||Jai Sri Gurudev||

Sri Adichunchanagiri Shikshana Trust®

BGS INSTITUTE OF TECHNOLOGY

(Affiliated to VTU Belagavi, Approved by AICTE, New Delhi)

BG Nagara – 571448 (Bellur Cross)

Nagamangala Taluk, Mandya District.

HDL LABORATORY MANUAL 17ECL58

For

VII Semester B.E. E&CE

2019-2020

DEPARTMENT OF

ELECTRONICS AND COMMUNICATION ENGINEERING

Prepared by:

1. Mr. Manojkumar S B, Asst. Prof
2. Ms. Srividya C N, Asst. Prof
3. Mrs. Rashmi S, Asst. Prof

Approved by:

Dr. M.B. Anandaraju

Head, Dept. ECE.

DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

Vision:

To develop high quality engineers with technical knowledge, skills and ethics in the area of Electronics and Communication Engineering to meet industrial and societal needs.

Mission:

1. To provide high quality technical education with up-to-date infrastructure and trained human resources to deliver the curriculum effectively in order to impart technical knowledge and skills.
2. To train the students with entrepreneurship qualities, multidisciplinary knowledge and latest skill sets as required for industry, competitive examinations, higher studies and research activities.
3. To mould the students into professionally-ethical and socially-responsible engineers of high character, team spirit and leadership qualities.

Program Educational Objectives (PEO'S):

After 3 to 5 years of graduation, the graduates of Electronics and Communication Engineering will –

1. Engage in industrial, teaching or any technical profession and pursue higher studies and research.
2. Apply the knowledge of Mathematics, Science as well as Electronics and Communication Engineering to solve social engineering problems.
3. Understand, Analyze, Design and Create novel products and solutions.
4. Display professional and leadership qualities, communication skills, team spirit, multidisciplinary traits and lifelong learning aptitude.

Program Specific Outcomes (PSO'S):

1. To apply the knowledge of Electronics and Communication Engineering as well as automation tools to create electronic circuits, systems and solutions.
2. To collaborate effectively with Electronics and Information Technology industries through internship, induction, technical seminar, technical project, research, product design and development, industrial visit, staff training in order to provide the actual industrial exposure to students and faculties.

COURSE OBJECTIVES

This laboratory course enables students to get practical experience in design, assembly, testing and evaluation of:

- Familiarize with the CAD tool to write HDL programs.
- Understand simulation and synthesis of digital design.
- Program FPGAs/CPLDs to synthesize the digital designs.
- Interface hardware to programmable ICs through I/O ports.
- Choose either Verilog or VHDL for a given Abstraction level.

COURSE OUTCOMES

Students will be able to:

- Write the Verilog/VHDL programs to simulate Combinational circuits in Dataflow, Behavioral and Gate level Abstractions.
- Describe sequential circuits like flip flops and counters in Behavioral description and obtain simulation waveforms.
- Synthesize Combinational and Sequential circuits on programmable ICs and test the hardware.
- Interface the hardware to the programmable chips and obtain the required output.

PROGRAMME SPECIFIC OUTCOMES

Graduates will be able to:

- Exhibit competency in embedded system domain.
- Exhibit competency in RF & Signal processing domain.

PROGRAMME EDUCATION OBJECTIVES

Graduates will be able to:

- Work as professionals in the area of Electronics and Allied Engineering fields.
- Pursue higher studies and involve in interdisciplinary research work.
- Exhibit ethics, professional skills and leadership qualities in their profession.

PROGRAMME OUTCOMES *Our graduates will be able to*	
1.	Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2.	Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3.	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.
4.	Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide validconclusions.
5.	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.
6.	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineeringpractice.
7.	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8.	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9.	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10.	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.
11.	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.
12.	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.

CONTENTS:

HDL Lab
B.E., V Semester, EC/TC
[As per Choice Based Credit System (CBCS) scheme]

Subject Code	17ECL58	CIE Marks	40
Number of Lecture Hours/Week	01Hr Tutorial (Instructions) + 02 Hours Laboratory=03	SEE Marks	60
RBT Levels L1, L2, L3	RBT Levels L1, L2, L3	Exam Hours 03	03

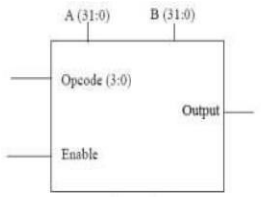
CREDITS – 02

Course objectives: This course will enable students to:
· Familiarize with the CAD tool to write HDL programs.
· Understand simulation and synthesis of digital design.
· Program FPGAs/CPLDs to synthesize the digital designs.
· Interface hardware to programmable ICs through I/O ports.
· Choose either Verilog or VHDL for a given Abstraction level.

