] swt;
19     begin
20        expected_led[0] = ~swt[0];
21        expected_led[1] = swt[1] & ~swt[2];
22        expected_led[3] = swt[2] & swt[3];
23        expected_led[2] = expected_led[1] | expected_led[3];
24        expected_led[7:4] = swt[7:4];
25     end
26     endfunction
27
28     initial
29     begin
30         for (i=0; i < 255; i=i+2)
31         begin
32             #50 switches=i;
33             #10 e_led = expected_led(switches);
34             if(leds == e_led)
35                 $display("LED output matched at", $time);
36             else
37                 $display("LED output mis-matched at ",$time,": expected: %b, actual: %b", e_led, leds);
38         end
39     end
40
41 endmodule
42
```

Figure 13. The self-checking testbench

The testbench defines the simulation step size and the resolution in line 1. The testbench module definition begins on line 5. Line 15 instantiates the DUT (device/module under test). Lines 17 through 26 define the same module functionality for the expected value computation. Lines 28 through 39 define the stimuli generation, and compare the expected output with what the DUT provides. Line 41 ends the testbench. The $display task will print the message in the simulator console window when the simulation is run.

2-2. Simulate the design for 200 ns using the Vivado simulator.

2-2-1. Select **Simulation Settings** under the *Project Manager* tasks of the *Flow Navigator* pane.

A **Project Settings** form will appear showing the **Simulation** properties form.

2-2-2. Select the **Simulation** tab, and set the **Simulation Run Time** value to 200 ns and click **OK**.

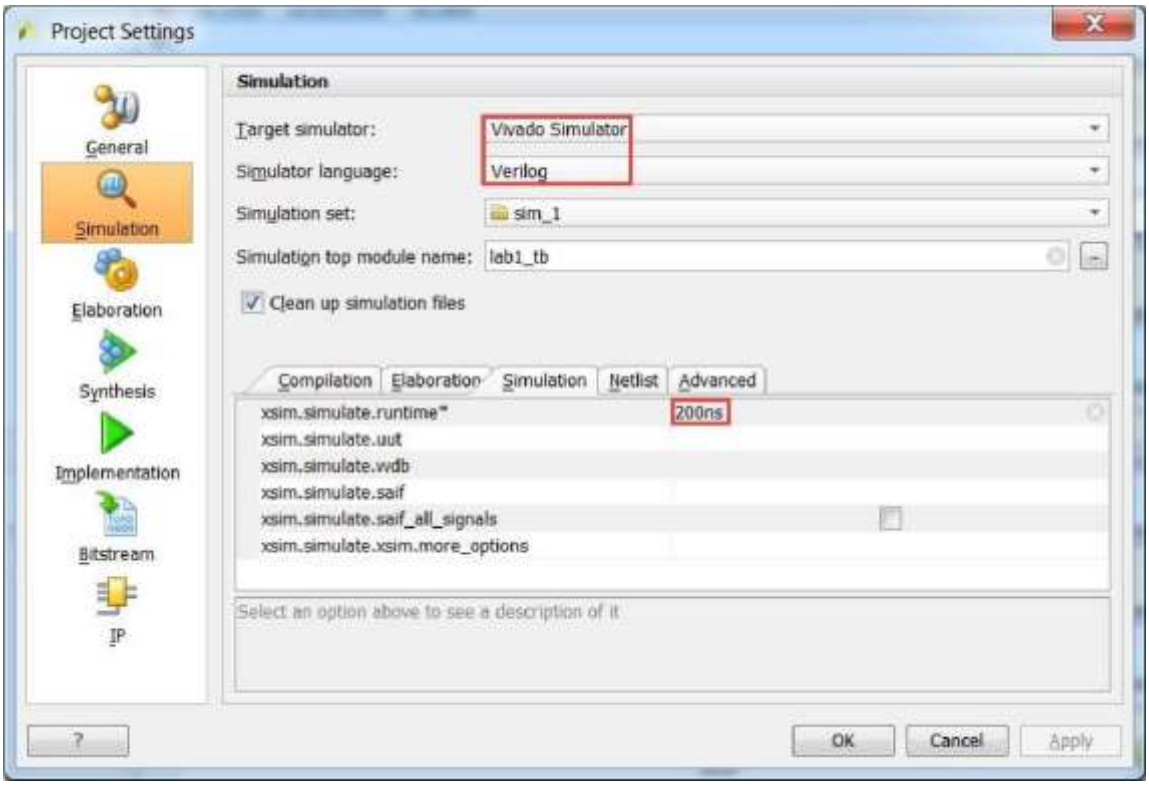


Figure 14. Setting simulation run time

2-2-3. Click on **Run Simulation > Run Behavioral Simulation** under the *Project Manager* tasks of the

Flow Navigator pane.

The testbench and source files will be compiled and the Vivado simulator will be run (assuming no errors). You will see a simulator output. Click on the Zoom Fit ( ) button and you will see output similar to the one shown below.

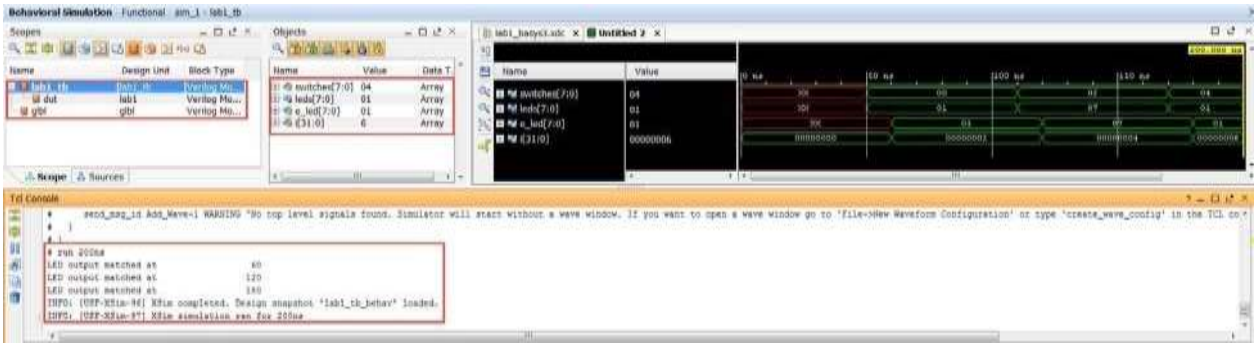


Figure 15. Simulator output

You will see four main views: (i) *Scopes,* where the testbench hierarchy as well as glbl instances are displayed, (ii) *Objects,* where top-level signals are displayed, (iii) the waveform window, and (iv) *Tcl Console* where the simulation activities are displayed. Notice that since the testbench used is selfchecking, the results are displayed as the simulation is run.

Notice that the **lab1.sim** directory is created under the **lab1** directory, along with several lower-level directories.

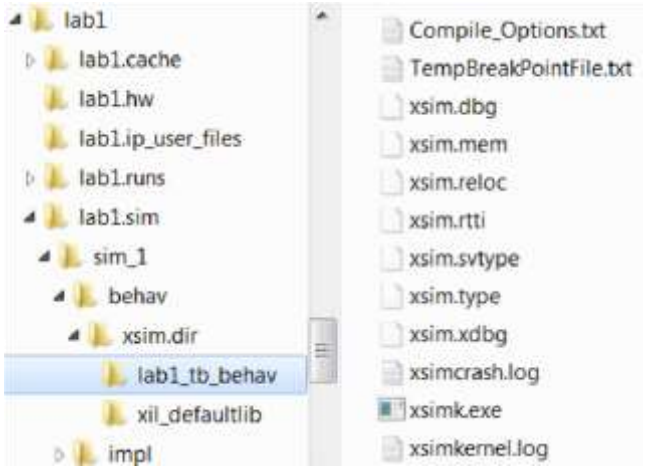


Figure 16. Directory structure after running behavioural simulation

You will see several buttons next to the waveform window which can be used for the specific purpose as listed in the table below.

Table 1: Various buttons available to view the waveform

Waveform options
Save the waveform
Zoom In
Zoom Out
Zoom Fit
Zoom to cursor
Go to Time 0
Go to Last Time
Previous Transition

2-2-4. Click on the *Zoom Fit* button ( ) to see the entire waveform.

Notice that the output changes when the input changes.

You can also float the simulation waveform window by clicking on the Float button on the upper right hand side of the view. This will allow you to have a wider window to view the simulation waveforms. To reintegrate the floating window back into the GUI, simply click on the Dock Window button.

Figure 17. Float Button

Figure 18. Dock Window Button

2-3. Change display format if desired.

2-3-1. Select **i[31:0]** in the waveform window, right-click, select *Radix*, and then select *Unsigned Decimal* to view the for-loop index in *integer* form. Similarly, change the radix of **switches[7:0]** to *Hexadecimal*. Leave the **leds[7:0]** and **e_led[7:0]** radix to *binary* as we want to see each output bit.

2-4. Add more signals to monitor the lower-level signals and continue to run the simulation for 500 ns.

2-4-1. Expand the **lab1_tb** instance, if necessary, in the *Scopes* window and select the **dut** instance.

The swt[7:0] and led[7:0] signals will be displayed in the *Objects* window.

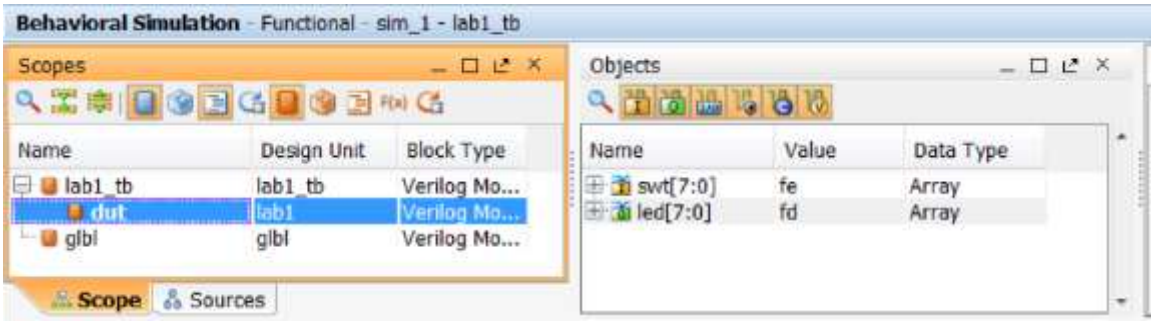
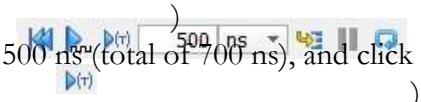


Figure 19. Selecting lower-level signals

2-4-2. Select **swt[7:0]** and **led[7:0]** and drag them into the waveform window to monitor those lower- level signals.

2-4-3. On the simulator tool buttons ribbon bar, type 500 over in the simulation run time field, click on the drop-down button of the units field and select ns ( ) since we want to run for 500 ns (total of 700 ns), and click on the ( ) button.



The simulation will run for an additional 500 ns.

2-4-4. Click on the *Zoom Fit* button and observe the output.

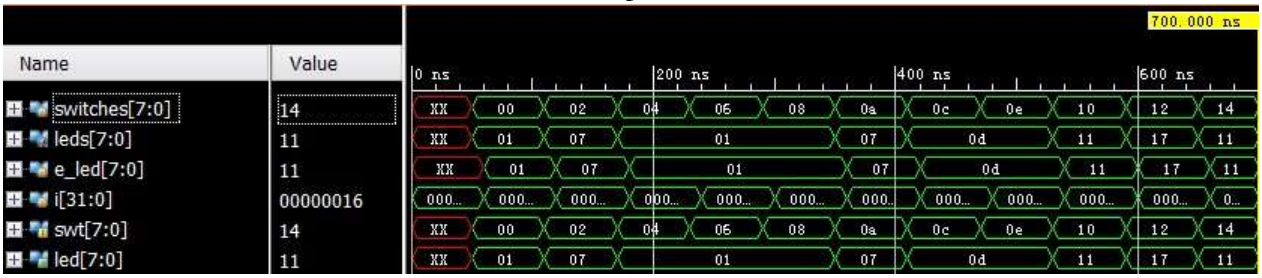


Figure 20. Running simulation for additional 500 ns

Observe the Tcl Console window and see the output is being displayed as the testbench uses the $display task.

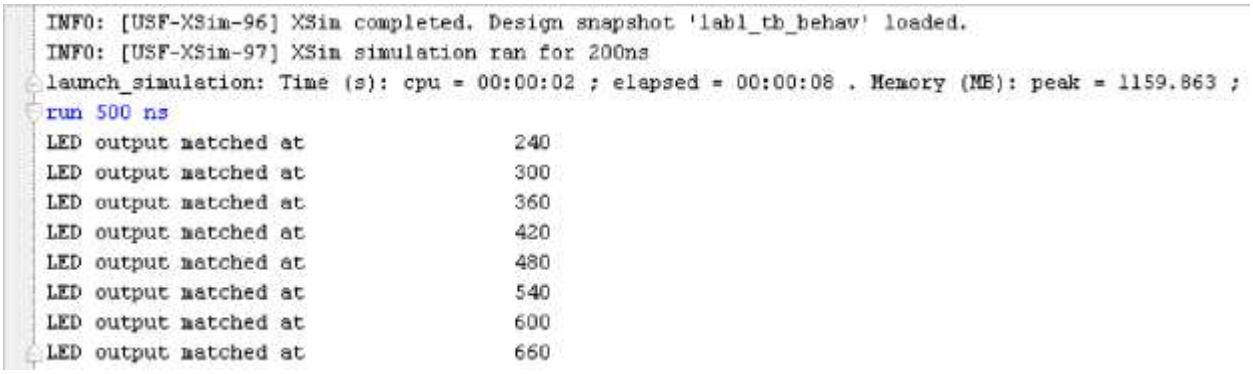


