ATME College of Engineering

Department of Electronics and Communication Engineering

(ACADEMIC YEAR 2025-26)

LABORATORY MANUAL

SUBJECT: FPGA Based System Design Lab Using Verilog

SUB CODE: BECL657A

SEMESTER: VI

Institute Vision & Mission

VISION:

Development of academically excellent, culturally vibrant, socially responsible, and globally competent human resources

MISSION:

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional, and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence

Department Vision and Mission

VISION:

To develop highly skilled and globally competent professionals in the field of Electronics and Communication Engineering to meet industrial and social requirements with ethical responsibility

MISSION:

- To provide State-of-art technical education in Electronics and Communication at undergraduate and post-graduate levels to meet the needs of the profession and society and to adopt the best educational methods and achieve excellence in teaching-learning and research.
- To develop talented and committed human resource, by providing an opportunity for innovation, creativity, and entrepreneurial leadership with high standards of professional ethics, transparency, and accountability.
- To function collaboratively with technical Institutes/Universities/Industries and offer opportunities for Long-term interaction with academia and industry

PROGRAM OUTCOMES (POs)

Engineering Graduates will be able to:

PO1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6. The engineer and society: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to professional engineering practice.

PO7. Environment and sustainability: Understand the impact of professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

PO9. Individual and teamwork: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10.Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAM SPECIFIC OUTCOMES (PSOs)

- To have the capability to understand and adopt to technological advancements with the usage of modern tool to analyze and design embedded system or processes for variety of applications.
- To work effectively in a group as an independent visionary, team member and leader having the ability to understand the requirement and develop feasible solutions to emerge as potential core or electronic engineer.

PROGRAM EDUCATIONAL OBJECTIVES (PEO's)

- To Produce Graduates to excel in the profession, higher education and pursue research exercises in Electronics and Communication Engineering.
- To create technically able alumni with the capacity to examine, plan, create and execute Electronics and Communication frameworks thereby involving in deep rooted learning.

Introduction to HDL

An HDL is a programming language used to describe electronic circuit essentially digital logic circuits. It can be used to describe the operation, design and organization of a digital circuit. It can also be used to verify the behavior by means of simulations. The principle difference between HDL and other programming languages is that HDL is a concurrent language whereas the others are procedural i.e. single threaded. HDL has the ability to model multiple parallel processes like adders, flip-flops etc. which execute automatically and independently of each other. It is like building many circuits that can operate independently of each other.

The two widely used HDLs are:

- VHDL: Very High Speed Integrated Circuits HDL
- Verilog HDL

VHDL (**VHSIC Hardware Description Language**) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. **VHDL** can also be used as a general purpose parallel programming language.

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. It is also used in the verification of analog circuits and mixed-signal circuits, as well as in the design of genetic circuits.

Difference between Verilog and VHDL

1. VHDL is based on Pascal and ADA while Verilog is based on C language.
2. VHDL is strongly typed i.e., does not allow the intermixing, or operation of variables, with different classes whereas Verilog is weakly typed.
3. VHDL is case insensitive and Verilog is case sensitive.
4. Verilog is easier to learn compared to VHDL.
5. Verilog has very simple data types, while VHDL allows users to create more complex data types.
6. Verilog lacks the library management, like that of VHDL.

FPGA DESIGN FLOW

1. **Design Entry –** the first step in creating a new design is to specify it's structure and functionality. This can be done either by writing an HDL model using some text editor or drawing a schematic diagram using schematic editor.

2. **Design Synthesis** – next step in the design process is to transform design specification into a more suitable representation that can be further processed in the later stages in the design flow. This representation is called the netlist. Prior to netlist creation synthesis tool checks the model syntax and analyze the hierarchy of your design which ensures that your design is optimized for the design architecture you have selected. The resulting netlist is saved to a Native Generic Circuit (NGC) file (for Xilinx® Synthesis Technology (XST) compiler) or an Electronic Design Interchange Format (EDIF) file (for Precision, or Simplify /Simplify Pro tools).

3. **Design Implementation**

   Implementation step maps netlist produced by the synthesis tool onto particular device's internal structure. It consists from three steps:

   3.1 **Translate step** – merges all incoming netlists and constraints into a Xilinx Native Generic Database (NGD) file.

3.2 **Map step** - maps the design, specified by an NGD file, into available resources on the target FPGA device, such as LUTs, Flip-Flops, BRAMs,... As a result, an Native Circuit Description (NCD) file is created.

3.3 **Place and Route step** - takes a mapped Native Circuit Description (NCD) file, places and routes the design, and produces an NCD file that is used as input for bit stream generation.

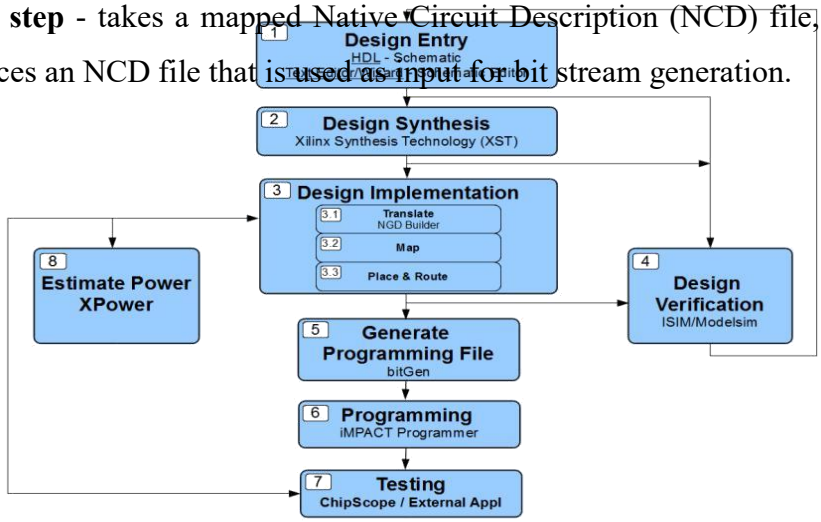


Figure: FPGA Design Flow

4. **Design Verification** – is very important step in design process. Verification is comprised of seeking out problems in the HDL implementation in order to make it compliant with the design specification. A verification process reduces to extensive simulation of the HDL code. Design Verification is usually performed using two approaches: Simulation and Static Timing Analysis.

   There are two types of simulation:

- **Functional (Behavioral) Simulation** – enables you to simulate or verify a code syntax and functional capabilities of your design. This type of simulation tests your design decisions before the design is implemented and allows you to make any necessary changes early in the design process. In functional (behavioral) simulation no timing information is provided.

- **Timing Simulation** – allows you to check does the implemented design meet all functional and timing requirements and behaves as you expected. The timing simulation uses the detailed information about the signal delays as they pass through various logic and memory components and travel over connecting wires. Using this information it is possible to accurately simulate the behaviour of the implemented design. This type of simulation is performed after the design has been placed and routed for the target PLD, because accurate signal delay information can now be estimated. A process of relating accurate timing information with simulation model of the implemented design is called Back-Annotation.
- **Static Timing Analysis** – helps you to perform a detailed timing analysis on mapped, placed only or placed and routed FPGA design. This analysis can be useful in evaluating timing performance of the logic paths, especially if your design doesn't meet timing requirements. This method doesn't require any type of simulation.

5. **Generate Programming File** – this option runs BitGen, the Xilinx bitstream generation program, to create a bitstream file that can be downloaded to the device.
6. **Programming** – iMPACT Programmer uses the output from the Generate Programming File process to configure your target device.

7. **Testing** – after configuring your device, you can debug your FPGA design using the Xilinx ChipScope Pro tool or some external logic analyzer.

8. **Estimate Power** – after implementation, you can use the XPower Analyzer for estimation and power analysis. XPower Analyzer is delivered with ISE Design Suite. With this tool you can estimate power, based on the logic and routing resources of the actual design.

ABOUT XILINX ISE SOTWARE

Xilinx ISE (Integrated Synthesis Environment) is a software tool produced by Xilinx for synthesis and analysis of HDL designs, enabling the developer to synthesize ("compile") their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer.

Xilinx ISE is a design environment for FPGA(Field programmable gate arrays) products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors. The Xilinx ISE is primarily used for circuit synthesis and design, while ISIM or the ModelSim logic simulator is used for system-level testing

STEPS TO EXECUTE A PROGRAM

1) Starting the ISE software

Start _ program _ XILINX ISE 7 _ Project Navigator

2) Creating a New Project in ISE

A project is a collection of all files necessary to create and to download a design to a selected FPGA or CPLD devices

Project name:

Project location:

Top-Level Source Type: HDL

Click **Next** to move to the project properties page.

3) Fill in the properties in the table as shown below

Device Family: Spartan 3

Device: XC3S50

Package: PQ208Speed

Speed: -5

Top-Level Module Type: HDL

HDL Synthesis Tool: XST(VHDL/VERILOG)

Simulator: ISE Simulator (VHDL/ Verilog)

4) Creating an HDL Source

Create a top-level HDL file for the design. Determine the language that you wish to use (Verilog module or VHDL module).

This simple AND Gate design has two inputs: A and B. This design has one output called C

Click New Source in the New Project Wizard to add one new source to your project.

a) Select **VERILOG MODULE** as the source type in the New Source dialog box.
b) Type in the file name **for ex:** and_gate
c) Verify that the Add to project checkbox is selected.
d) Click Next.
e) Define the ports for your Verilog source.