Note: Programming can be done using any compiler. Download the programs on a FPGA/CPLD boards such as Apex/Acex/Max/Spartan/Sinfi or equivalent and performance testing may be done using 32 channel pattern generator and logic analyzer apart from verification by simulation with tools such as Altera/Modelsim or equivalent.

Laboratory Experiments

Part–A: PROGRAMMING
1. Write Verilog code to realize all the logic gates
2. Write a Verilog program for the following combinational designs
   a. 2 to 4 decoder
   b. 8 to 3 (encoder without priority & with priority)
   c. 8 to 1 multiplexer.
   d. 4 bit binary to gray converter
   e. Multiplexer, de-multiplexer, comparator.
3. Write a VHDL and Verilog code to describe the functions of a Full Adder using three modeling styles.
4. Write a Verilog code to model 32 bit ALU using the schematic diagram shown below



- ALU should use combinational logic to calculate an output based on the four-bit op-code input.
- ALU should pass the result to the out bus when enable line in high, and tristate the out bus when the enable line is low.
- ALU should decode the 4-bit op-code according to the example given below.

OPCODE	ALU OPERATION
1	A+B
2	A-B
3	A Complement
4	A*B
5	A AND B
6	A OR B
7	A NAND B
8	A XOR B

5. Develop the Verilog code for the following flip-flops, SR, D, JK and T.
. Design a 4-bit binary, BCD counters (Synchronous reset and Asynchronous reset) and "any sequence" counters, using Verilog code.

Part–B: INTERFACING (at least four of the following must be covered using VHDL/Verilog)

1. Write HDL code to display messages on an alpha numeric LCD display.
2. Write HDL code to interface Hex key pad and display the key code on seven segment display.
3. Write HDL code to control speed, direction of DC and Stepper motor.
4. Write HDL code to accept Analog signal, Temperature sensor and display the data on LCD or Seven segment display.
5. Write HDL code to generate different waveforms (Sine, Square, Triangle, Ramp etc.,) using DAC - change the frequency.
6. Write HDL code to simulate Elevator operation.

Course Outcomes: At the end of this course, students should be able to:

· Write the Verilog/VHDL programs to simulate Combinational circuits in Dataflow, Behavioral and Gate level Abstractions.
· Describe sequential circuits like flip flops and counters in Behavioral description and obtain simulation waveforms.
· Synthesize Combinational and Sequential circuits on programmable ICs and test the hardware.
· Interface the hardware to the programmable chips and obtain the required output.

Conduct of Practical Examination:

1. All laboratory experiments are to be included for practical examination.
2. Strictly follow the instructions as printed on the cover page of answer script for breakup of marks.
3. Change of experiment is allowed only once and Marks allotted to the procedure part to be made zero.

OVERVIEW OF HDL LAB

1. HDL

In electronics, a hardware description language or HDL is any language from a class of Computer languages for formal description of electronic circuits. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation

HDLs are standard text-based expressions of the spatial, temporal structure and behavior of electronic systems. In contrast to a software programming language, HDL syntax, semantics include explicit notations for expressing time and concurrency, which are the attributes of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as netlist languages.

HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this execute ability that gives HDLs the illusion of being programming languages. Simulators capable of supporting discrete-event and continuous-time (analog) modeling exist, and HDLs targeted for each are available. It is certainly possible to represent hardware semantics using traditional programming languages such as C++, although to function such programs must be augmented with extensive and unwieldy class libraries. Primarily, however, software programming languages function as a hardware description language

Using the proper subset of virtually any language, a software program called a synthesizer can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives to implement the specified behavior.

This typically requires the synthesizer to ignore the expression of any timing constructs in the text.

The two most widely-used and well-supported HDL varieties used in industry are

- VHDL (VHSIC HDL)
- Verilog

1.1Verilog

Verilog is a hardware description language (HDL) used to model electronic systems. The language supports the design, verification, and implementation of analog, digital, and mixed - signal circuits at various levels of abstraction the designers of Verilog wanted a language with syntax similar to the C programming language so that it would be familiar to engineers and readily accepted. The language is case- sensitive, has a preprocessor like C, and the major control flow keywords, such as "if" and "while", are similar. The formatting mechanism in the printing routines and language operators and their precedence are also similar

The language differs in some fundamental ways. Verilog uses Begin/End instead of curly braces to define a block of code. The concept of time, so important to a HDL won't be found in C The language differs from a conventional programming language in that the execution of statements is not strictly sequential. A Verilog design consists of a hierarchy of modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. Concurrent and sequential statements define the behavior of the module by defining the relationships between the ports, wires, and registers Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But all concurrent statements and all begin/end blocks in the design are executed in parallel, qualifying Verilog as a Dataflow language. A module can also contain one or more instances of another module to define sub-behavior

A subset of statements in the language is synthesizable. If the modules in a design contains a netlist that describes the basic components and connections to be implemented in hardware only synthesizable statements, software can be used to transform or synthesize the design into the net list may then be transformed into, for example, a form describing the standard cells of an integrated circuit (e.g. ASIC) or a bit stream for a programmable logic device (e.g. FPGA).