Figure 21. Tcl Console output after running the simulation

for additional 500 ns 2-4-5. Close the simulator by selecting **File > Close Simulation**.

2-4-6. Click **OK** and then click **Discard** to close it without saving the waveform.

Synthesize the Design Step 3

3-1. Synthesize the design with the Vivado synthesis tool and analyze the Project Summary output.

3-1-1. Click on **Run Synthesis** under the *Synthesis* tasks of the *Flow Navigator* pane.

The synthesis process will be run on the lab1.v file (and all its hierarchical files if they exist). When the process is completed a *Synthesis Completed* dialog box with three options will be displayed.

3-1-2. Select the **Open Synthesized Design** option and click **OK** as we want to look at the synthesis output before progressing to the implementation stage.

Click **Yes** to close the elaborated design if the dialog box is displayed.

3-1-3. Select the **Project Summary** tab and understand the various windows.

If you don't see the Project Summary tab then select **Layout > Default Layout,** or click the **Project Summary** icon.

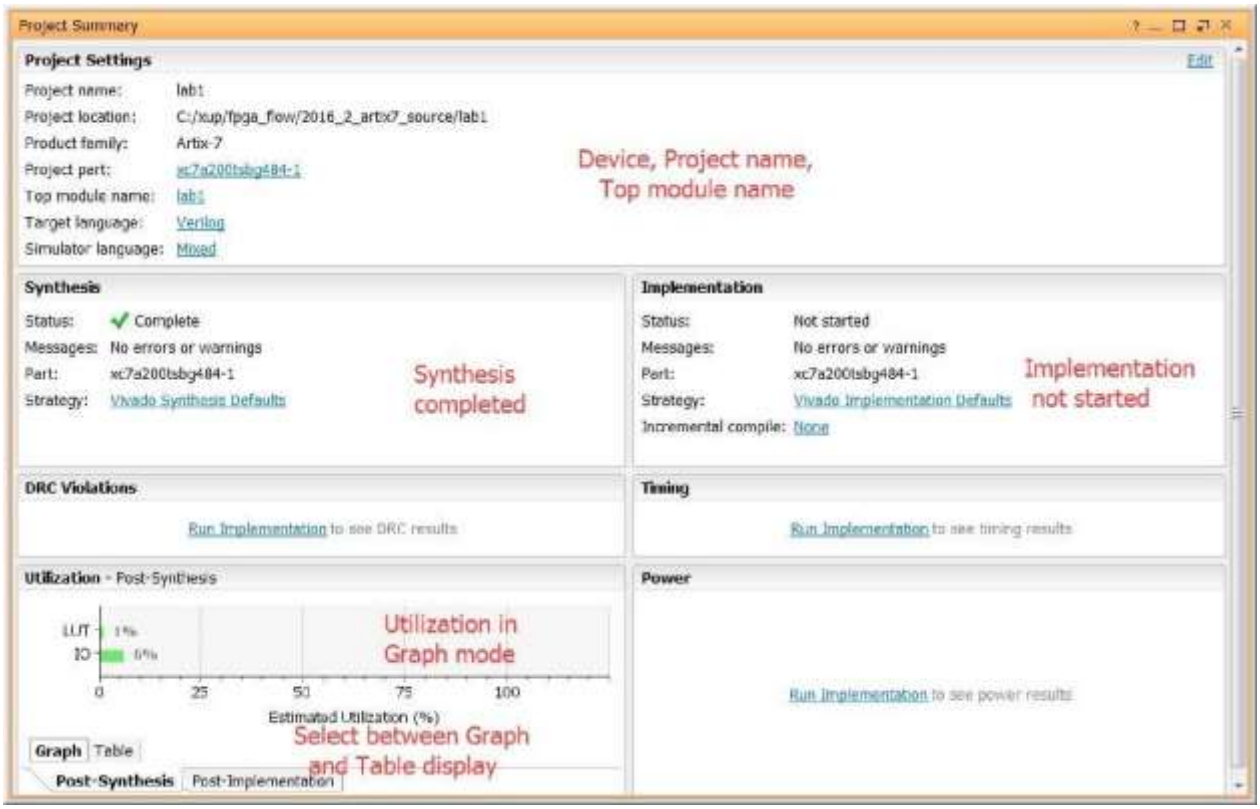


Figure 22. Project Summary view

Click on the various links to see what information they provide and which allows you to change the synthesis settings.

3-1-4. Click on the **Table** tab in the **Project Summary** tab.

Notice that there are an estimated three LUTs and 16 IOs (8 input and 8 output) that are used.

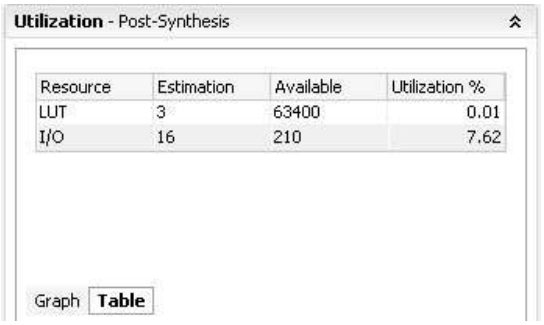


Figure 23. Resource utilization estimation summary for the Nexys4 DDR

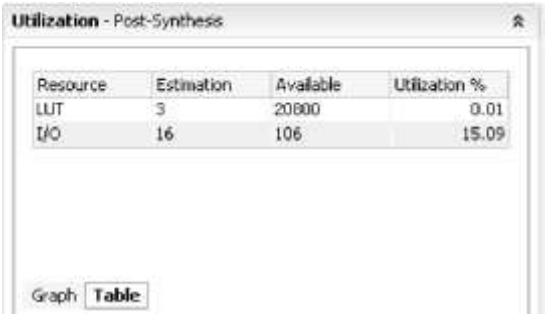


Figure 23. Resource utilization estimation summary for the Basys3

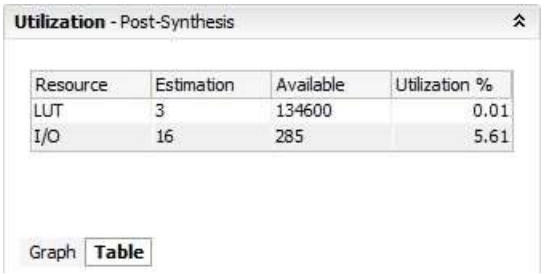


Figure 23. Resource utilization estimation summary for the Nexys Video

3-1-5. In The *Flow Navigator*, under *Synthesis* (expand *Synthesized Design* if necessary), click on **Schematic** to view the synthesized design in a schematic view.

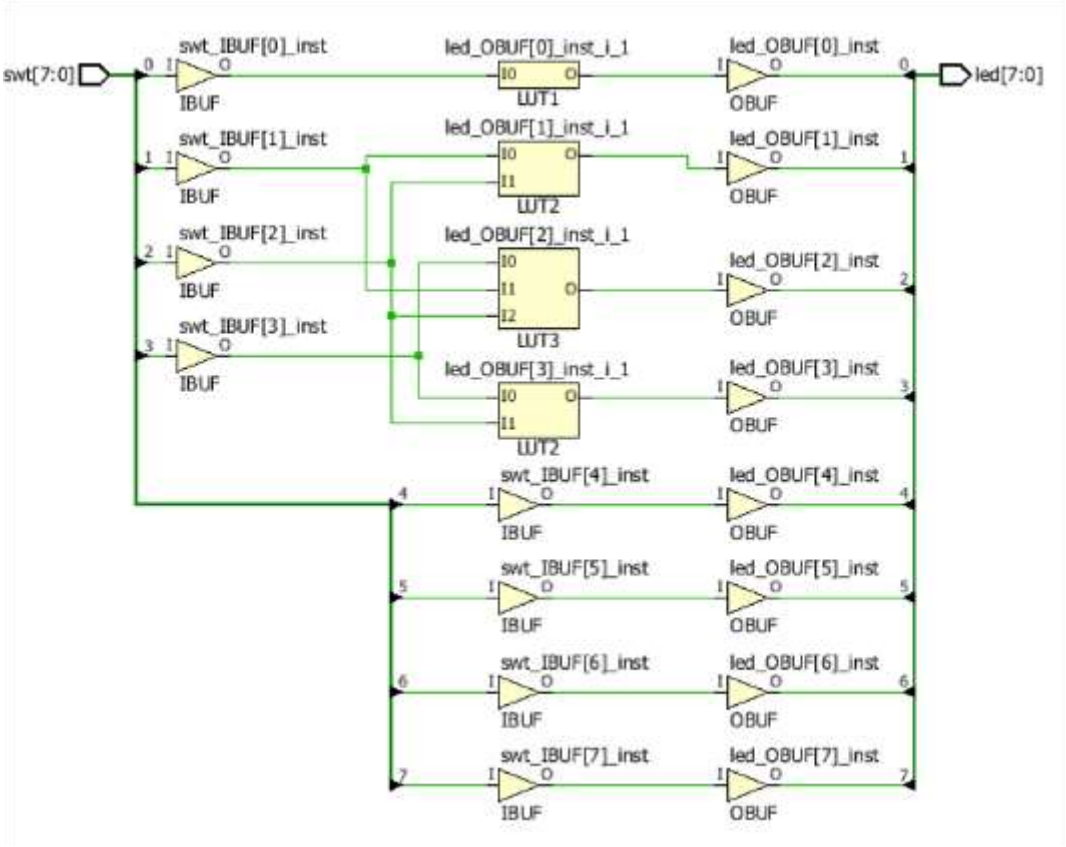


Figure 24. Synthesized design's schematic view

Notice that IBUFs and OBUFs are automatically instantiated (added) to the design as the input and output are buffered. The logical gates are implemented in LUTs (1 input is listed as LUT1, 2 input is listed as LUT2, and 3 input is listed as LUT3). Four gates in RTL analysis output are mapped onto four LUTs in the synthesized output.

Using Windows Explorer, verify that **lab1.runs** directory is created under **lab1**. Under the **runs** directory, **synth_1** directory is created which holds several files related to synthesis.

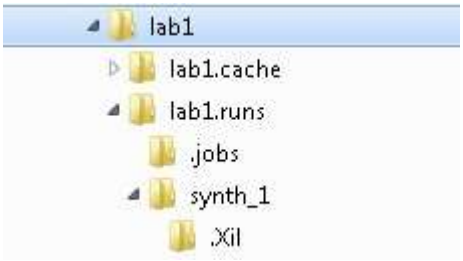


Figure 25. Directory structure after synthesizing the design

Implement the Design Step 4

4-1. Implement the design with the Vivado Implementation Defaults settings and analyze the Project Summary output.

4-1-1. Click on **Run Implementation** under the *Implementation* tasks of the *Flow Navigator* pane.

The implementation process will be run on the synthesized design. When the process is completed an *Implementation Completed* dialog box with three options will be displayed.

4-1-2. Select **Open implemented design** and click **OK** as we want to look at the implemented design in a Device view tab.

4-1-3. Click **Yes,** if prompted, to close the synthesized design.

The implemented design will be opened.

4-1-4. In the *Netlist* pane, select one of the nets (e.g. led_OBUF[1]) and notice that the net displayed in the X1Y1 (Nexys4 DDR), X0Y0 (Basys3), or X0Y0 (Nexys Video) clock region in the Device view tab (you may have to zoom in to see it).

4-1-5. If it is not selected, click the *Routing Resources* icon  to show routing resources.

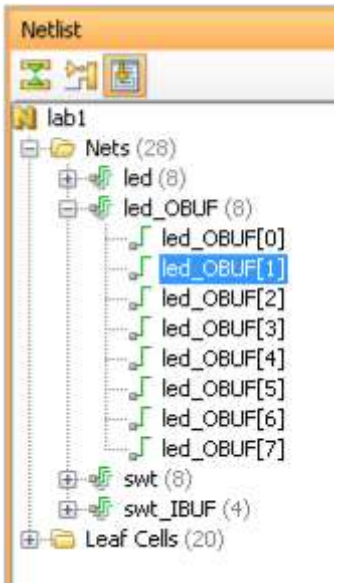


Figure 26. Selecting a net

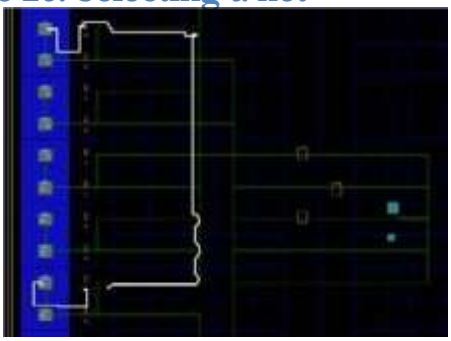


Figure 27. Viewing implemented design for the Nexys4 DDR

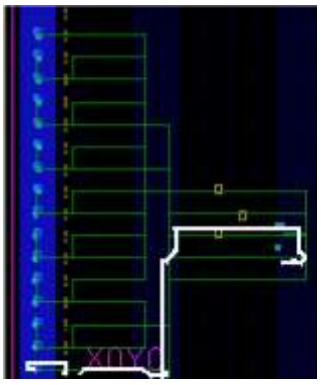


Figure 27. Viewing implemented design for the Basys3

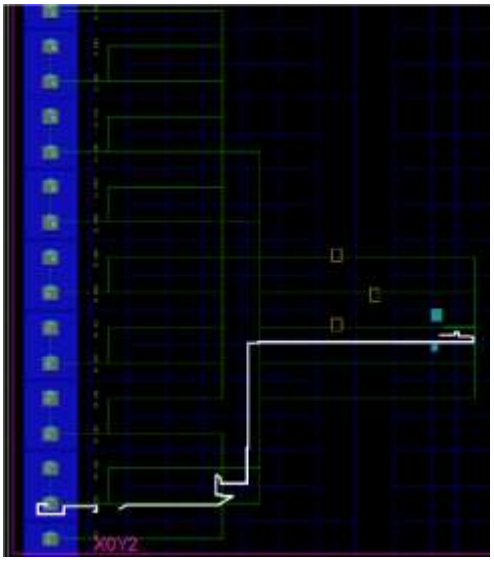


Figure 27. Viewing implemented design for the Nexys Video

4-1-6. Close the implemented design view by selecting **File > Close Implemented Design**, and select the **Project Summary** tab (you may have to change to the Default Layout view) and observe the results.

Select the Post-Implementation tab.

Notice that the actual resource utilization is three LUTs and 16 IOs. Also, it indicates that no timing constraints were defined for this design (since the design is combinatorial).

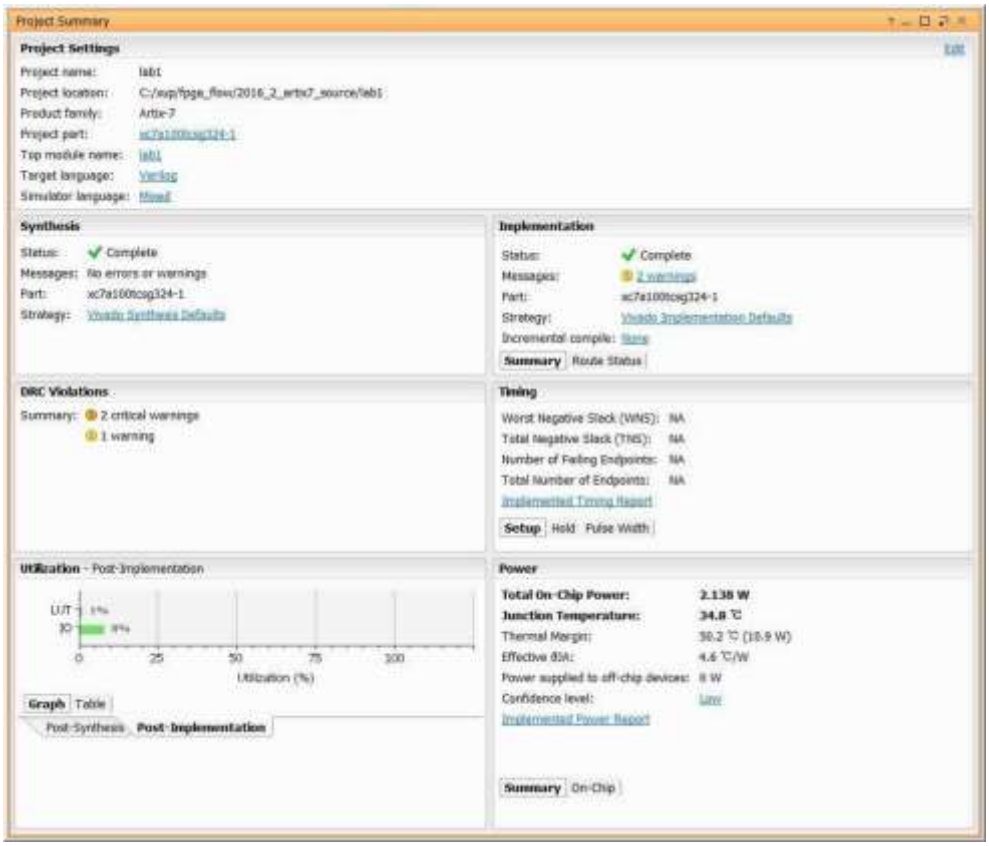


Figure 28. Implementation results for the Nexys4 DDR

Using the Windows Explorer, verify that **impl_1** directory is created at the same level as **synth_1** under the **lab1.runs** directory. The **impl_1** directory contains several files including the implementation report files.

4-1-7. In Vivado, select the **Reports** tab in the bottom panel (if not visible, click *Window* in the menu bar and select **Reports**), and double-click on the *Utilization Report* entry under the *Place Design* section. The report will be displayed in the auxiliary view pane showing resource utilization. Note that since the design is combinatorial no registers are used.