In the Port Name column, type the **port names** on three separate rows: A, B and C.
In the Direction column, indicate whether **each port is an input, output, or inout**.
For A and B, select in from the list. For C, select out from the list.

5) Click next in the Define Verilog Source dialog box.

6) Click Finish in the New Source Information dialog box to complete the new source file template. Click Next in the New Project Wizard. Click next again.

7) Click Finish in the New Project Information dialog box.

ISE creates and displays the new project in the Sources in Project window and adds the and_gate.v file to the project.

8) Double-click on the **and_gate.v** file in the Sources in Project window to open the Verilog file in the ISE Text Editor.

The and_gate.v file contains:

Module name with the inputs and outputs declared.

9) Add the relationship between input and output after the input and output declared in module. Save the file by selecting File > Save.

10) When the source files are complete, the next step is to check the syntax of the design. Syntax errors and typos can be found using this step.

    a) Select the counter design source in the **ISE Sources window** to display the related processes in the Processes for Source window.
    b) Click the "+"next to the **Synthesize-XST** process to expand the hierarchy.
    c) Double-click the **Check Syntax** process.

11) When an ISE process completes, you will see a status indicator next to the process name.

    a) If the process completed successfully, a **green check** mark appears.
    b) If there were errors and the process failed, a red X appears.
    c) A yellow exclamation point means that the process completed successfully, but some Warnings occurred.
    d) An orange question mark means the process is out of date and should be run again.
    e) Look in the Console tab of the Transcript window and read the output and status messages produced by any process that you run.

Caution! You must correct any errors found in your source files. If you continue without valid syntax, you will not be able to simulate or synthesize your design.

12) After the successful check syntax in the process Examine RTL diagrams.

13) To Create Testbench waveform, Right click on file name in source window, and_gate.v and add source.

14) Add testbench waveform source with a new file name and click next.

15) A timing window pops up. Click on combinatorial and click next.

16) A graphical window of input and output appears. Make changes according to the truth table and save.

17) <file_name>.tb file is added to the project.

18) In source window change implementation to behavioral simulation.

19) In process window click on Xilix ISE simulator and RUN. Output window appears. Analyze the wave forms according to the truth table.

20) Double-click the Assign Package Pins process found in the User Constraints process group. ISE runs the Synthesis and Translate step and automatically creates a User Constraints File(UCF). You will be prompted with the following message.

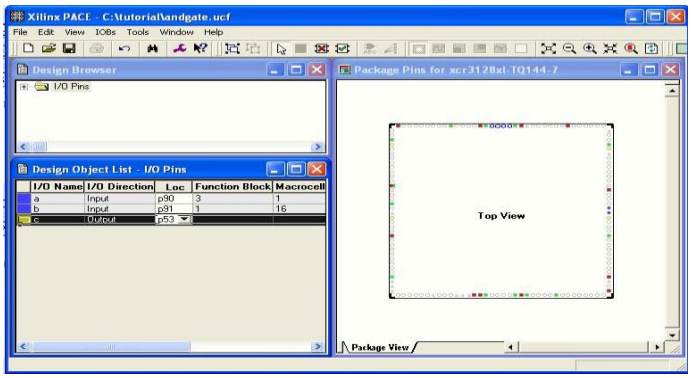
21) Click Yes to add the UCF file to your project. The file is added to your project and is visible in the Sources in Project.

22) Now the Xilinx Pin out and Area Constraints Editor (PACE) opens.

23) You can see your I/O Pins listed in the Design Object List window. Enter a pin location for each pin in the Loc column as specified below

A: P1, B:P2, C:P3

24) Click on the Package View tab at the bottom of the window to see the pins you just added. Put your mouse over grid number to verify the pin assignment.



25) Close PACE

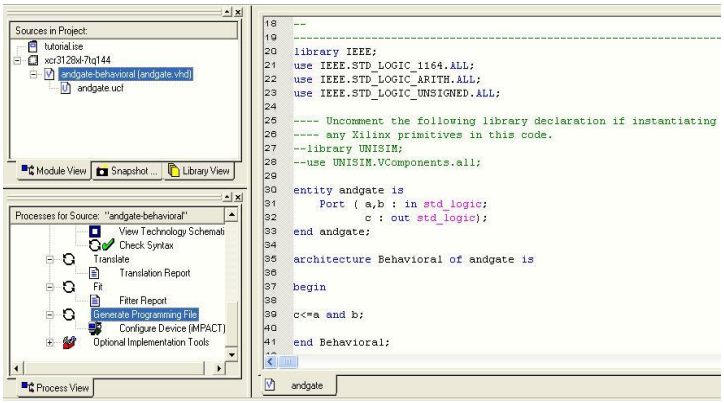
Creating Configuration Data

The Program File is a encoded file that is the equivalent of the design in a form that can be downloaded into the CPLD device.

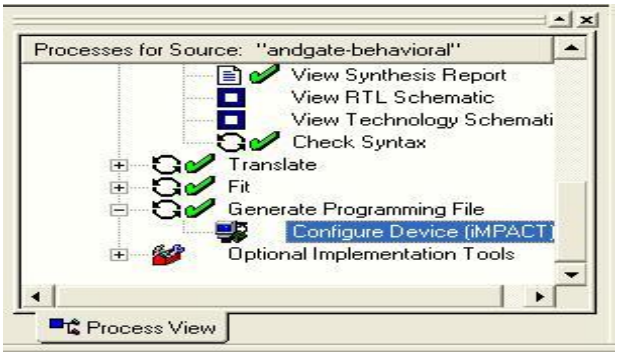
The final phase in the software flow is to generate a program file and configure the device

Generating a Program File

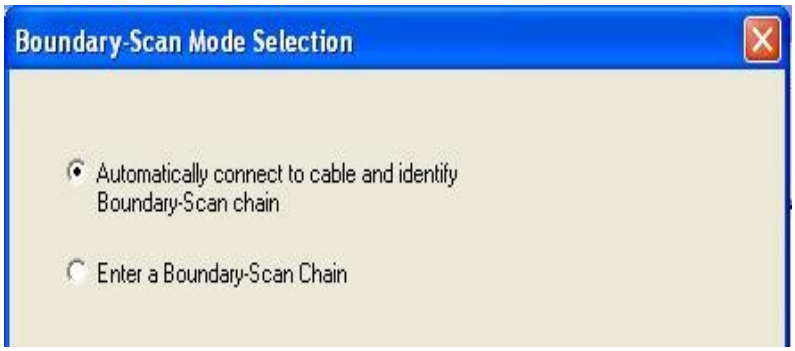
1. The Program File is created. It is written into a file called andgate.jed This is the actual configuration data
2. Double Click the Generate Programming File process located near the bottom of the Processes for Source window.



3. This section provides simple instructions for configuring a Spartan-3 xc3s200 device connected to your PC.
4. *Note:* Your board must be connected to your PC before proceeding. If the device on your board does not match the device assigned to the project, you will get errors. Please refer to the IMPACT Help for more information. To access the help, select Help > Help Topics
5. To configure the device:
6. Click the "+" sign to expand the Generate Programming File processes

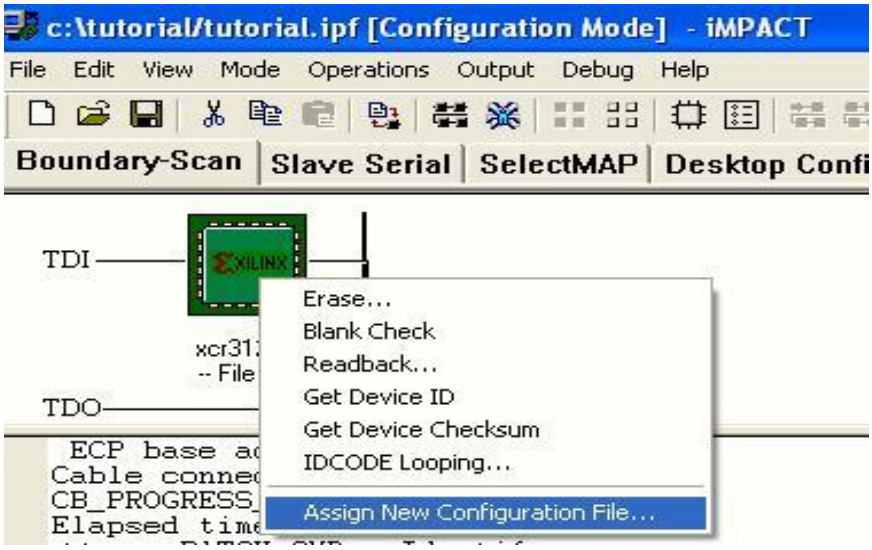


7. Double click on the Configure device IMPACT
8. In the Configure Devices dialog box, verify that Boundary-Scan Mode is selected and Click Next
9. Verify that Automatically connect to cable and identify Boundary-Scan chain is selected and click Finish.