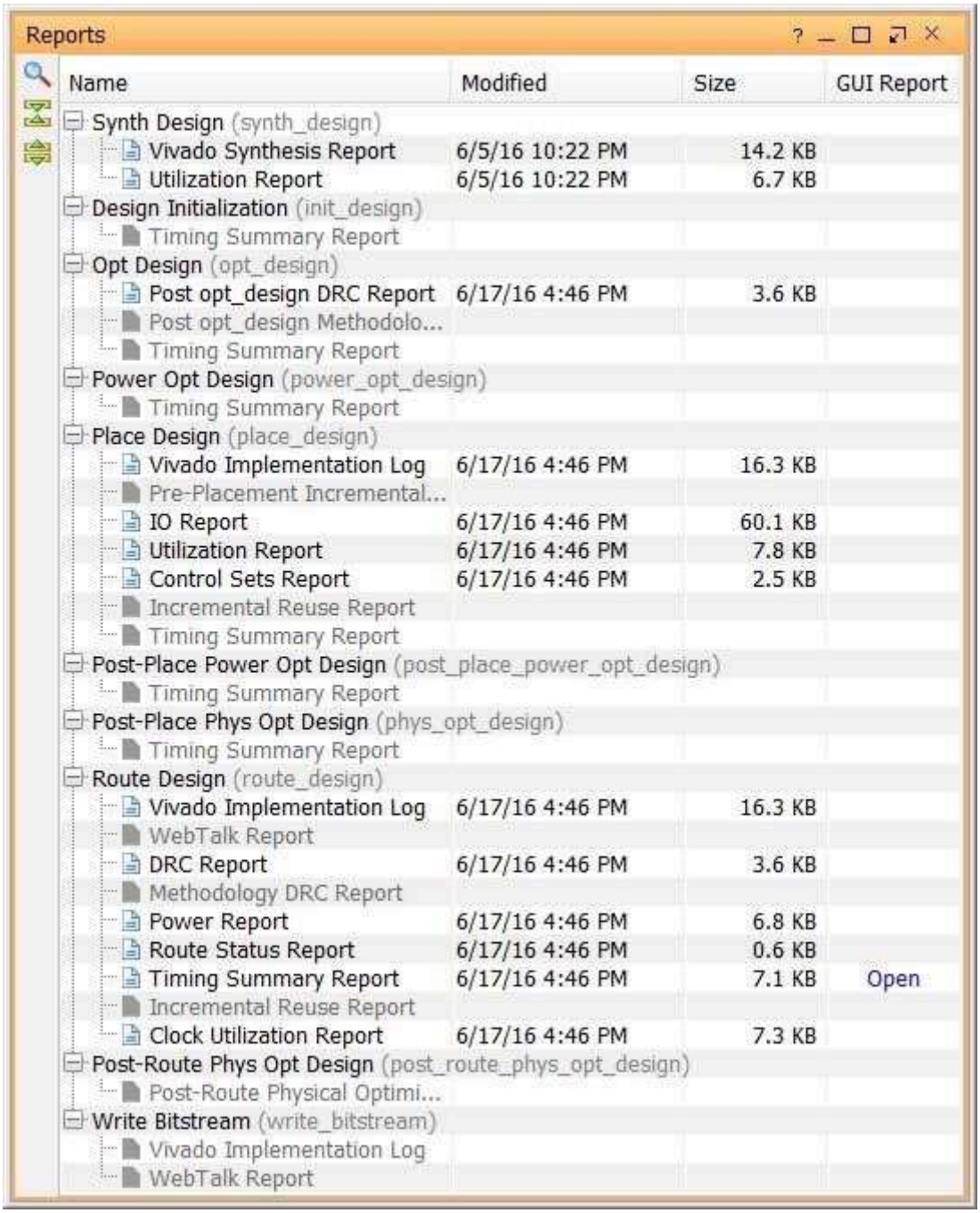


Figure 29. Available reports to view

Perform Timing Simulation Step 5

5-1. Run a timing simulation.

5-1-1. Select **Run Simulation > Run Post-Implementation Timing Simulation** process under the *Simulation* tasks of the *Flow Navigator* pane.

The Vivado simulator will be launched using the implemented design and **lab1_tb** as the top-level module.

Using the Windows Explorer, verify that **timing** directory is created under the **lab1.sim > sim_1 > impl** directory. The **timing** directory contains generated files to run the timing simulation.

5-1-2. Click on the **Zoom Fit** button to see the waveform window from 0 to 200 ns.

5-1-3. Right-click at 50 ns (where the switch input is set to 0000000b) and select **Markers > Add Marker**.

5-1-4. Similarly, right-click and add a marker at around 59.260 ns where the **leds** changes.

5-1-5. You can also add a marker by clicking on the Add Marker button ( ). Click on the **Add Marker** button and left-click at around 60 ns where **e_led** changes.

Figure 30. Timing simulation output

Notice that we monitored the expected led output at 10 ns after the input is changed (see the testbench) whereas the actual delay is about 5 to 5.9 ns (depending on the board).

5-1-6. Close the simulator by selecting **File > Close Simulation** without saving any changes.

Generate the Bitstream and Verify Functionality Step 6

6-1. Connect the board and power it ON. Generate the bitstream, open a hardware session, and program the FPGA.

6-1-1. Make sure that the Micro-USB cable is connected to the JTAG PROG connector (next to the power supply connector for the Nexys4 DDR and the Basys 3. The Nexys Video JTAG PROG connector is located next to the UART connector).

6-1-2. Make sure that the board is set to use USB power (via the Power Select jumper JP3 on the Nexys4 DDR and JP2 on the Basys3)

The Nexys Video requires a separate power source, connected via J21, located next to the POWER switch.

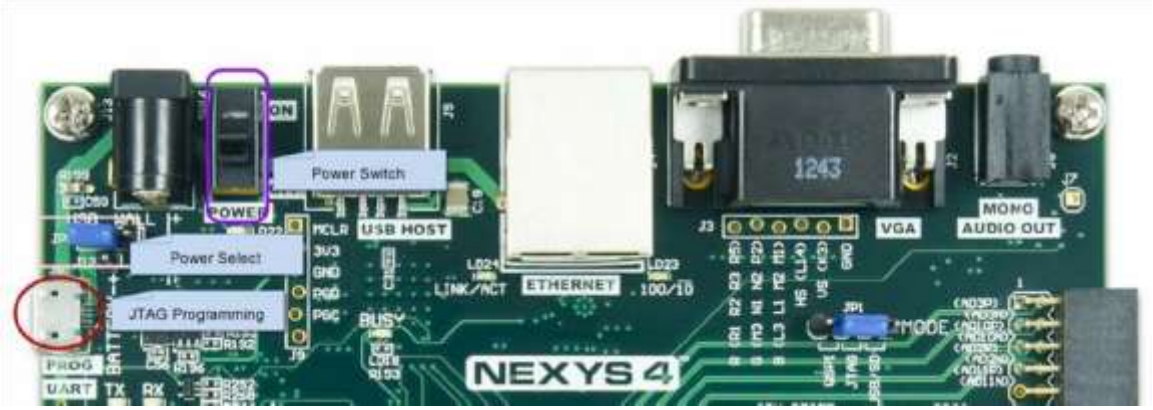


Figure 31. Board connection for the Nexys4 DDR*

6-1-3. Power **ON** the board.

6-1-4. Click on the **Generate Bitstream** entry under the *Program and Debug* tasks of the *Flow Navigator* pane.

The bitstream generation process will be run on the implemented design. When the process is completed a *Bitstream Generation Completed* dialog box with three options will be displayed.

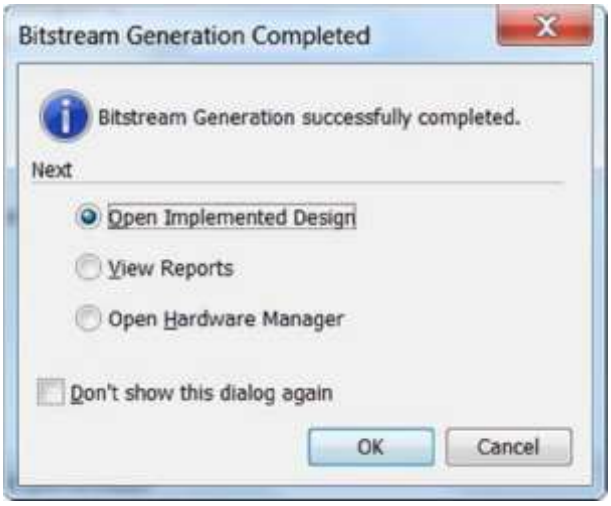


Figure 32. Bitstream generation

This process will have generated a **lab1.bit** file under **impl_1** directory in the **lab1.runs** directory.

6-1-5. Expand the *Open Hardware Manager* option and click *Open Target* option.

6-1-6. Click on the Select *Auto Connect* option.

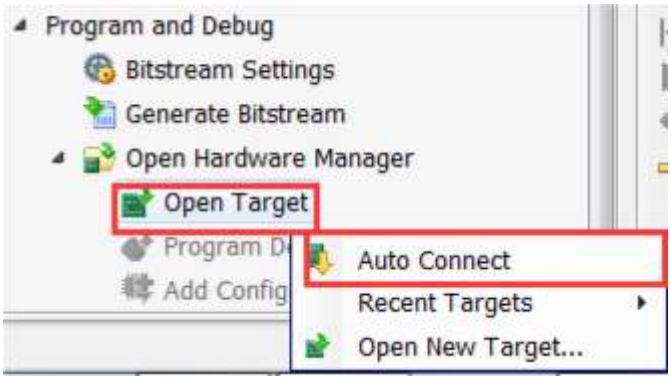


Figure 33. Selecting Auto Connect

6-1-7. The Hardware Manager Window will open automatically. You can find your device listed in the

Hardware Window. Also notice that the Status indicates that it is not programmed.

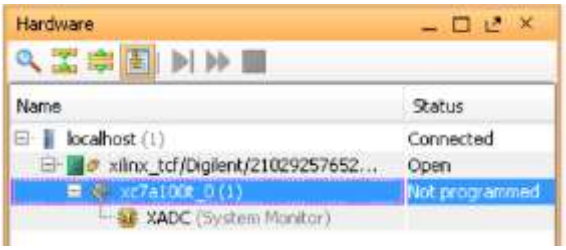


Figure 34. Opened hardware session for the Nexys4 DDR

6-1-8. Select the device and verify that the lab1.bit is selected as the programming file in the General tab.

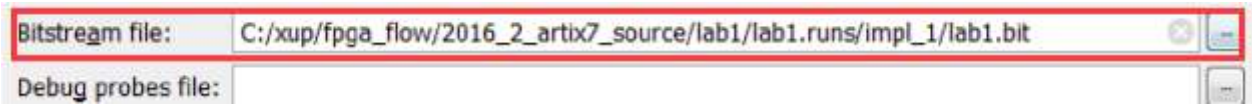


Figure 35. Programming file

6-1-9. Click on the *Program device* > *XC7A100T_0* or the *XC7A35T_0* link in the green information bar to program the target FPGA device.

Another way is to right click on the device and select *Program Device*…

Figure 36. Selecting to program

the FPGA 6-1-10. Click

Program to program the FPGA.

The DONE light will light when the device is programmed. You may see some other LEDs lit depending on switch positions.

6-1-11. Verify the functionality by flipping switches and observing the output on the LEDs (Refer to the earlier logic diagram).

6-1-12. When satisfied, power **OFF** the board.

6-1-13. Close the hardware session by selecting **File > Close**

Hardware Manager. 6-1-14. Click **OK** to close the session.

6-1-15. Close the **Vivado** program by selecting **File > Exit** and click **OK**.

Conclusion

The Vivado software tool can be used to perform a complete HDL based design flow. The project was created using the supplied source files (HDL model and user constraint file). A behavioral simulation using the provided testbench was done to verify the model functionality. The model was then synthesized, implemented, and a bitstream was generated. The timing simulation was run on the implemented design using the same testbench. The functionality was verified in hardware using the generated bitstream.

Lab Exercises:

1. Write down the Verilog code for Full adder and testbench.
2. Perform functional simulation, synthesis and Implementation.
3. Submit and comment on the differences among functional, post-synthesis timing and post-implementation timing simulation.

CODE

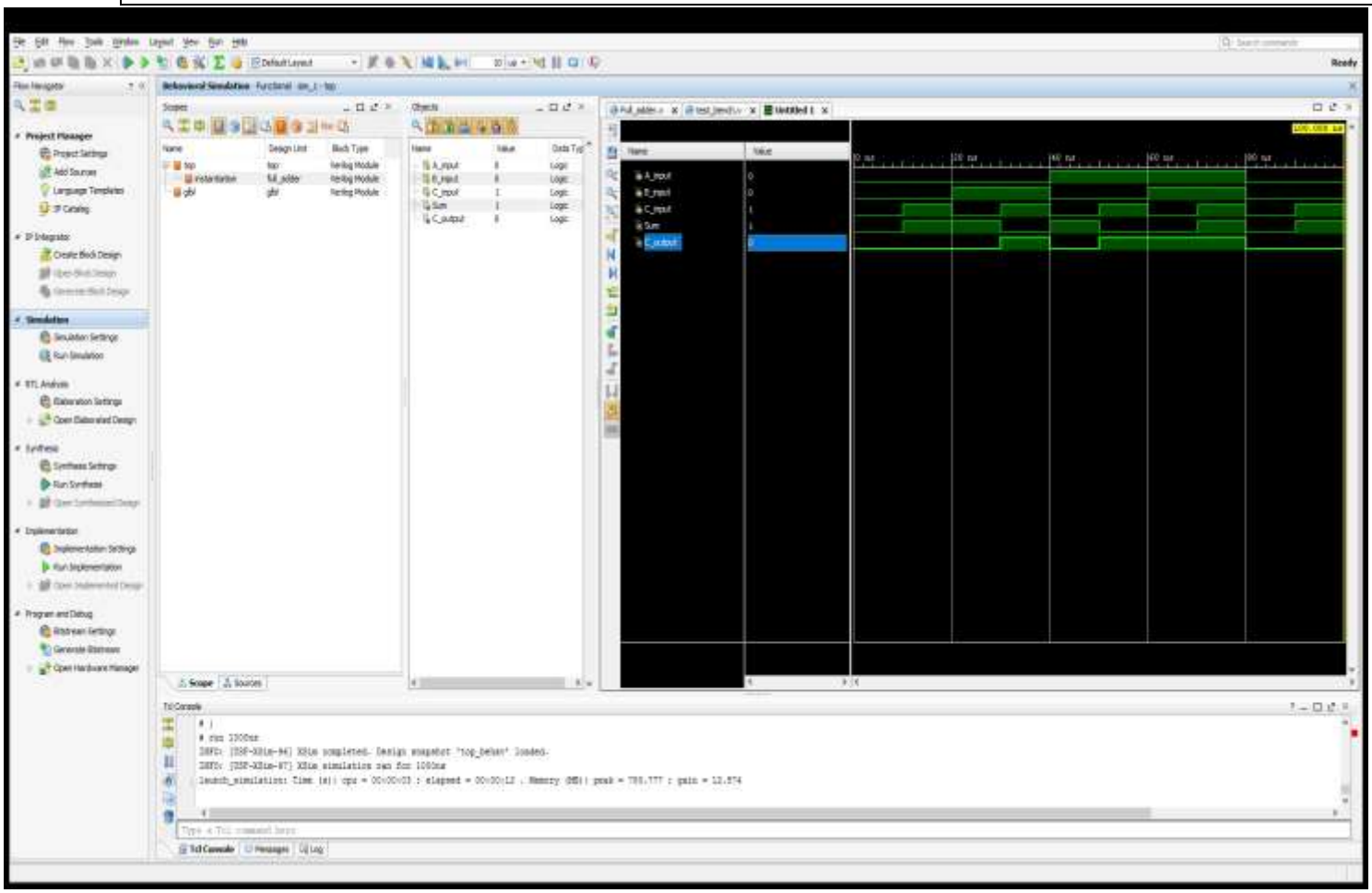
```
`timescale 1ns / 1ps
module full_adder( A, B, Cin, S, Cout);