10. If you get a message saying that there was one device found, click OK to continue

11. The iMPACT will now show the detected device, right click the device and select New Configuration File.



12. The Assign New Configuration File dialog box appears. Assign a configuration file to each device in the JTAG chain. Select the andgate.jed file and click Open

13. Right-click on the counter device image, and select Program... to open the Program Options dialog box.
14. Click OK to program the device. ISE programs the device and displays Programming Succeeded if the operation was successful
15. Close IMPACT without saving

FPGA Based System design Lab Using Verilog

Course Code: **BECL657A** CIE Marks: **50**
SEE Marks: **50**

1. Write a Verilog description for the following combinational logic, verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. Structural modeling of Full adder using two half adders and or Gate
    b. BCD to Excess-3 code converter

2. Write a Verilog description for the following Sequential Circuits, Verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. Mod-N counter
    b. Random sequence counter

3. Write a Verilog description for the following Sequential Circuits, Verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. SISO and PISO shift register
    b. Ring counter

4. Write a Verilog description for the following Digital Circuits, Verify the functionality using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. 4-Bit Ripple Carry Adder
    b. 4-Bit Linear Feedback shift register

5. Write a Verilog description for the following Digital Circuits, Verify the functionality using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. 4-bit Array Multiplication
    b. 4-bit Booth Multiplication@12112024

6. Write a Verilog description to design a clock divider circuit that generates $^1/_2$, $^1/_3{}^{rd}$ and $^1/_4{}^{th}$ clock from a given input clock. Port the design to FPGA and validate the functionality using output device.

7. Interface a Stepper motor to FPGA and Write a Verilog description to control Stepper motor rotation.

8. Interface a DAC to FPGA and Write a Verilog description to generate Square wave of frequency F KHz. Modify the code to down sample the frequency to F/2 KHz. Display the original and down sampled signals by connecting them to an output device.

9. Write a Verilog description to convert an analog input of a sensor to digital form and to display the same on a suitable display like set of simple LEDs like 7-Segment display digits.

FPGA Based System design Lab Using Verilog

Course Code: **BECL657A** CIE Marks: **50**
SEE Marks: **50**

Cycle – 1

1. Write a Verilog description for the following combinational logic, verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. Structural modeling of Full adder using two half adders and or Gate
    b. BCD to Excess-3 code converter
2. Write a Verilog description for the following Sequential Circuits, Verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. Mod-N counter
    b. Random sequence counter
3. Write a Verilog description for the following Sequential Circuits, Verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. SISO and PISO shift register
    b. Ring counter

Cycle – 2

4. Write a Verilog description for the following Digital Circuits, Verify the functionality using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. a.4-Bit Ripple Carry Adder
    b. 4-Bit Linear Feedback shift register
5. Write a Verilog description for the following Digital Circuits, Verify the functionality using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.
    a. 4-bit Array Multiplication
    b. 4-bit Booth Multiplication@12112024
6. Write a Verilog description to design a clock divider circuit that generates $^1/_2$, $^1/_3{}^{rd}$ and $^1/_4{}^{th}$ clock from a given input clock. Port the design to FPGA and validate the functionality using output device.

Cycle – 3

7. Interface a Stepper motor to FPGA and Write a Verilog description to control Stepper motor rotation.
8. Interface a DAC to FPGA and Write a Verilog description to generate Square wave of frequency F KHz. Modify the code to down sample the frequency to F/2 KHz. Display the original and down sampled signals by connecting them to an output device.
9. Write a Verilog description to convert an analog input of a sensor to digital form and to display the same on a suitable display like set of simple LEDs like 7-Segment display digits.

DETAILS OF ON BOARD CONNECTIONS AND SPARTAN – 3 IC		
Signal	**CN9 Connection**	**XC3S50 - SPARTAN**
AN1	PIN 5	PIN 2
AN2	PIN 6	PIN 3
AN3	PIN 7	PIN 7
AN4	PIN 8	PIN 9
Signal	**CN12 Connection**	**XC3S50 - SPARTAN**
AN5	PIN 37	PIN 166
AN6	PIN 38	PIN 167
Signal	**CN9 Connection**	**XC3S50 - SPARTAN**
SEG 1/RS	PIN 9	PIN 10
SEG 2/RW	PIN 10	PIN 11
SEG 3/EI	PIN 11	PIN 12
SEG 4/DT0	PIN 12	PIN 13
SEG 4/DT1	PIN 13	PIN 15
SEG 4/DT2	PIN 14	PIN 16
SEG 4/DT3	PIN 15	PIN 18
SEG 8	PIN 16	PIN 19
IN 1	PIN 20	PIN 21
IN 2	PIN 22	PIN 27
IN 3	PIN 24	PIN 29
IN 4	PIN 26	PIN 35
IN 5	PIN 28	PIN 37
IN 6	PIN 30	PIN 40
IN 7	PIN 32	PIN 43
IN 8	PIN 34	PIN 45
IN 9	PIN 36	PIN 48
IN 10	PIN 38	PIN 52
Signal	**CN10 Connection**	**XC3S50 - SPARTAN**
IN 11	PIN 6	PIN 58
IN 12	PIN 8	PIN 62
IN 13	PIN 10	PIN 64
IN 14	PIN 12	PIN 67
IN 15	PIN 14	PIN 71
IN 16	PIN 16	PIN 74
IN 17	PIN 24	PIN 58
IN 18	PIN 26	PIN 62
IN 19	PIN 28	PIN 64
IN 20	PIN 30	PIN 67
IN 21	PIN 32	PIN 71
IN 22	PIN 34	PIN 74
IN 23	PIN 36	PIN 74
Signal	**CN11 Connection**	**XC3S50 - SPARTAN**
IN 24	PIN 4	PIN 107
IN 25	PIN 6	PIN 113
IN 26	PIN 8	PIN 115

IN 27	PIN 10	PIN 117
IN 28	PIN 12	PIN 120
IN 29	PIN 14	PIN 123
IN 30	PIN 16	PIN 125
IN 31	PIN 18	PIN 131
IN 32	PIN 20	PIN 133
Signal	**CN9 Connection**	**XC3S50 - SPARTAN**
OPLED 1	PIN 19	PIN 20
OPLED 2	PIN 21	PIN 26
OPLED 3	PIN 23	PIN 28
OPLED 4	PIN 25	PIN 34
OPLED 5	PIN 27	PIN 36
OPLED 6	PIN 29	PIN 39
OPLED 7	PIN 31	PIN 42
OPLED 8	PIN 33	PIN 44
OPLED 9	PIN 35	PIN 46
OPLED 10	PIN 37	PIN 51
Signal	**CN10 Connection**	**XC3S50 - SPARTAN**
OPLED 11	PIN 5	PIN 57
OPLED 12	PIN 7	PIN 61
OPLED 13	PIN 9	PIN 63
OPLED 14	PIN 11	PIN 65
OPLED 15	PIN 13	PIN 68
OPLED 16	PIN 15	PIN 72
OPLED 17	PIN 23	PIN 78
OPLED 18	PIN 25	PIN 81
OPLED 19	PIN 27	PIN 85
OPLED 20	PIN 29	PIN 87
OPLED 21	PIN 31	PIN 93
OPLED 22	PIN 33	PIN 95
OPLED 23	PIN 35	PIN 101
Signal	**CN11 Connection**	**XC3S50 - SPARTAN**
OPLED 24	PIN 3	PIN 44
OPLED 25	PIN 5	PIN 46
OPLED 26	PIN 7	PIN 51
OPLED 27	PIN 9	PIN 36
OPLED 28	PIN 11	PIN 39
OPLED 29	PIN 13	PIN 42
OPLED 30	PIN 15	PIN 44
OPLED 31	PIN 17	PIN 46
OPLED 32	PIN 19	PIN 51
10MHzCLK	PIN 18	PIN 79
CLK1	PIN 19	PIN 76
CLK2	PIN 20	PIN 77
GREEN_E	PIN 23	PIN 187
RED_E	PIN 24	PIN 185

GREEN_S	PIN 25	PIN 184
RED_S	PIN 27	PIN 183
GREEN_W	PIN 28	PIN 194
RED_W	PIN 29	PIN 191
Hexa Keypad & Seven Segment Display (UCF)	**Gnerator of waveform (UCF)**	**Elevator (UCF)**
NET"clk"LOC="p79"	NET"clk"LOC="p79"	NET"pclk"LOC="p79"
NET"disp<0>"LOC="p10"	NET"rst"LOC="p21"	NET"pdspseg<0>"LOC="p176"
NET"disp<1>"LOC="p11"	NET"dout<0>"LOC="p187"	NET"pdspseg<1>"LOC="p176"
NET"disp<2>"LOC="p12"	NET"dout<1>"LOC="p185"	NET"pdspseg<2>"LOC="p176"
NET"disp<3>"LOC="p13"	NET"dout<2>"LOC="p190"	NET"pdspseg<3>"LOC="p176"
NET"disp<4>"LOC="p15"	NET"dout<3>"LOC="p189"	NET"fircal<0>"LOC="p183"
NET"disp<5>"LOC="p16"	NET"dout<4>"LOC="p194"	NET"fircal<1>"LOC="p184"
NET"disp<6>"LOC="p18"	NET"dout<5>"LOC="p191"	NET"fircal<2>"LOC="p181"
NET"disp_ent<0>"LOC="p2"	NET"dout<6>"LOC="p197"	NET"fircal<3>"LOC="p182"
NET"disp_ent<1>"LOC="p3"	NET"dout<7>"LOC="p196"	NET"crnt_fir<0>"LOC="p189"
NET"disp_ent<2>"LOC="p7"	**STEPPER MOTOR (UCF)**	NET"crnt_fir<1>"LOC="p190"
NET"disp_ent<3>"LOC="p9"	NET"clk"LOC="p79"	NET"crnt_fir<2>"LOC="p185"
NET"read_1_in<3>"LOC="p147"	NET"rst"LOC="p21"	NET"crnt_fir<3>"LOC="p187"
NET"read_1_in<2>"LOC="p146"	NET"dir"LOC="p29"	**DC MOTOR (UCF)**
NET"read_1_in<1>"LOC="p144"	NET"dout<0>"LOC="p169"	NET"clk"LOC="p79"
NET"read_1_in<0>"LOC="p143"	NET"dout<1>"LOC="p175"	NET"pdcm"LOC="p203"
NET"cscan<3>"LOC="p141"	NET"dout<2>"LOC="p176"	NET"psw<0>"LOC="p21"
NET"cscan<2>"LOC="p140"	NET"dout<3>"LOC="p178"	NET"psw<1>"LOC="p27"
NET"cscan<1>"LOC="p139"	**Clock devider**	NET"psw<2>"LOC="p29"
NET"cscan<0>"LOC="p138"	NET"clk"LOC="p79"	**CN - 10**
	NET"new_clk"LOC="p57"	5
	NET"clk_by_2"LOC="p61"	7
	NET"clk_by_3"LOC="p63"	9
	NET"clk_by_4"LOC="p65"	11

Experiment 1a:

Logic Diagram:

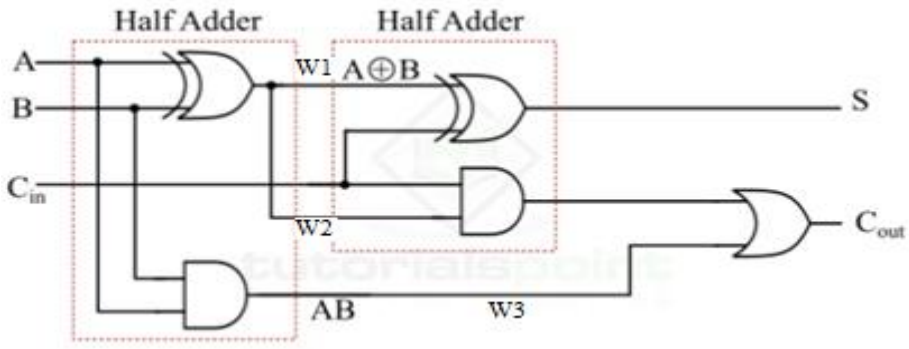


Figure 1.1: Logic Diagram for Full Adder Using Two Half Adders

Truth Table:

INPUTS			OUTPUTS	
A	**B**	**Cin**	**S**	**Cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Date:

Experiment-1: Write a Verilog description for the following combinational logic, verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.

a. Structural modeling of Full adder using two half adders and or Gate
b. BCD to Excess-3 code converter

Objective: To verify and synthesize the structural modeling of Full adder and BCD Excess-3 Code Converter using FPGA

Full Adder Module:

```
module fulladder (A, B, Cin, Sum, Carry);
input A, B, Cin;
output Sum, Carry;
wire W1,W2,W3;
halfadder h1(.A(A),.B(B),.S(W1),.Cout(W2));
halfadder h2(.A(W1),.B(Cin),.S(Sum),.Cout(W3));
assign Carry=W2|W3;
endmodule

module halfadder (A, B, S, Cout);
input A, B;
output S, Cout;
xor g1(S, A, B);
and g2(Cout, A, B);
endmodule
```

User Constraint File (UCF):

```
NET "A"  LOC = "p21" | IOSTANDARD = LVTTL ;
NET "B"  LOC = "p27" | IOSTANDARD = LVTTL ;
NET "Cin"  LOC = "p29" | IOSTANDARD = LVTTL ;
NET "S<0>"  LOC = "p20" | IOSTANDARD = LVTTL ;
NET "Cout<1>"  LOC = "p26" | IOSTANDARD = LVTTL ;
```

Test Bench Waveform for Full Adder:

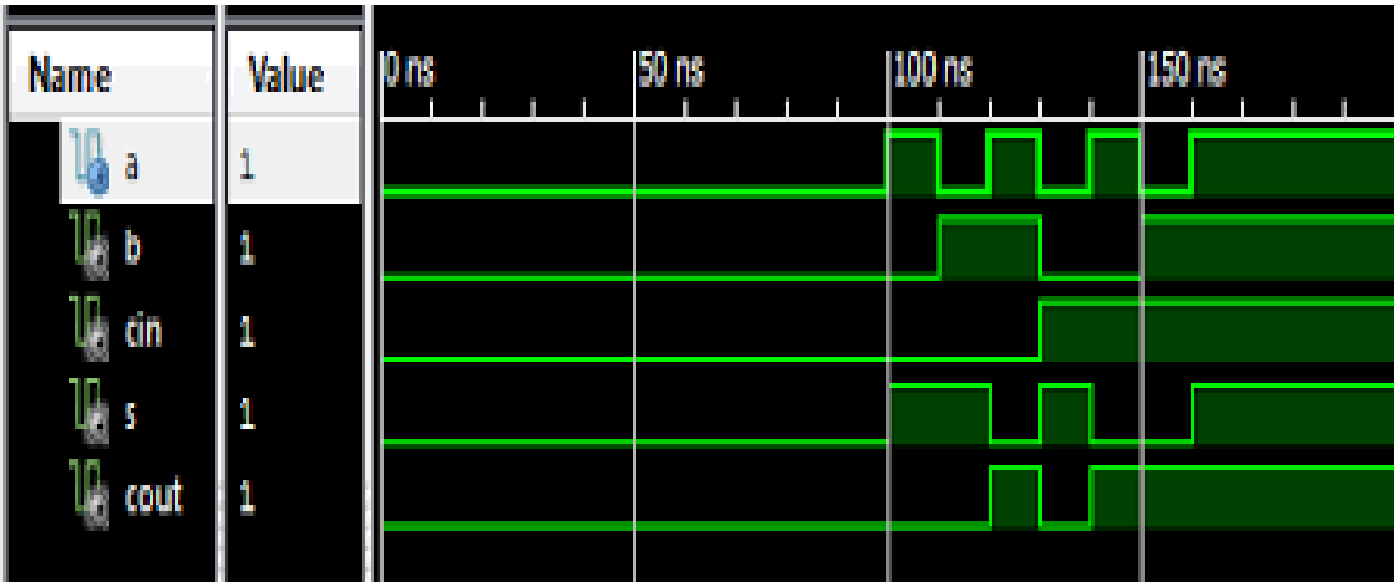


Figure: Test bench waveform for Full adder

Experiment-1b

Truth Table:

INPUTS				OUTPUTS			
B[3]	**B[2]**	**B[1]**	**B[0]**	**E[3]**	**E[2]**	**E[1]**	**E[0]**
0	0	0	0	0	0	**1**	**1**
0	0	0	**1**	0	**1**	0	0
0	0	**1**	0	0	**1**	0	**1**
0	0	**1**	**1**	0	**1**	**1**	0
0	**1**	0	0	0	**1**	**1**	**1**
0	**1**	0	**1**	**1**	0	0	0
0	**1**	**1**	0	**1**	0	0	**1**
0	**1**	**1**	**1**	**1**	0	**1**	0
1	0	0	0	**1**	0	**1**	**1**
1	0	0	1	**1**	**1**	0	0
1	0	**1**	0	X	X	X	X
1	0	**1**	**1**	X	X	X	X
1	**1**	0	0	X	X	X	X
1	**1**	0	**1**	X	X	X	X
1	**1**	**1**	0	X	X	X	X
1	**1**	**1**	**1**	X	X	X	X

Experiment 1b: BCD to Excess-3 Converter

BCD to Excess-3 Converter Module:

```
module BCD_to_Excess3 (
   input [3:0] BCD,      // 4-bit BCD input
   output  reg [3:0] Excess3  // 4-bit Excess-3 code output
);
   always @(BCD)
         begin
         if(BCD<=4'b1001)
         Excess3 = BCD + 4'b0011;
else
Excess3=4'bxxxx;
end
         // Add 3 (Excess-3) to the BCD input
endmodule
```

User Constraint File (UCF):

```
NET "BCD[0]"  LOC = "p21"  ;
NET "BCD[1]"  LOC = "p27"  ;
NET "BCD[2]"  LOC = "p29"  ;
NET "BCD[3]"  LOC = "p35"  ;
NET "Excess3[0]"  LOC = "p20"  ;
NET "Excess3[1]"  LOC = "p26"  ;
NET "Excess3[2]"  LOC = "p28"  ;
NET "Excess3[3]"  LOC = "p34"  ;
```

Outcome: Verified and synthesized the structural modeling of Full adder and BCD Excess-3 Code Converter using FPGA

Exepriment-2: **Write a Verilog description for the following Sequential Circuits, Verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.**

A. **Mod-N counter**
B. **Random sequence counter**

Objective: To verify and synthesize the structural modeling of Mod-N Counter and Random Sequence Counter using FPGA.