input wire A, B, Cin;
output reg S, Cout;

always @(A or B or Cin)
begin
 if(A==0 && B==0 && Cin==0)
  begin
   S=0;
   Cout=0;
  end

 else if(A==0 && B==0 && Cin==1)
  begin
   S=1;
   Cout=0;
  end

 else if(A==0 && B==1 && Cin==0)
  begin
   S=1;
   Cout=0;
  end

 else if(A==0 && B==1 && Cin==1)
  begin
   S=0;
   Cout=1;
  end

 else if(A==1 && B==0 && Cin==0)
  begin
   S=1;
   Cout=0;
  end

 else if(A==1 && B==0 && Cin==1)
  begin
   S=0;
   Cout=1;
  end
```

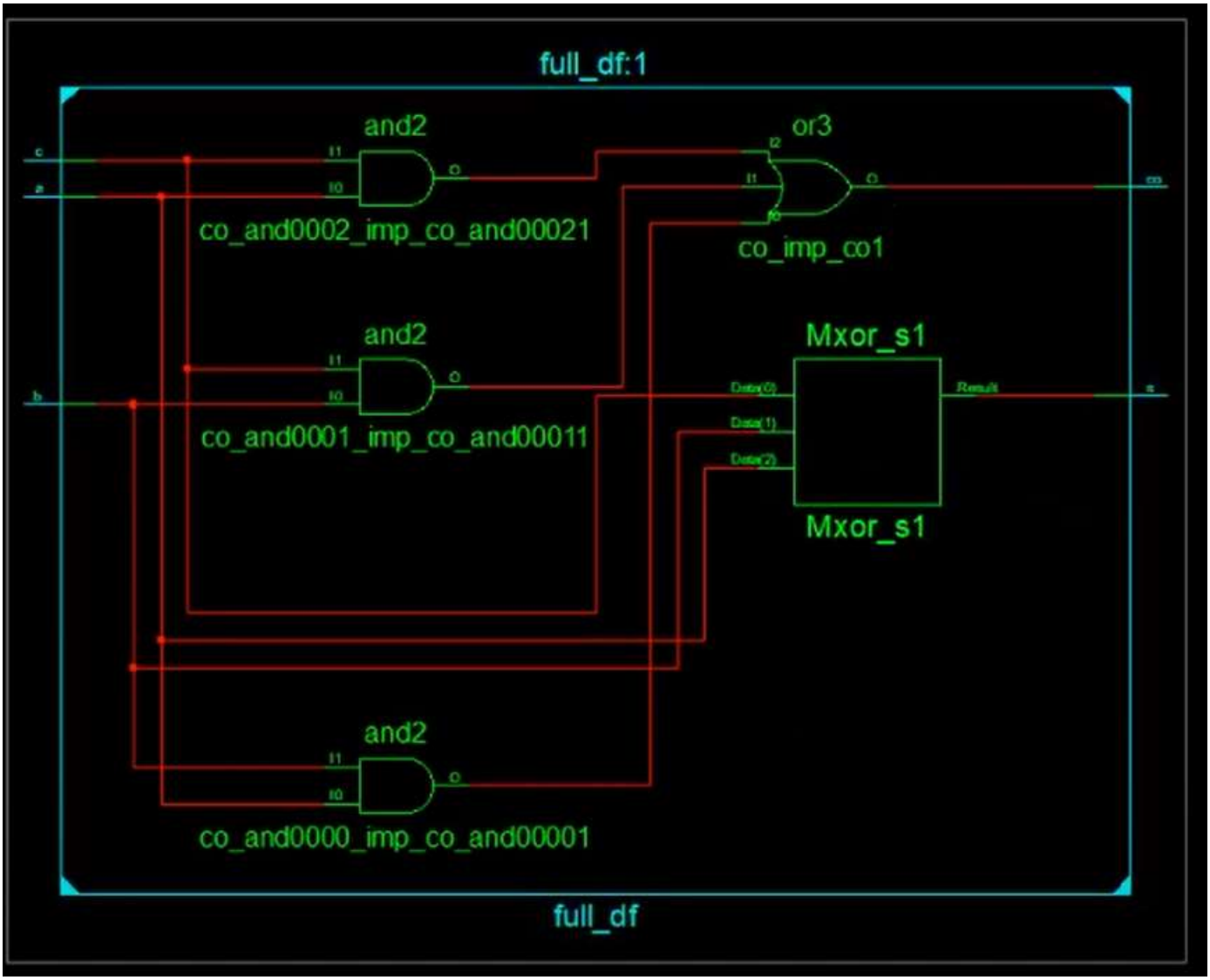
```
 else if(A==1 && B==1 && Cin==0)
  begin
   S=0;
   Cout=1;
  end