2A. Mod-N Counter

```
module mod_n_counter (
   input wire clk,          // System clock
   input wire reset,        // Asynchronous reset
   output reg [3:0] count   // 4-bit counter output (can be adjusted for higher Mod-N)
);
   // Parameter for the modulus (N)
   parameter N = 16; // Change this for different Mod-N (e.g., 10 for Mod-10)

   // Clock divider parameters
   parameter DIVISOR = 5000000; // Assuming 5 MHz clock
   reg [25:0] clk_div; // Divider register (adjust size based on the clock frequency)

   // Clock enable
   reg clk_1Hz_en;

   // Clock divider to create a 1 Hz clock enable
   always @(posedge clk or posedge reset) begin
      if (reset)
begin
         clk_div <= 0;
         clk_1Hz_en <= 0;
      end
else begin
         if (clk_div == (DIVISOR - 1)) begin
            clk_div <= 0;
            clk_1Hz_en <= 1; // Generate clock enable signal
         end else begin
            clk_div <= clk_div + 1;
            clk_1Hz_en <= 0; // Hold clock enable low
         end
      end
   end
```

```
 // Mod-N counter
     always @(posedge clk or posedge reset) begin
      if (reset) begin
         count <= 0;
      end else if (clk_1Hz_en) begin
         if (count == (N - 1)) begin
            count <= 0; // Reset to 0 when reaching N
         end else begin
            count <= count + 1; // Increment counter
         end
      end
   end
endmodule
```

User Constraint File (UCF):

```
NET "clk" LOC="p79”
NET "reset" LOC = "p27"
NET "Count [3]" LOC = "p20"
NET "Count [2]" LOC = "p26"
NET "Count [1]" LOC = "p28"
NET "Count [0]" LOC = "p34"
```

2B. Random Sequence Counter

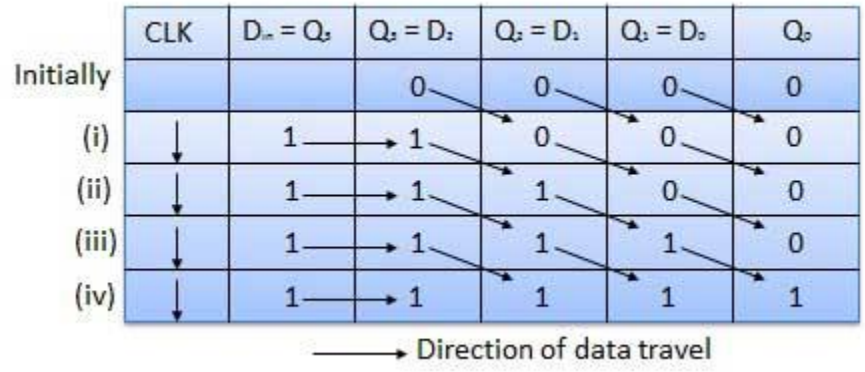
```
2. module random_sequence_generator (
   input wire clk,          // System clock (e.g., 50 MHz)
   input wire reset,        // Asynchronous reset
   output reg [3:0] rand_out // 4-bit random output
);
   // Clock divider parameters
   parameter DIVISOR =10000000; // Assuming a 10 MHz clock
   reg [25:0] clk_div; // Divider register
   reg clk_1Hz; // 1 Hz clock signal

   // Simple Linear Feedback Shift Register (LFSR) for random sequence
```

```
 reg [3:0] lfsr; // 4-bit LFSR
wire lfsr_feedback;

// LFSR feedback logic (example taps)
assign lfsr_feedback = lfsr[3] ^ lfsr[2]; // Feedback from bits 3 and 2

// Clock divider to create a 1 Hz clock
always @(posedge clk or posedge reset) begin
    if (reset) begin
    clk_div <= 0;
        clk_1Hz <= 0;
    end else begin
        if (clk_div == (DIVISOR - 1)) begin
            clk_div <= 0;
            clk_1Hz <= ~clk_1Hz; // Toggle the 1Hz clock
        end else begin
            clk_div <= clk_div + 1;
        end
    end
end

// LFSR for random number generation
always @(posedge clk_1Hz or posedge reset) begin
    if (reset) begin
        lfsr <= 4'b1010; // Seed value for LFSR
    end else begin
        lfsr <= {lfsr[2:0], lfsr_feedback}; // Shift left and insert feedback
    end
end

// Assign the output
always @(posedge clk_1Hz or posedge reset) begin
    if (reset) begin
        rand_out <= 4'b0000; // Reset output
    end else begin
        rand_out <= lfsr; // Output the current LFSR value
    end
end

endmodule
```

Outcome: Verified and synthesized the structural modeling of Mod-N Counter and Random Sequence Counter using FPGA.

Experiment-3:

Truthtable for SISO:



Truth table for PISO:

Clk	Qa	Qb	Qc	Qd (Out)
0	0	0	0	0
1	1	1	0	1
2	0	1	1	0
3	0	0	1	1
4	0	0	0	1

Experiment-3: **Write a Verilog description for the following Sequential Circuits, Verify the design using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.**

a. SISO and PISO shift register
b. Ring counter

Objective: To verify and synthesize SISO, PISO and Ring Counter using FPGA Counter.

3A. SISO (Serial in Serial Out) Shift Register

```
module SISo (
input clk,            // Input clock
input rst_n,          // Active-low reset
input serial_in,      // Serial input data
output reg serial_out, // Serial output data
output reg [3:0] shift_reg  // Internal 8-bit shift register
);
// Clock divider to reduce the clock to 1 Hz
reg [24:0] clk_div_counter; // Assuming 50 MHz input clock, 50 million counts for 1 Hz

always @(posedge clk or negedge rst_n) begin
    if (~rst_n) begin
        clk_div_counter <= 25'd0;  // Reset the counter
    end else if (clk_div_counter == 25'd49999999) begin
        clk_div_counter <= 25'd0; // Reset the counter when it reaches 50 million
    end else begin
        clk_div_counter <= clk_div_counter + 1;
    end
end

wire clk_1Hz = (clk_div_counter == 25'd49999999); // Generate 1 Hz clock signal

// Shift register logic
always @(posedge clk_1Hz or negedge rst_n) begin
    if (~rst_n) begin
        shift_reg <= 4'b0;    // Reset the shift register
        serial_out <= 0;      // Reset the serial output
    end else begin
        shift_reg <= {shift_reg[2:0], serial_in}; // Shift left and insert serial input at the LSB
        serial_out <= shift_reg[3];               // Output the MSB of the shift register
    end
```

```
    end
endmodule
```

User Constraint File (UCF):

```
NET "clk"  LOC = "p79"  ;
NET "rst_n"  LOC = "p21"  ;
NET "serial_in"  LOC = "p48"  ;
NET "serial_out"  LOC = "p20"  ;
NET "shift_reg[0]"  LOC = "p46"  ;
NET "shift_reg[1]"  LOC = "p51"  ;
NET "shift_reg[2]"  LOC = "p57"  ;
NET "shift_reg[3]"  LOC = "p61"  ;
```

3A. PISO (Parallel in Serial Out) Shift Register

```
 module PISO_1Hz (
    input wire clk,            // System clock (e.g., 50 MHz)
    input wire reset,          // Reset signal
    input wire [3:0] parallel_in, // 8-bit parallel input
    output reg serial_out      // Serial output
);

    // 50 MHz clock divider for 1 Hz
    reg [25:0] counter;  // 26-bit counter to count up to 50 million

    // 8-bit shift register
    reg [3:0] shift_reg;

    // Reset or shift logic
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            counter <= 26'd0;  // Reset counter
            shift_reg <= parallel_in;  // Load parallel input to shift register
            serial_out <= 1'b0;  // Initial serial output
        end else begin
            if (counter == 26'd5000000 - 1) begin
                counter <= 26'd0;  // Reset counter when it reaches 50 million
                serial_out <= shift_reg[3];  // Output the most significant bit
                shift_reg <= shift_reg << 1;  // Shift left by 1
            end else begin
                counter <= counter + 1;  // Increment counter
            end
        end
```

```
    end

endmodule
```

User Constraint File (UCF):

```
NET "clk"  LOC = "p76"  ;
NET "parallel_in[0]"  LOC = "p21"  ;
NET "parallel_in[1]"  LOC = "p27"  ;
NET "parallel_in[2]"  LOC = "p29"  ;
NET "parallel_in[3]"  LOC = "p35"  ;
NET "reset"  LOC = "p48"  ;
NET "serial_out"  LOC = "p20"  ;
```

Truth table for Ring Counter:

Clock Input	Q_3	Q_2	Q_1	Q_0
1	**0**	**0**	**0**	**1**
2	**0**	**0**	**1**	**0**
3	**0**	**1**	**0**	**0**
4	**1**	**0**	**0**	**0**
5	**0**	**0**	**0**	**1**
6	**0**	**0**	**1**	**0**
7	**0**	**1**	**0**	**0**
8	**1**	**0**	**0**	**0**

3B. Ring Counter

```
module ring_counter (
   input clk,          // System clock
   input rst_n,        // Active-low reset
   output reg [3:0] q  // 4-bit ring counter output
);

   // Clock divider to generate 1Hz clock from high-frequency input clock
   reg [24:0] clk_div_counter; // Assuming 50 MHz input clock

   always @(posedge clk or negedge rst_n) begin
      if (~rst_n) begin
         clk_div_counter <= 25'd0;
      end else if (clk_div_counter == 25'd49999999) begin
         clk_div_counter <= 25'd0;
      end else begin
         clk_div_counter <= clk_div_counter + 1;
      end
   end

   wire clk_1Hz = (clk_div_counter == 25'd49999999); // 1Hz clock pulse

   // 4-bit Ring Counter
   always @(posedge clk_1Hz or negedge rst_n) begin
      if (~rst_n)
         q <= 4'b0001; // Reset state (initial value)
      else
         q <= {q[2:0], q[3]}; // Shift left in a ring pattern
   end

endmodule
```

Outcome: Verified and synthesized SISO, PISO and Ring Counter using FPGA Counter.