 else if(A==1 && B==1 && Cin==1)
  begin
   S=1;
   Cout=1;
  end

end

endmodule
```

Test_Bench

```
//timescale directive
`timescale 1ns / 1ps
module top;

  //declare testbench variables
  reg  A_input, B_input, C_input;
  wire Sum, C_output;

 //instantiate the design module and connect to the testbench variables
  full_adder instantiation(.A(A_input), .B(B_input), .Cin(C_input), .S(Sum), .Cout(C_output));

  initial
    begin
      $dumpfile("xyz.vcd");
      $dumpvars;

      //set stimulus to test the code
      A_input=0;
      B_input=0;
      C_input=0;
       #100 $finish;
    end

//provide the toggling input (just like truth table input)
//this acts as the clock input
always #40 A_input=~A_input;
```

```
always #20 B_input=~B_input;
always #10 C_input=~C_input;

//display output if there's a change in the input event
  always @(A_input or B_input or C_input)
     $monitor("At TIME(in ns)=%t, A=%d B=%d C=%d Sum = %d Carry = %d", $time, A_input,
B_input, C_input, Sum, C_output);

endmodule
```





In short the post-implementation, or timing simulation takes into account the delays associated with the actual synthesis and logic placement. It is a more accurate picture of how robust your design is. While the behavioral simulation has to account for clocked delays like registers it is naive about routing delays. Generally, behavioral simulation is good for checking logic. Of course it's only as good as your testbench. If you are doing serious work then formal methods are better for catching events or conditions that you might not have thought about

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#08

Download HDL design using Open-Source FPGA Tools

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#08

Download HDL design using Open-Source FPGA Tools

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali

033-18-0037

Lab Learning Objectives:

After completing this session, student should be able to:

- Download and install open-source FPGA tools.
- Work with open-source FPGA development board.
- Build, verify and download an HDL design using open-source tools.

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	☐ Desired output	☐ Minor mistakes	☐ Critical mistakes	
2	Timing	☐ Submitted within	☐ 1 day late	☐ More than 1 day	

		the given time		late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. FPGA development board.
2. Desktop/Laptop Computer

Background Theory:

Open-source software represents software for which the original source code is made freely available and may be redistributed and modified. It is released under a license in which the copyright holder grants users the rights to use, study, change, and distribute the software and its source code to anyone and for any purpose. Open-source software is usually easier to obtain than proprietary software, often resulting in free and increased use.

Vivado Design Suite is a proprietary software suite by Xilinx for synthesis and analysis of hardware description language designs. We will explore here a set of different tools which we can utilize to bypass proprietary tools for FPGA development, some of these are listed as under.

Step#	Design Flow	Tools	Description
1	Design Entry	(text editor)	Write HDL code for your program, e.g., Notepad or Scriptum.
2	Synthesis	**yosys**	Takes your HDL code and translates in a generic gate-level representation i.e. a complete circuit with logical elements (gates, flip flops, etc.) for the design. It is platform independent and gives estimation of hardware utilization.

3	Implementation	**nextpnr**	It takes the generic design and converts into actual platform specific gate and wire connection. The output is an ASCII file. The hardware utilization at this stage could differ from synthesis step due to optimization.
4	Program FPGA	**icepack + iceprog**	**icepack** takes ASCII file and convert into machine readable binary file. **iceprog** uploads the binary file to flash memory of the FPGA.

Apio Toolbox:

Apio is a python-based multiplatform toolbox to verify, synthesize, simulate and upload your verilog designs. The following list contains some of Apio commands with corresponding tools it uses.

Step#	Command	Description
1	apio verify	Syntax verification for the HDL code.
2	apio build	It is used for Synthesis and Impementation of the HDL design. It calls yosys + nextPNR + icepack tools.
3	apio upload	It is used for uploading the implanted design to FPGA. It calls iceprog tool.

FPGA development board – iCEBreaker: iCEBreaker is the open-source iCE40 FPGA development board. It has iCE40UP5K FPGA chip. We will download and verify our FPGA program on this board. Please read more specs on https://www.crowdsupply.com/1bitsquared/icebreaker-fpga.

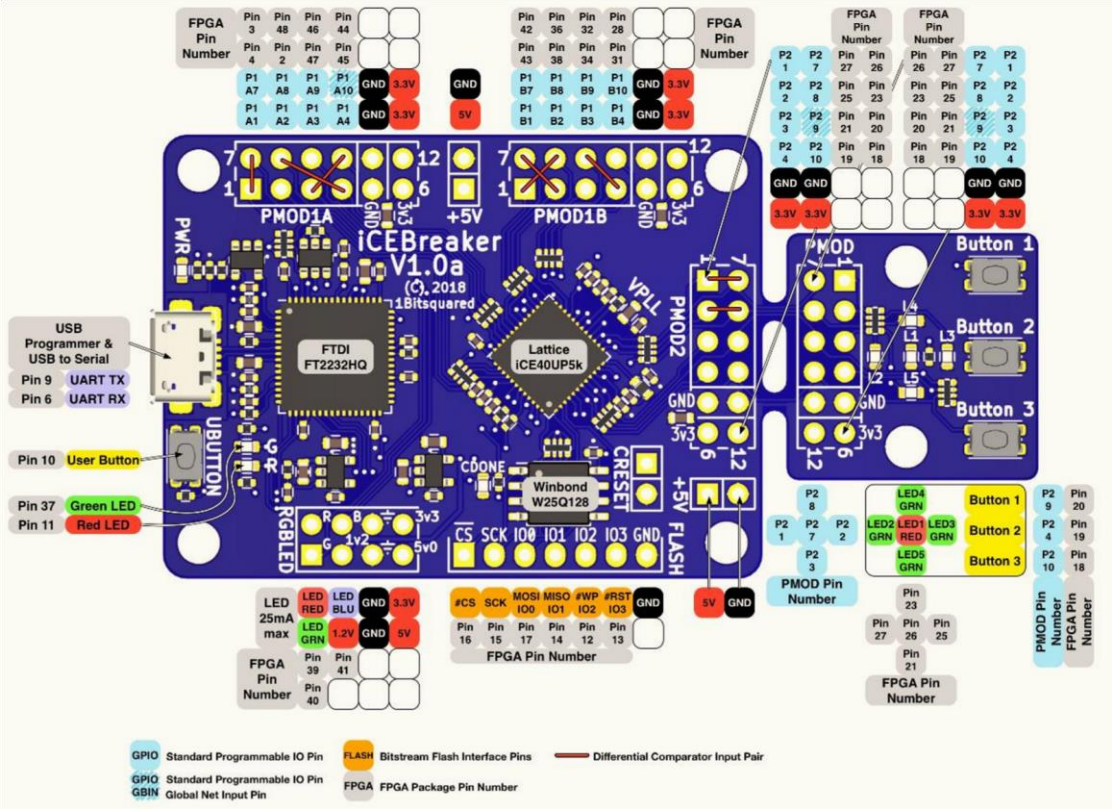


Fig 1: iCEBreaker board pin diagram.

Procedure:

1. Installation of the Apio toolbox

First install Python3 and then go through the following the set of commands on command prompt in Windows platform.

python –version // Checks python version. Python 3.5+ is better.
python -m pip install apio==0.6.7 // Install Apio v0.6.7
apio –version // Checks Apio version apio install --all // Install required open-source FPGA tools this may take an hour.
apio install drivers // Install drivers for the board. apio
drivers --ftdi-enable // FTDI drivers configuration apio
system --lsftdi // Lists connected FTDI device.

2. Download examples in a folder

Make a directory for example folder. Say F:>apio\

C:\Users\Admin>**f:**
F:\>**mkdir apio**
F:\>**cd apio**
F:\apio>**apio examples -l** // it will list all the downloaded examples.

// You may choose which board\examples to install. For example, we may choose iCEBreaker\buttons, iCEBreaker\blink and iCEBreaker\blinky.

F:\apio>**apio examples -d iCEBreaker\buttons**
F:\apio>**apio examples -d iCEBreaker\blink**
F:\apio>**apio examples -d iCEBreaker\blinky**

// Open these folders and you may see HDL design files.
// Open .pcf (pins contrained file). This defines how FPGA pins are connected on the dev board.

3. Download an HDL design on iCEBreaker board.

First choose a design and go to the corresponding folder, say iCEBreaker\buttons.

F:\apio>**cd icebreaker\buttons**
F:\apio\iCEBreaker\buttons>**apio verify**
F:\apio\iCEBreaker\buttons>**apio build**
F:\apio\iCEBreaker\buttons>**apio upload**

Lab Exercises:

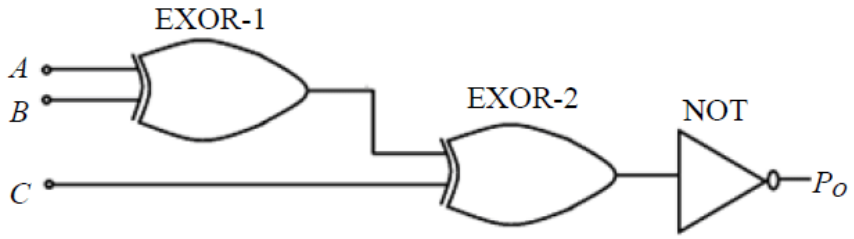
Make a combinational logic circuit, download and verify its functionality on the iCEBreaker board. Submit your report.

Code :

```
// Example of using 3 buttons for 3-bit even parity checker
// Board: iCE40-UP5K
module rgb_test (
        input BTN_N,
        input BTN1,
        input BTN2,
        input BTN3,
```

```
        output LEDR_N,
        output LEDG_N,
        output LED4,
        output LED1,
        output LED5
);


assign LED4 = ~((BTN1 ^ BTN3) ^ BTN2);

endmodule
```

Circuit



Conclusion:

To implement the given circuit in open source we used icebreaker open source board, we initialized input with button BTN1, BTN2 and BTN3 and output LED4 is used. XOR gate basically is used to display the odd parity logic but not gate in the end makes it the circuit logic for even parity design. Hence this circuit is the logical representation for even parity bit checker. This open source Icebreaker circuit is very much useful and helpful and sensible. Icebreaker open source circuit with easy interface make it easy to implement the even parity

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#09

Simulate HDL design using Open-Source FPGA Tools

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#09

Simulate HDL design using Open-Source FPGA Tools

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali

Lab Learning Objectives:

After completing this session, student should be able to:

- Download and install open-source FPGA tools.
- Simulate an HDL design using open-source tools.

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	☐ Desired output	☐ Minor mistakes	☐ Critical mistakes	
2	Timing	☐ Submitted within the given time	☐ 1 day late	☐ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. Desktop/Laptop Computer

Background Theory:

Open-source software represents software for which the original source code is made freely available and may be redistributed and modified. It is released under a license in which the copyright holder grants users the rights to use, study, change, and distribute the software and its source code to anyone and for any purpose. Open-source software is usually easier to obtain than proprietary software, often resulting in free and increased use.

Vivado Design Suite is a proprietary software suite by Xilinx for synthesis and analysis of hardware description language designs. We will explore here a set of different tools which we can utilize to bypass proprietary tools for FPGA development, some of these are listed as under.

Step#	Design Flow	Tools	Description
1	Design Entry	(text editor)	Write HDL code for your program, e.g., Notepad or Scriptum.
2	Simulate	**gtkwave**	GTKWave is an analysis tool used to perform debugging on Verilog or VHDL simulation models.
3	Synthesis	**yosys**	Takes your HDL code and translates in a generic gate-level representation i.e. a complete circuit with logical elements (gates, flip flops, etc.) for the design. It is platform independent and gives estimation of hardware utilization.
4	Implementation	**nextpnr**	It takes the generic design and converts into actual platform specific gate and wire connection. The output is an ASCII file. The hardware utilization at this stage could differ from synthesis step due to optimization.

5	Program FPGA	**icepack + iceprog**	**icepack** takes ASCII file and convert into machine readable binary file. **iceprog** uploads the binary file to flash memory of the FPGA.

Apio Toolbox:

Apio is a python-based multiplatform toolbox to verify, synthesize, simulate and upload your verilog designs. The following list contains some of Apio commands with corresponding tools it uses.

Step#	Command	Description
1	apio verify	Syntax verification for the HDL code.
2	apio sim	It is used for simulation using a testbench.
3	apio build	It is used for Synthesis and Impementation of the HDL design. It calls yosys + nextPNR + icepack tools.
4	apio upload	It is used for uploading the implanted design to FPGA. It calls iceprog tool.

Procedure:

1. Installation of the Apio toolbox

First install Python3 and then go through the following the set of commands on command prompt in Windows platform.

```
python –version                    // Checks python version. Python 3.5+ is better.
python -m pip install apio==0.6.7  // Install Apio v0.6.7
apio –version                      // Checks Apio version apio install --all
                                   // Install required open-source FPGA tools this may take an hour.
apio install drivers               // Install drivers for the board. apio
drivers --ftdi-enable              // FTDI drivers configuration apio
system --lsftdi                    // Lists connected FTDI device.
```

2. Create your HDL design files.

Make a directory for your HDL design, say F:\apio\eq2>

C:\Users\Admin>**f:**
F:\>**mkdir apio\eq2**
F:\>**cd apio\eq2**

Here you may create an HDL design file using a text editor. For the reference purpose, listings 1.1 (eq1), 1.4 (eq2), 1.7 (eq2_testbench) are used here. These design files could be downloaded from https://github.com/fpga-logi/logi-pong-chuexamples/tree/master/pong-chu-logi-edu-examples-verilog/ch01 and add to your eq2 folder.

```
f:\apio\eq2>dir
 Volume in drive F has no label.
 Volume Serial Number is D059-074D

 Directory of f:\apio\eq2

01/20/2022  10:09 AM    <DIR>          .
01/20/2022  10:09 AM    <DIR>          ..
01/17/2022  01:00 PM               286 eq1.v
01/17/2022  12:58 PM               386 eq2.v
01/19/2022  05:37 PM             1,283 eq2_tb.v
               3 File(s)          1,955 bytes
               2 Dir(s)  118,226,653,184 bytes free
```

Figure 2: HDL design files.

For the testbench file to work with GTKWave, we need to modify the example testbench file as follow:

1. Rename module to eq2_tb() and save as eq2_tb.v
2. Add these two lines after initial begin
   $dumpfile("eq2_tb.vcd"); // simulator generates output file for the waveforms data
   $dumpvars;
3. Replace $stop with $finish.

Now you may execute the following commands:

F:\apio\eq2>**apio verify**

It is highly likely that an error of a missing board file appears. You may then enter the command **apio init --board iCEBreaker** and then again verify your design. After success, you may proceed to the simulation step.

3. Simulate an HDL design and analyse on GTKWave.

F:\apio\eq2>**apio sim**

After the simulation, vcd file is generated with simulated data. Next, GTKWave will appear vcd file data. Choose the signals of interest and drag them to the Signals area as shown in fig 3. You may then study the behavior of signals in Waves section.

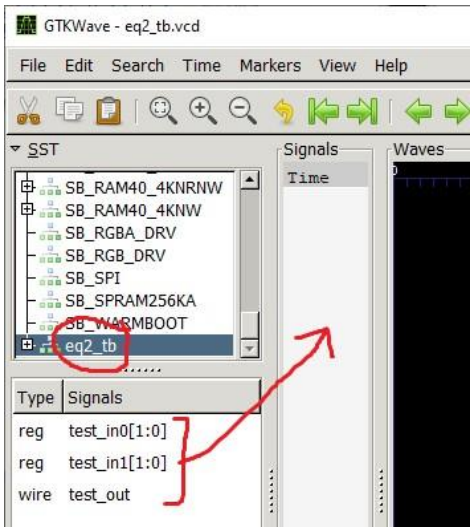
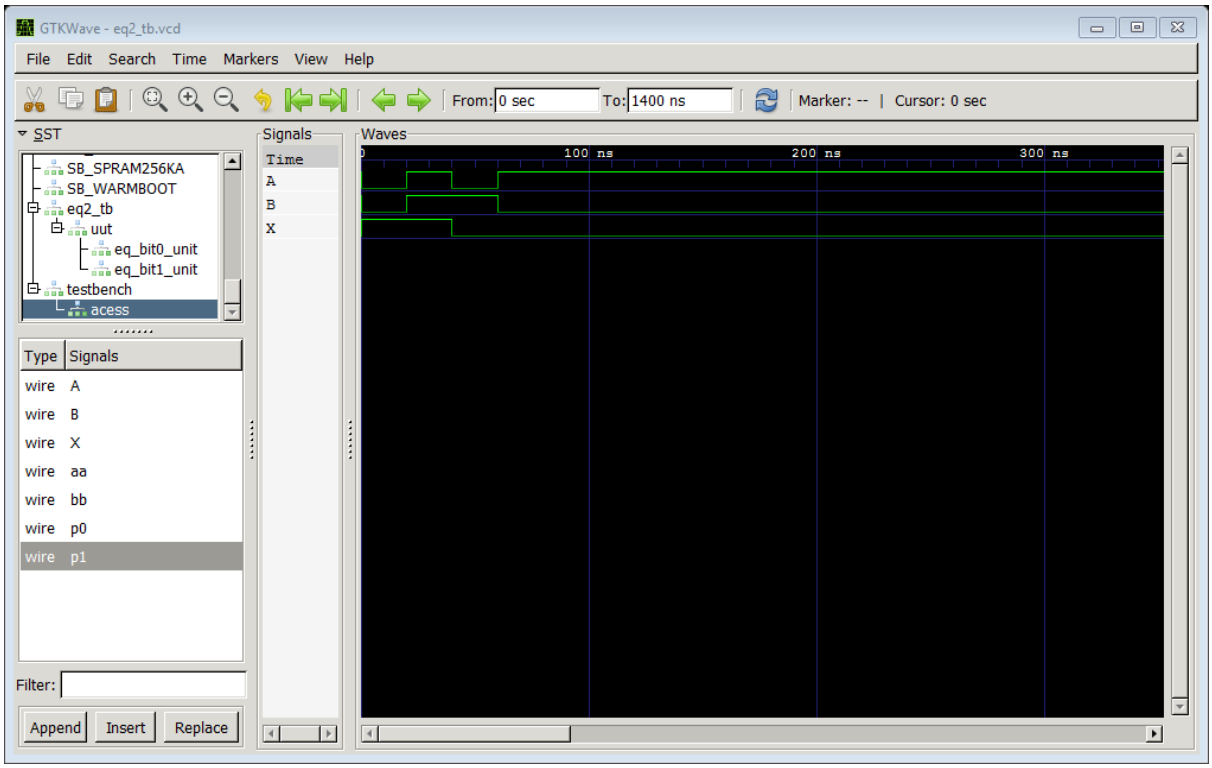


Figure 3: GTKWave signals of interest.

Lab Exercises:

Make a combinational logic circuit using 5 gates of your choice with its testbench and simulate its behaviour in GTKWave. Submit your analysis report.

Results:- Using basic gates we made a **XNOR** gate with test bench. After that test bench is simulated in GTK Wave. The results are attached below. Observation and analytic for apio GTKWave are well defined.




```
========================================== [SUCCESS] Took 0.86 seconds ==========================================