Experiment-4A

Logic Diagram

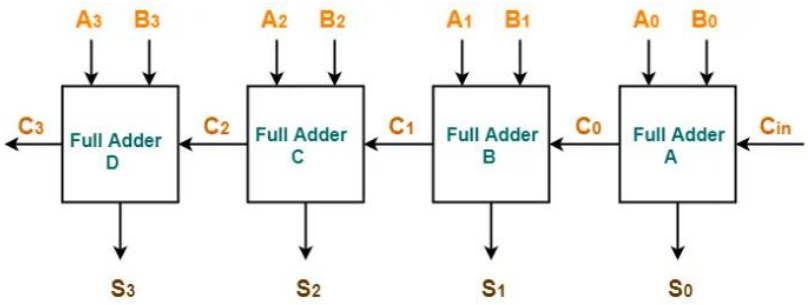


Figure 4A: 4-bit Ripple Carry Adder

Design Object List - I/O Pins

	I/O Name	I/O Direction	Loc	Bank	I/O
	A[0]	Input	p45	BANK6	
	A[1]	Input	p43	BANK6	
	A[2]	Input	p40	BANK6	
	A[3]	Input	p37	BANK6	
	B[0]	Input	p62	BANK5	
	B[1]	Input	p58	BANK5	
	B[2]	Input	p52	BANK6	
	B[3]	Input	p48	BANK6	
	Cin	Input	p21	BANK7	
	Cout	Output	p20	BANK7	
	Sum[0]	Output	p44	BANK6	
	Sum[1]	Output	p42	BANK6	
	Sum[2]	Output	p39	BANK6	
	Sum[3]	Output	p36	BANK6	

UCF Pin Configuration for 4-bit Ripple Carry Adder

Experiment-4: **Write a Verilog description for the following Digital Circuits, Verify the functionality using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.**

A. 4-Bit Ripple Carry Adder
B. 4-Bit Linear Feedback shift register

A. 4-Bit Ripple Carry Adder

A Ripple Carry Adder (RCA) adds two binary numbers bit by bit, where each sum bit depends on the carry from the previous stage.

```
module ripple_carry_adder (
input [3:0] A,        // 4-bit input A
input [3:0] B,        // 4-bit input B
input Cin,            // Carry-in (initial carry)
output [3:0] Sum,     // 4-bit sum
output Cout );        // Carry-out
  wire c1, c2, c3;  // Internal carry signals
// First bit (LSB)
    full_adder FA0 (.A(A[0]), .B(B[0]), .Cin(Cin), .Sum(Sum[0]), .Cout(c1) );
// Second bit
    full_adder FA1 (.A(A[1]), .B(B[1]), .Cin(c1), .Sum(Sum[1]), .Cout(c2));
// Third bit
    full_adder FA2 ( .A(A[2]), .B(B[2]), .Cin(c2), .Sum(Sum[2]), .Cout(c3));
// Fourth bit (MSB)
    full_adder FA3 ( .A(A[3]), .B(B[3]), .Cin(c3), .Sum(Sum[3]), .Cout(Cout));
    endmodul

// Full Adder module used in the Ripple Carry Adder
  module full_adder (
  input A, B, Cin,     // 1-bit inputs
  output Sum, Cout      // 1-bit outputs (Sum and Carry-out)
);
assign Sum = A ^ B ^ Cin;  // Sum is XOR of inputs
assign Cout = (A & B) | (Cin & (A ^ B));  // Carry-out is generated from ANDs and ORs
endmodule
```

Experiment-4B

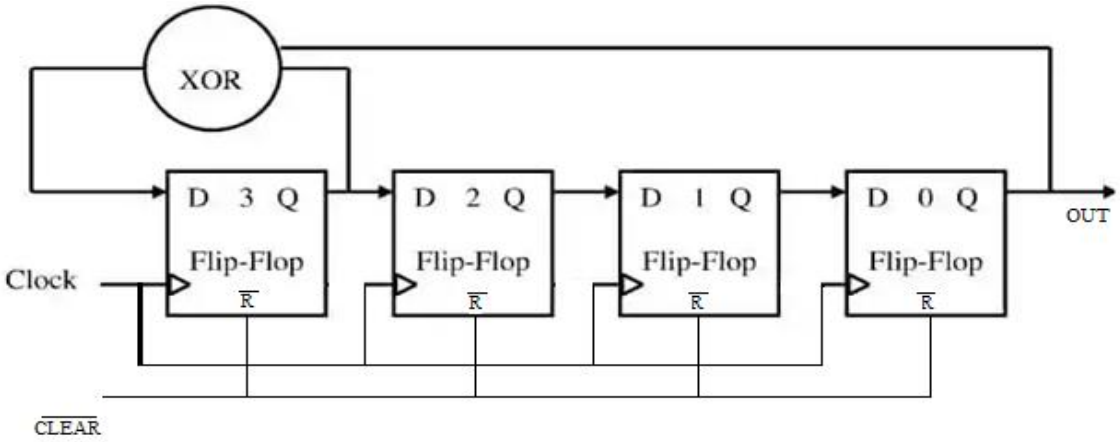


Figure 4B: Logic Diagram for 4 bit LFSR

Outcome: Verified and Synthesized 4-bit Ripple Carry Adder and 4-bit LFSR using FPGA.

Cycle	lfsr[3]	lfsr[2]	lfsr[1]	lfsr[0]	Feedback (XOR) lfsr[0] ^ lfsr[3]	New LFSR Value
0	1	0	0	1	1 XOR 1 = 0	0100
1	0	1	0	0	0 XOR 0 = 0	0010
2	0	0	1	0	0 XOR 0 = 0	0001
3	0	0	0	1	1 XOR 0 = 1	1000
4	1	0	0	0	0 XOR 1 = 1	1100
5	1	1	0	0	0 XOR 1 = 1	1110
6	1	1	1	0	0 XOR 1 = 1	1111
7	1	1	1	1	1 XOR 1 = 0	0111
8	0	1	1	1	1 XOR 0 = 1	1011
9	1	0	1	1	1 XOR 1 = 0	1101
10	0	1	0	1	1 XOR 0 = 1	1010
11	1	0	1	0	0 XOR 1 = 1	0101
12	1	0	0	1	1 XOR 1 = 0	1001

4B. 4-Bit Linear Feedback Shift Register (LFSR)

```
module LFSR_with_clock_divider(
    input wire clk,       // 10 MHz system clock
    input wire rst,       // Asynchronous reset
    output reg [3:0] lfsr, // 4-bit LFSR output
    output reg clk_out    // 1 Hz output clock
);
// Clock Divider Logic (for 1 Hz from 10 MHz)
reg [23:0] counter; // 24-bit counter for clock division
always @(posedge clk or posedge rst) begin
    if (rst) begin
        counter <= 24'b0;
        clk_out <= 0;
    end else begin
        if (counter == 24'd9999999) begin // Counter value for 1 Hz clock (10 MHz / 10^7 = 1 Hz)
            counter <= 24'b0;
            clk_out <= ~clk_out; // Toggle output clock
        end else begin
            counter <= counter + 1;
        end
    end
end
// LFSR Logic
always @(posedge clk_out or posedge rst) begin
    if (rst) begin
        lfsr <= 4'b1001; // Initial seed for LFSR (can be any non-zero value)
    end else begin
        // Shift left and feedback XOR the 4th and 3rd bits
        lfsr[3] <= (lfsr[3] ^ lfsr[0]); // Feedback from bit 3 and 0
        lfsr[2] <= lfsr[3];
        lfsr[1] <= lfsr[2];
        lfsr[0] <= lfsr[1];
    end
end
```

endmodule

Experiment-5A

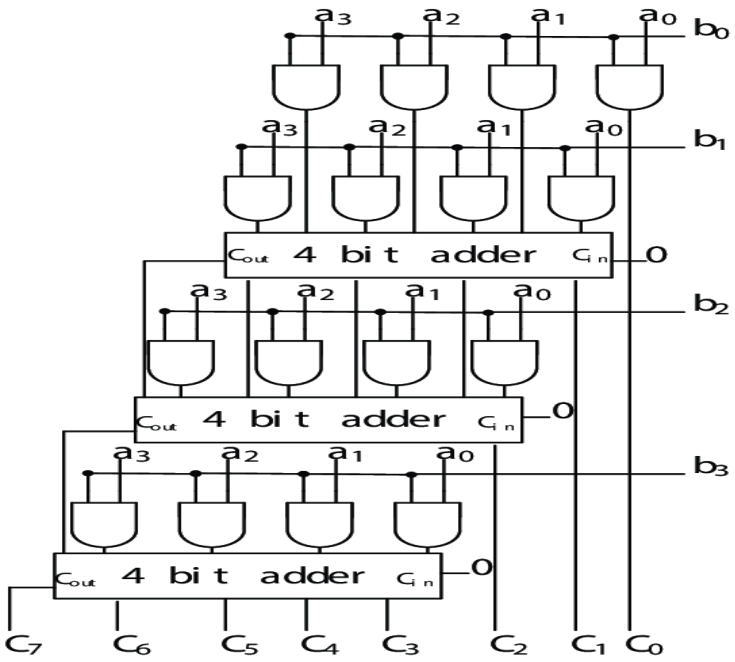


Figure 5A: Logic Diagram of 4-bit Array mutliplier

Experiment-5: **Write a Verilog description for the following Digital Circuits, Verify the functionality using Verilog test bench and perform the synthesis by downloading the design on to FPGA device.**

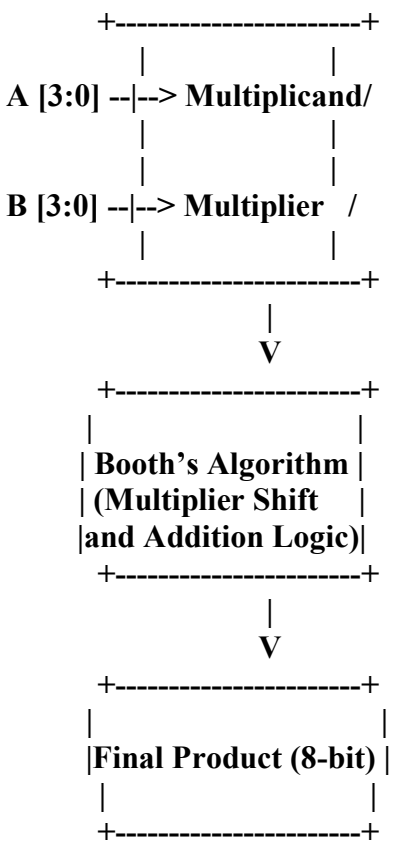
A. 4-bit Array Multiplication
B. 4-bit Booth Multiplication

Objective: To verify and Synthesize 4-bit Array multiplier and Booth Multiplier using FPGA.

5A. 4-bit Array Multiplier

```
module Array_Multiplier(
   input [3:0] A,        // 4-bit input A
   input [3:0] B,        // 4-bit input B
   output [7:0] Product   // 8-bit output product
);
// Internal wires to hold partial products
wire [3:0] p0, p1, p2, p3;

// Generate partial products
assign p0 = A[0] ? B : 4'b0000;
assign p1 = A[1] ? B : 4'b0000;
assign p2 = A[2] ? B : 4'b0000;
assign p3 = A[3] ? B : 4'b0000;

// Sum partial products
assign Product = p0 + (p1 << 1) + (p2 << 2) + (p3 << 3);
endmodule
```

4-bit Booth Multiplier Block Diagram:



5B. 4-bit Booth Multiplier:

```
module booth_multiplier (
    input [3:0] A,         // 4-bit input A
    input [3:0] B,         // 4-bit input B
    output reg [7:0] Product  // 8-bit output product
);
reg [7:0] M, Q, Qn1; // Registers for multiplicand, multiplier, and Qn1
integer i;
always @(A or B) begin
    // Initialize the registers
    M = {4'b0000, A};  // Extend A with zeroes (4 bits)
    Q = {4'b0000, B};  // Extend B with zeroes (4 bits)
    Qn1 = 1'b0;        // Initialize Qn1 (the previous Q0 bit)
    Product = 8'b0;    // Clear the product register

  for (i=0; i<4; i=i + 1) begin
        case ({Q[0],Qn1})
            2'b01:Product = Product + M;  // Add M to the product
            2'b10:Product = Product - M;  // Subtract M from the product
            endcase

        // Arithmetic shift right (ASR) to prepare for the next iteration
        Qn1 = Q[0];
        Q = {Product[0], Q[7:1]};  // Shift the Q register
        Product = {Product[7], Product[7:1]};  // Shift the Product register
    end
end
endmodule
```

Outcome: Verified and Synthesized 4-bit Array multiplier and Booth Multiplier using FPGA.

Experiment-6: **Write a Verilog description to design a clock divider circuit that generates $^1/_2$, $^1/_3{}^{rd}$ and $^1/_4{}^{th}$ clock from a given input clock. Port the design to FPGA and validate the functionality using output device.**

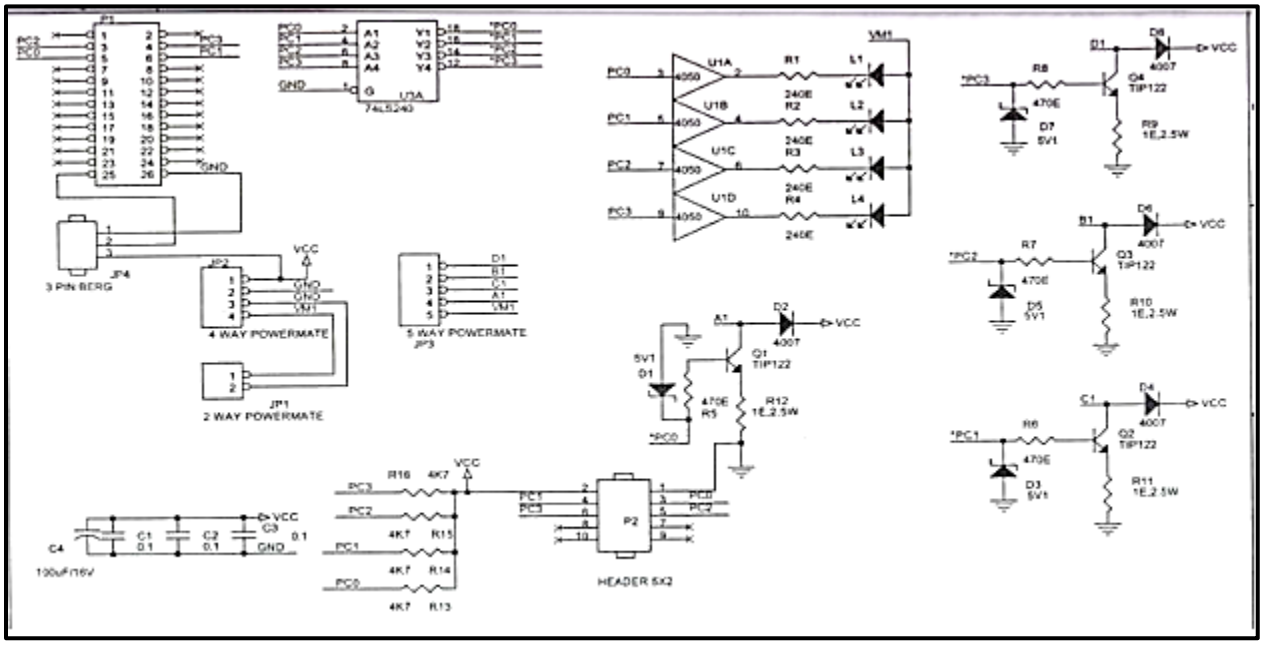
Verilog Code for Clock Divider Circuit:

```
module ClockDivider (
   input wire clk_in,    // Input clock signal
   input wire reset,     // Reset signal
   output reg clk_out_2, // Output clock signal divided by 2
   output reg clk_out_3, // Output clock signal divided by 3
   output reg clk_out_4  // Output clock signal divided by 4
);

   // Internal counters for dividing the clock
   reg [1:0] counter_2;  // 2-bit counter for division by 2
   reg [1:0] counter_3;  // 2-bit counter for division by 3
   reg [2:0] counter_4;  // 3-bit counter for division by 4

   // Process to divide the input clock for 1/2, 1/3, and 1/4 clocks
   always @(posedge clk_in or posedge reset) begin
      if (reset) begin
         // Reset the counters and clock outputs
         counter_2 <= 2'b00;
         counter_3 <= 2'b00;
         counter_4 <= 3'b000;
         clk_out_2 <= 0;
         clk_out_3 <= 0;
         clk_out_4 <= 0;
      end else begin
         // Dividing by 2
         counter_2 <= counter_2 + 1;
         if (counter_2 == 2'b01) begin
            clk_out_2 <= ~clk_out_2;
            counter_2 <= 2'b00;
         end

         // Dividing by 3
         counter_3 <= counter_3 + 1;
         if (counter_3 == 2'b10) begin
            clk_out_3 <= ~clk_out_3;
            counter_3 <= 2'b00;
         end
```

```
        // Dividing by 4
        counter_4 <= counter_4 + 1;
        if (counter_4 == 3'b011) begin
            clk_out_4 <= ~clk_out_4;
            counter_4 <= 3'b000;
        end
    end
end

endmodule
```

Outcome: Verified clock division using FPGA.