E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\eq2>apio sim
C:/Users/M.Bytes/.apio
iverilog -o eq2_tb.out -D VCD_OUTPUT=eq2_tb C:/Users/M.Bytes/.apio\packages\toolchain-yosys\share\yosys/ice40/cells_sim.
v basicgate.v basicgate_test.v eq1.v eq2.v eq2_tb.v
vvp eq2_tb.out
VCD info: dumpfile eq2_tb.vcd opened for output.
gtkwave eq2_tb.vcd eq2_tb.gtkw
```

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#10

Open-Source FPGA Tools: Introduction to Icestudio

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#10

Open-Source FPGA Tools: Introduction to Icestudio

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali

033-18-0037

Lab Learning Objectives:

After completing this session, student should be able to:

- Download and install Icestudio.
- Upload an HDL design to the FPGA IC using Icestudio.

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	ϒ Desired output	ϒ Minor mistakes	ϒ Critical mistakes	
2	Timing	ϒ Submitted within the given time	ϒ 1 day late	ϒ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. Desktop/Laptop Computer
2. Open-source FPGA hardware development board

Background Theory:

Icestudio is a design tool focused for newcomers to the FPGA world. It is a GUI interface where the users with limited HDL programming skills could work with readily available blocks.

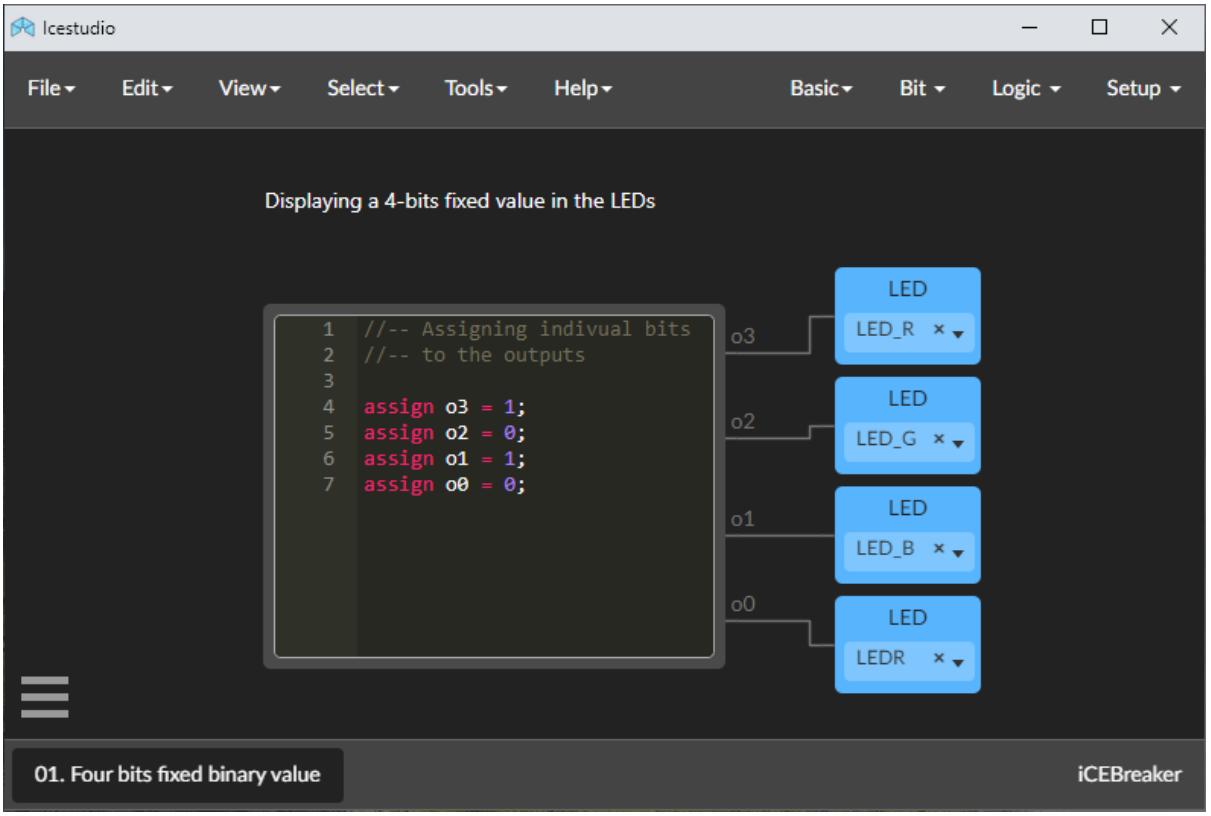


Fig 1: Icestudio interface with an example program

Source: https://github.com/FPGAwars/icestudio/wiki

Procedure:

1. Installation of Icestudio

- Please download a stable version of Icestudio compatible with Win64 from https://icestudio.io/ and install.
- Update the toolchain (Tools > Toolchain > Update)
- Enable drivers (Tools > Drivers > Enable)

2. Create your HDL design files.

Explore the design blocks tab, i.e. Basic, Build and Logic etc. Conversely you may also start exploring from available examples. For examples, load a program, where LED may turn on while pressing a button, from File>Examples>Basic>Pushbutton and LED.

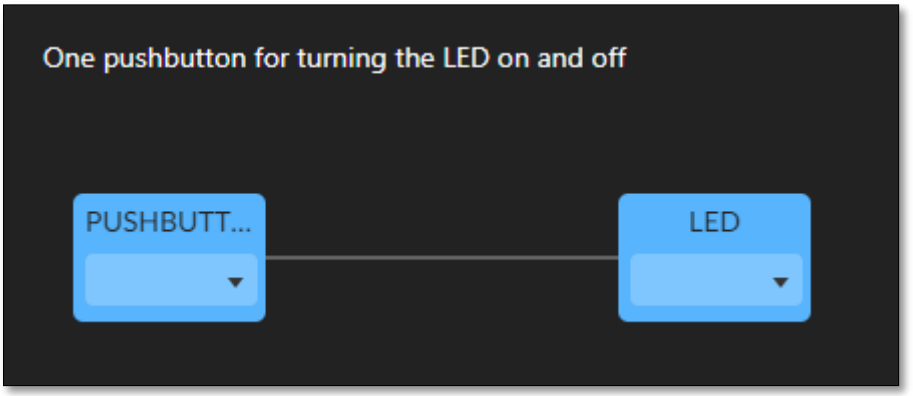


Figure 2: HDL design example.

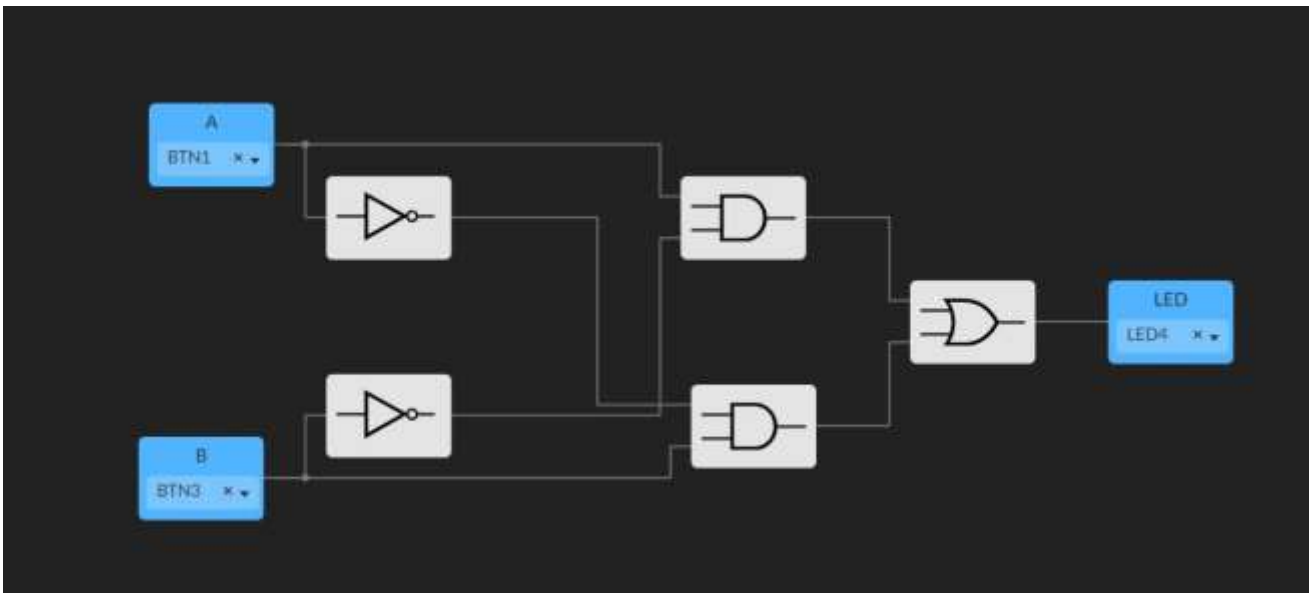
Choose BTN from the dropdown list of PUSHBUTTON and LEDG from the list under LED.

3. Download your HDL design to the development board.

Go though the process of Verify>Build>Upload from Tools tab. Make sure you have the development board connected. Once your design is uploaded, you may check if it works accordingly.

Lab Exercises:

Make a combinational logic circuit using 5 gates of your choice and upload the design to the board. Submit your report.



Attached circuit is about XNOR gate. The circuit is built on Icestudio. The ouput of XNOR gate is given to the LEDs of open source FPGA tool.

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Handout#11

Open-Source FPGA Tools: Icarus Verilog and GTKWave

Student's name: Sajjad Ali Enrollment ID: 033-18-0037

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Department of
Electrical Engineering

Digital System Design (ESE-412)

Handout#11

Open-Source FPGA Tools: Icarus Verilog and GTKWave

Instructor: Dr. Nabeel Siddiqui

Sajjad Ali

Lab Learning Objectives:

After completing this session, student should be able to:

- Download and install Icarus.
- Compile an HDL design on Icarus.
- Simulate/Visualize using GTKWave.

Lab Report Rubrics					Total Marks
	Criterion	**0.5**	**0.25**	**0.125**	
1	Accuracy	ϒ Desired output	ϒ Minor mistakes	ϒ Critical mistakes	
2	Timing	ϒ Submitted within the given time	ϒ 1 day late	ϒ More than 1 day late	

Note: Submit this lab hand-out in the next lab with attached solved activities and exercises

Lab Hardware and Software Required:

1. Desktop/Laptop Computer
2. Open-source FPGA hardware development board

Background Theory:

Icarus Verilog is a Verilog simulation and synthesis tool. It operates as a compiler, compiling source code written in Verilog (IEEE-1364) into some target format.

GTKWave is an analysis tool used to perform debugging on Verilog or VHDL simulation models. It has been developed to perform debug tasks on large systems on a chip and has been used in this capacity as an offline replacement for third-party debug tools.

Source: http://iverilog.icarus.com/
http://gtkwave.sourceforge.net/gtkwave.pdf

Procedure:

1. Installation of Icarus

- Please download a stable version of Icarus compatible with Win64 from https://bleyer.org/icarus/ and install. Make sure you add the executables directory to the PATH variable.
  iverilog-0.9.7_setup.exe (latest stable release) [10.5MB]

- Add Environment variable:
  - Go Control Panel > Systems > Advanced Systems Settings > Environmental Variables
  - Find 'Path' row and edit > Click NEW
  - Paste bin folder path in iverilog: C:\iverilog\bin
  - Click OK and Exit

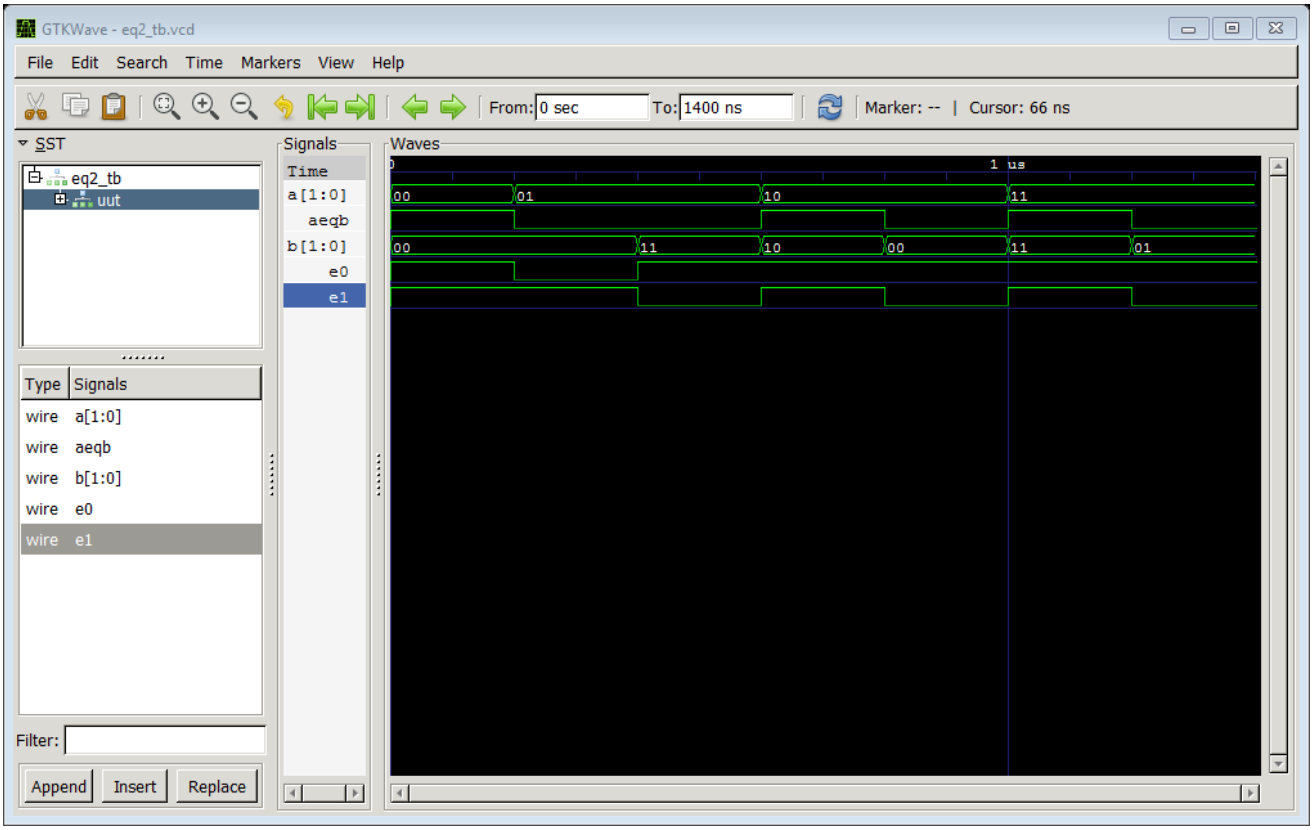
2. Use your HDL design files for compilation and visualize in GTKWave.

Copy sample listings eq1.v, eq2.v and eq2_tb.v in a folder.

Use command prompt to execute the following commands in this folder:

- `iverilog -o mycomp2 eq2_tb.v eq2.v eq1.v` `// Creates object file.`
- `vvp mycomp2` `// Object file to VCD file for gtkwave`
- `gtkwave eq2_tb.vcd` `// Visualize`

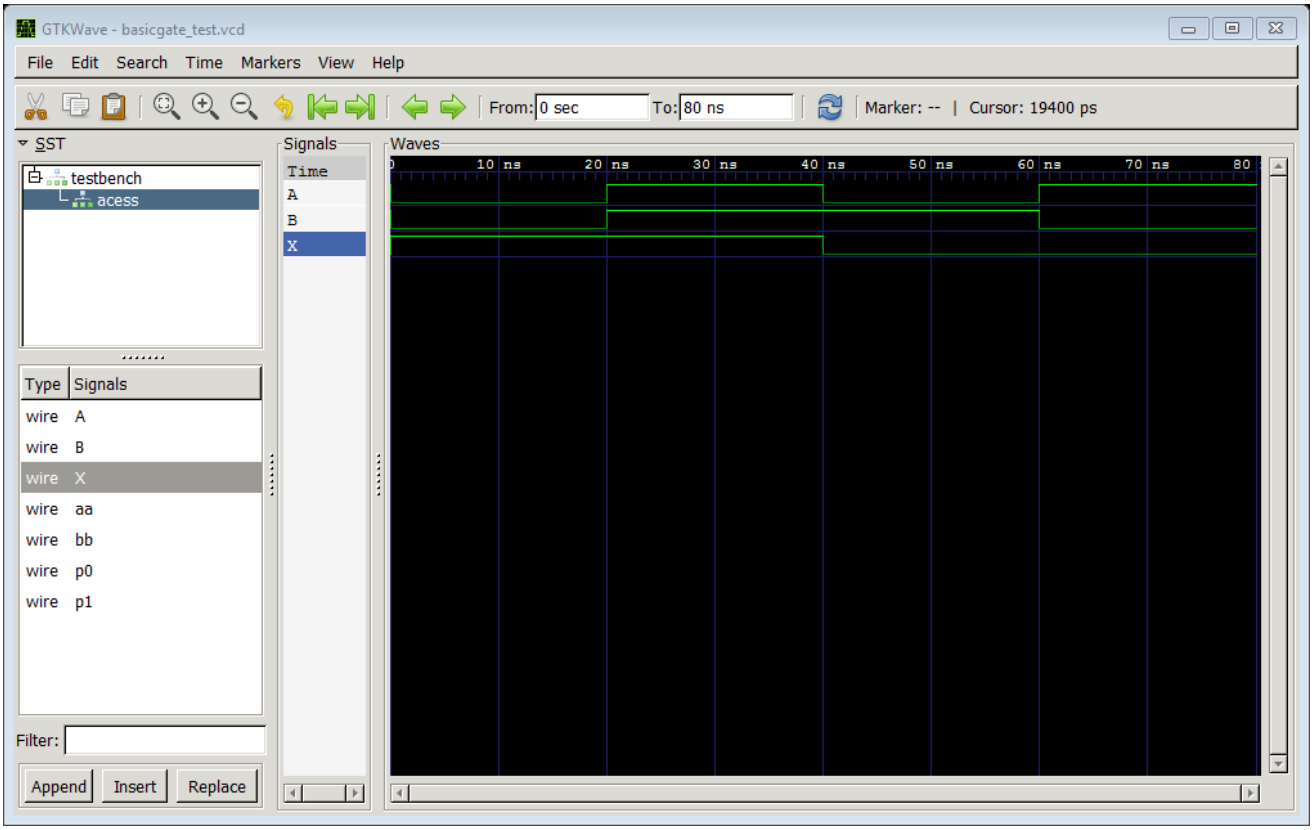
```
========================================== [SUCCESS] Took 3.18 seconds ==========================================