Experiment-7:



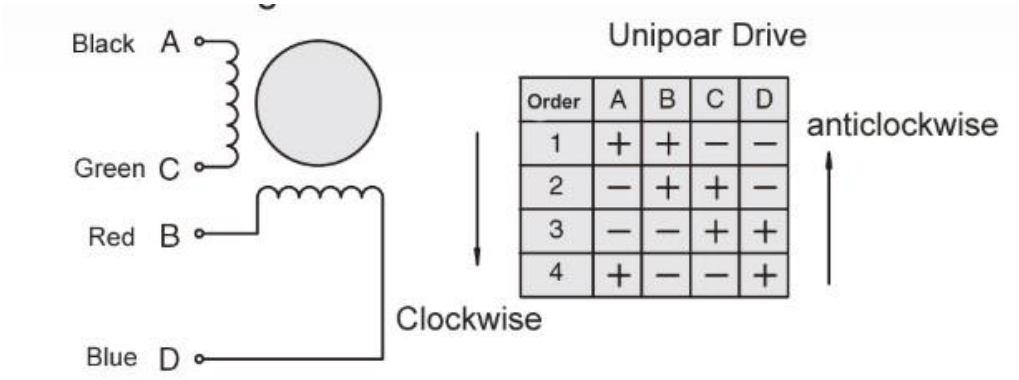


Figure: **Interfacing of a Stepper Motor**

Experiment-7: Interface a Stepper motor to FPGA and Write a Verilog description to control Stepper motor rotation.

Objective: To interface a Stepper motor to FPGA and control motor rotation

```
module stepper_motor_controller (
    input clk,              // Input clock signal (e.g., 50 MHz)
    input reset,            // Reset signal
    input direction,        // Direction control: 0 = clockwise, 1 = counterclockwise
    input [31:0] speed,     // Speed control: Delay count for each step
    output reg [3:0] motor_pins  // 4-bit output to control the stepper motor
);
reg [31:0] counter;          // Counter for speed control
reg [3:0] step_sequence [0:7];  // Step sequence for 4-phase control
reg [2:0] step_index;        // Index to keep track of the current step

// Step sequence for full step (could be modified for half-step)
initial begin
    step_sequence[0] = 4b0001;  // Step 1: Coil 1 active
    step_sequence[1] = 4b0010;  // Step 2: Coil 2 active
    step_sequence[2] = 4b0100;  // Step 3: Coil 3 active
    step_sequence[3] = 4b1000;  // Step 4: Coil 4 active
    step_sequence[4] = 4b1000;  // Step 5: Coil 4 active
    step_sequence[5] = 4b0100;  // Step 6: Coil 3 active
    step_sequence[6] = 4b0010;  // Step 7: Coil 2 active
    step_sequence[7] = 4b0001;  // Step 8: Coil 1 active
end

// Main clock logic for stepping
always @(posedge clk or posedge reset) begin
    if (reset) begin
        counter <= 0;
        step_index <= 0;
        motor_pins <= 4b0001; // Start with step 1 (coil 1 active)
```

```
    end else if (counter < speed) begin
        counter <= counter + 1;
    end
else
 begin
        // Time to move to the next step
        counter <= 0;

        // Update step_index for direction
        if (direction == 0) begin  // Clockwise: Increase step_index
            step_index <= step_index + 1;
            if (step_index == 7) step_index <= 0;  // Wrap around
        end else begin  // Counterclockwise: Decrease step_index
            if (step_index == 0) step_index <= 7;
            else step_index <= step_index - 1;
        end
        // Set the motor control pins according to the current step
        motor_pins <= step_sequence[step_index];
    end
end
endmodule
```

Experiment-8: **Interface a DAC to FPGA and Write a Verilog description to generate Square wave of frequency F KHz. Modify the code to down sample the frequency to F/2 KHz. Display the original and down sampled signals by connecting them to an output device.**

Objective: To interface a DAC (Digital-to-Analog Converter) to an FPGA and generate a square wave signal of frequency F kHz.

```
module SquareWaveGenerator (
    input wire clk_in,    // Input clock signal (e.g., 100 MHz)
    input wire reset,     // Reset signal
    output reg dac_out_1, // Original square wave (F kHz)
    output reg dac_out_2  // Downsampled square wave (F/2 kHz)
);

    // Counter for original square wave (F kHz)
    reg [15:0] counter_1;
    reg [15:0] counter_2;  // Counter for downsampled square wave (F/2 kHz)
    reg clk_div_1;  // Clock for original square wave
    reg clk_div_2;  // Clock for downsampled square wave

    // Parameters to configure the frequency of the square wave (F kHz and F/2 kHz)
    // Assuming input clock is 100 MHz
    parameter CLK_FREQ = 100_000_000;  // 100 MHz clock input
    parameter F_kHz = 10_000;          // Target frequency F = 10 kHz
    parameter F2_kHz = F_kHz / 2;      // Downsampled frequency F/2

    // Calculate clock division factors for F kHz and F/2 kHz
    parameter DIV_FACTOR_1 = CLK_FREQ / (2 * F_kHz);  // Divide by 2 for square wave frequency
    parameter DIV_FACTOR_2 = CLK_FREQ / (2 * F2_kHz); // Divide by 2 for downsampled frequency

    always @(posedge clk_in or posedge reset) begin
        if (reset) begin
            // Reset counters and outputs
            counter_1 <= 0;
```

```
            counter_2 <= 0;
            dac_out_1 <= 0;
            dac_out_2 <= 0;
        end else begin
            // Generate original square wave with frequency F kHz
            if (counter_1 >= DIV_FACTOR_1 - 1) begin
                counter_1 <= 0;
                dac_out_1 <= ~dac_out_1;  // Toggle square wave
            end else begin
                counter_1 <= counter_1 + 1;
            end

            // Generate downsampled square wave with frequency F/2 kHz
            if (counter_2 >= DIV_FACTOR_2 - 1) begin
                counter_2 <= 0;
                dac_out_2 <= ~dac_out_2;  // Toggle downsampled square wave
            end else begin
                counter_2 <= counter_2 + 1;
            end
        end
    end

endmodule
```

Outcome: Learnt successful interfacing of the DAC to FPGA.

Experiment-9: **Write a Verilog description to convert an analog input of a sensor to digital form and to display the same on a suitable display like set of simple LEDs like 7-Segment display digits.**

Objective: To interface an ADC to FPGA to convert an analog input of a sensor to digital form and to display the same on a suitable display

Module 1: ADC Interface

```
module adc_simulator (
   input clk,
   input reset,
   output reg [9:0] adc_data // 10-bit ADC data (simulated)
);

   // Simulate an ADC value by incrementing it every clock cycle
   always @(posedge clk or posedge reset) begin
      if (reset)
         adc_data <= 10'b0;  // Reset ADC data
      else
         adc_data <= adc_data + 1;  // Simulate increasing ADC value
   end
endmodule
```

Module 2: 7-Segment Display Decoder

```
module seven_segment_decoder (
   input [3:0] digit,        // 4-bit input digit (0-15)
   output reg [6:0] seg      // 7-segment display output (a-g)
);
      always @(digit) begin
      case (digit)
         4'd0: seg = 7'b1111110; // 0
         4'd1: seg = 7'b0110000; // 1
         4'd2: seg = 7'b1101101; // 2
         4'd3: seg = 7'b1111001; // 3
         4'd4: seg = 7'b0110011; // 4
         4'd5: seg = 7'b1011011; // 5
         4'd6: seg = 7'b1011111; // 6
```

```
            4'd7: seg = 7'b1110000; // 7
            4'd8: seg = 7'b1111111; // 8
            4'd9: seg = 7'b1111011; // 9
            4'd10: seg = 7'b1110111; // A
            4'd11: seg = 7'b0011111; // b
            4'd12: seg = 7'b1001110; // C
            4'd13: seg = 7'b0111101; // d
            4'd14: seg = 7'b1001111; // E
            4'd15: seg = 7'b1000111; // F
            default: seg = 7'b0000000; // Default: Blank
        endcase
    end
endmodule
```

Module 3:

The top-level module interfaces the ADC simulator, the 7-segment display decoder, and handles displaying the ADC value on a 7-segment display.

```
module adc_to_7segment_display (
    input clk,              // Clock input
    input reset,            // Reset signal
    output [6:0] seg1,      // 7-segment display for digit 1 (MSB)
    output [6:0] seg2,      // 7-segment display for digit 2
    output [6:0] seg3,      // 7-segment display for digit 3
    output [6:0] seg4       // 7-segment display for digit 4
);
    wire [9:0] adc_data;    // 10-bit ADC data

    // Instantiate the ADC simulator

    adc_simulator adc_inst (
        .clk(clk),
        .reset(reset),
        .adc_data(adc_data)
    );

    wire [3:0] digit1, digit2, digit3, digit4;

    // Extract digits from the 10-bit ADC data (divide it into 4 digits)

    assign digit1 = adc_data[9:8];   // Most significant digit (MSD)
    assign digit2 = adc_data[7:4];   // 2nd digit
    assign digit3 = adc_data[3:2];   // 3rd digit
    assign digit4 = adc_data[1:0];   // Least significant digit (LSD)

    // Instantiate 7-segment display decoders for each digit
```

```
    seven_segment_decoder seg1_decoder (
        .digit(digit1),
        .seg(seg1)
    );
    seven_segment_decoder seg2_decoder (
        .digit(digit2),
        .seg(seg2)
    );
    seven_segment_decoder seg3_decoder (
        .digit(digit3),
        .seg(seg3)
    );
    seven_segment_decoder seg4_decoder (
        .digit(digit4),
        .seg(seg4)
    );
endmodule
```

Outcome: Learnt successful interfacing of ADC with FPGA.