E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\eq2>iverilog -o mycomp2 eq2_tb.v eq2.v
 eq1.v

E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\eq2>vvp mycomp2
VCD info: dumpfile eq2_tb.vcd opened for output.

E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\eq2>gtkwave eq2_tb.vcd

GTKWave Analyzer v3.3.48 (w)1999-2013 BSI

[0] start time.
[1400000] end time.
```



Lab Exercises:

Make a combinational logic circuit using 5 gates of your choice, simulate the design using a testbench and analyse on GTKWave. Submit your report.

```
E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\ghk>iverilog -o mycomp2 basicgate.v ba
sicgate_test.v

E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\ghk>vvp mycomp2
VCD info: dumpfile basicgate_test.vcd opened for output.

E:\Sajjad Documents\Semester Materials\Semester 7\Digital System Design\apio\apio\ghk>gtkwave basicgate_test.vcd

GTKWave Analyzer v3.3.48 (w)1999-2013 BSI

[0] start time.
[80000] end time.
```



In above lab, we use Icarus compiler. The exercise circuit is about xnor gate which gives the high signal at same inputs i-e 00 or 11. And we run the circuit simulation on GTKWave. In nut a shell we use the open circuit tool for this circuit names Icarus as a compiler.

Department of
Electrical Engineering

Digital System Design (ESE-412)

Instructor: Dr. Nabeel Siddiqui

Lab Report#12:

<Security Alarm System >

Open-Ended Lab

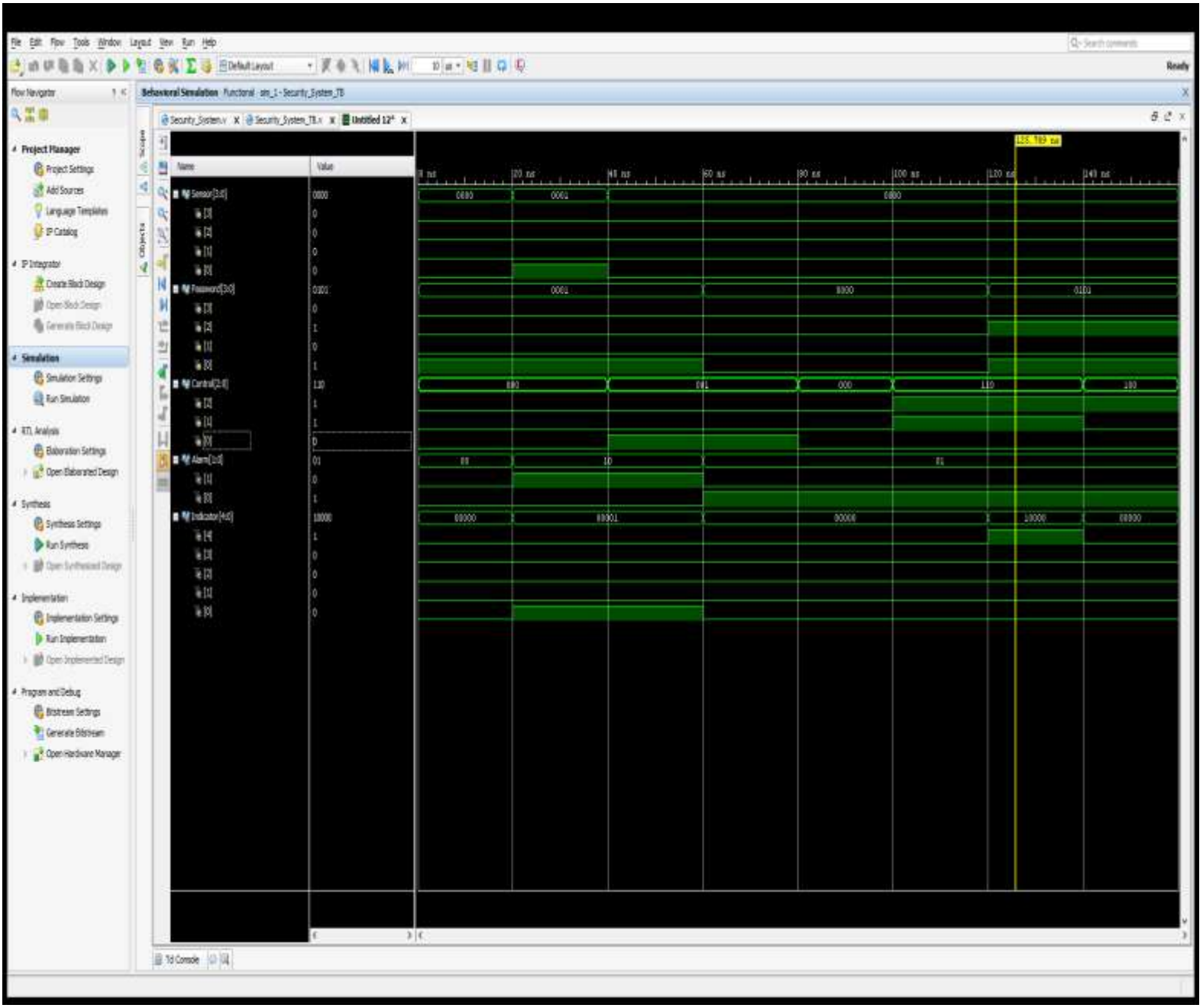
Student's name: Haris Nawaz & Sajjad ALi

LABORATORY PERFORMANCE RUBRIC

Criteria	Performance Indicator	Exemplary (100%)	Satisfactory (80%)	Developing (60%)	Unsatisfactory (0%)
Safety Rules (25 Points)		Experiment is carried out with full attention to relevant safety procedures.	Experiment is generally carried out with attention to relevant safety procedures.	Experiment is carried out with some attention to relevant safety procedures.	Safety Procedures were ignored. Always needs assistance.
Identify/Use/ Wiring of Equipment/Components (25 Points)	**CLO-3 PLO-5**	Always Identifies/Uses/Wires the equipment/component properly and explores it.	Always Identifies/Uses/Wires the equipment/component properly.	Identifies/Uses/Wires the equipment/component properly sometimes. Some assistance is needed.	Fails to identify/use and wire the equipment/component properly most of the time. Always needs assistance.
Experiment Performance (25 Points)	**CLO-3 PLO-5**	Quite able to perform the entire experiment with negligible help from the instructor.	Able to perform the experiment with some help from the instructor.	Able to perform the experiment with a lot help from the instructor.	Unable to perform the experiment, the lab instructor helps at almost every step.
Analysis and Interpretations (25 Points)	**CLO-3 PLO-5**	Analyses and interprets data/results correctly and precisely.	Analyzes and interprets the data/results correctly with minor mistakes.	Analyzes and interprets the data/results with major mistakes.	Unable to analyzes and interpret the data/results.
Total Points Obtained:					

Security Alarm System

In this lab, we have worked on Security of any office. We setup 4 indicators and 4sensors on the entrance door of office. And also setup 2 alarms which are tagged at near counter and security corner. The control of all the indicators and alarms are at head of office. If any stranger passes by sensor the indicator value trigger to 1 and alarm will turn on. The head of the office have access to login the access with password to turn off the alarm (reset alarm) and reverse back the indicator value. For password, one should know the first password which is setup already in the system. If any emergency situation the alarm will be not reset till second password not match with 1st password.



CODE

```
`timescale 1ns / 1ps



module Security_System(Alarm, Indicator, Sensor, Password, Control);


input  [3:0]Sensor;

input  [3:0]Password;

input  [2:0]Control;


output reg [1:0]Alarm;

output reg [4:0]Indicator;


reg [3:0]Password_Memory[1:0] = {4'b0000,4'b0000 };      //Two Passwords



wire Alarm_Reset = Control[0];      // Control Input for Alarm Reset

wire Password_Reset = Control[1];   // Control Input for passwrod reset

wire Authority = Control[2];        //Control Input for Knowing Authority Personnel


initial

begin
```

```
    Indicator <= 5'b00000;
    Alarm <= 2'b00;
end


always@*
begin
    if(Sensor[0] | Sensor[1] | Sensor[2] | Sensor[3])
        begin
        Alarm[1] <= 1'b1;        //RED Light ON
        Alarm[0] <= 1'b0;        //GREEN Light OFF
        end
    case(Sensor)
    4'b0001: Indicator[0] <= 1'b1;
    4'b0010: Indicator[1] <= 1'b1;
    4'b0100: Indicator[2] <= 1'b1;
    4'b1000: Indicator[3] <= 1'b1;
    endcase

    if(Alarm_Reset)
       if(Password == Password_Memory[0] | Password == Password_Memory[1])
          if(~Sensor[0] & ~Sensor[1] & ~Sensor[2] & ~Sensor[3])
          begin
          Alarm[1] <= 1'b0;       //RED Light OFF
```

```
        Alarm[0] <= 1'b1;       //GREEn LIGHT ON
        Indicator[3:0] <= 4'b0000;  //Turning Sensor Indicators LOW
        end

  if(Password_Reset)
    begin                  //1st If starts here
    Indicator[4] <= 1'b1;

    if(Password == Password_Memory[0] & ~Authority)
    begin
    Password_Memory[0] <= Password;
    Indicator[4] <= 1'b0;
    end

    if(Password == Password_Memory[1] &  Authority)
    begin
    Password_Memory[1] <= Password;
    Indicator[4] <= 1'b0;
    end

    end                    //1st if Ends here
   else Indicator[4] <= 1'b0;
end
```

```
endmodule
```

TESTBENCH

```
`timescale 1ns / 1ps


module Security_System_TB();


reg  [3:0]Sensor;

reg  [3:0]Password;

reg  [2:0]Control;


wire [1:0]Alarm;

wire [4:0]Indicator;


Security_System inst0(Alarm, Indicator, Sensor, Password, Control);


initial

begin


  Password <= 4'b0001;  //Current/Initial Password

  Control <= 3'b000;

end
```

```
initial

begin

   #00 Sensor <= 4'b0000;

   #20 Sensor <= 4'b0001;

   #20 Sensor <= 4'b0000; Control[0] <= 1'b1;      //Alarm Reset Raised

   #20 Password <=4'b0000;                        //Password made 0000 to give reset access

   #20 Control[0] <= 1'b0;                        //Alarm Reset gets LOW

   #20 Control[1] <= 1'b1; Control[2] <= 1'b1;     //Password Reset Control Raised, Authority
Personnel Chosen to be "1"

   #20 Password <= 4'b0101;                       // New password shall be "0101"

   #20 Control[1] <= 1'b0;                        //passwrod Reset Control gets LOW

   #20 $stop;



end

endmodule
```