2. DESIGN USING HDL

The vast majority of modern digital circuit design revolves around an HDL description of the desired circuit, device, or subsystem Most designs begin as a written set of requirements or a high-level architectural diagram. The process of writing the HDL description is highly dependent on the designer's diagram. The process of writing the HDL description is highly dependent on the designer's background and the circuit's nature. The HDL is merely the 'capture language'–often begin with a high-level algorithmic description such as MATLAB or a C++ mathematical model Control and decision structures are often prototyped in flowchart applications, or entered in a state- diagram editor. Designers even use scripting languages (such as PERL) to automatically generate repetitive circuit structures in the HDL language. Advanced text editors (such as PERL) to automatically generate repetitive circuit structures in the HDL language. Advanced text editors (such as Emacs) offer editor templates for automatic indentation, syntax- dependent coloration, and macro-based expansion of entity/architecture/signal declaration.

As the design's implementation is fleshed out, the HDL code invariably must undergo code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers enforce standardized code guidelines, identifying ambiguous code construct before they can cause misinterpretation by downstream synthesis, and check for common logical coding errors, such as dangling ports or shorted outputs.

In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate net list, this net list is passed off to the back - end stage. Depending on the physical technology (FPGA, ASIC gate-array, ASIC standard- cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which cannot be becomes progressively more laden with technology-specific information, which cannot be stored in a generic HDL-description. Finally, a silicon chip is manufactured in a fab.

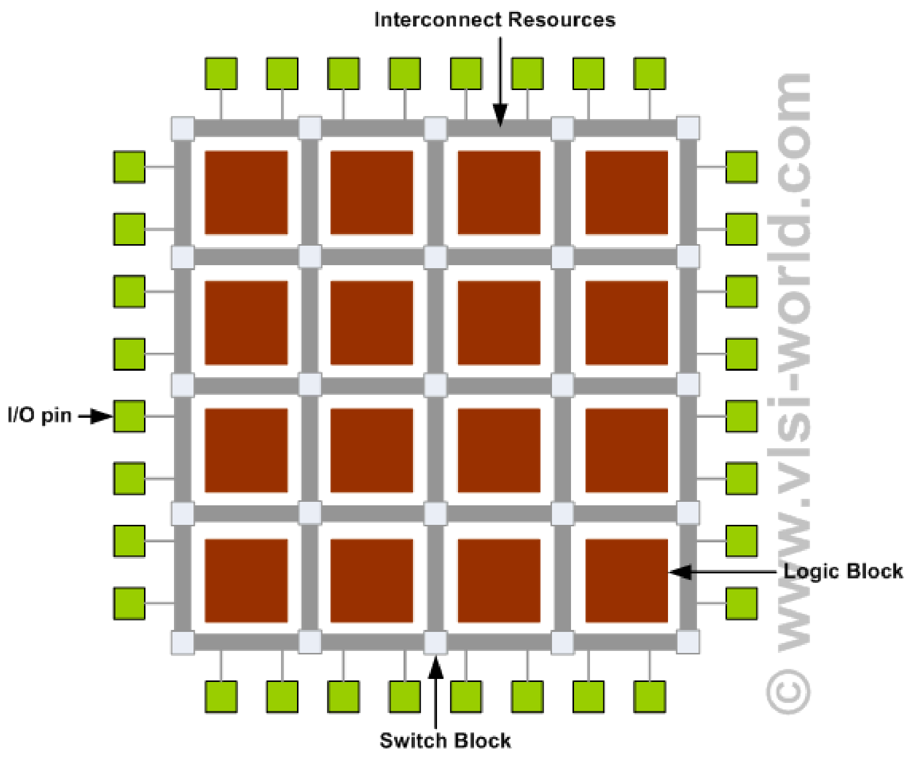
3. SIMULATING AND DEBUGGING HDL CODE

Essential to HDL design is the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. Thus, simulation is critical for successful HDL design To simulate an HDL model, an engineer writes a top-level simulation environment (called a test bench). At minimum, a test bench contains an instantiation of the model (called the device under test or DUT), pin/signal declarations for the model's I/O, and a clock waveform. An HDL simulator–the program that executes the test bench– maintains the simulator clock, which is the master reference for all events in the test bench simulation Events occur only at the instants dictated by the test bench HDL, or in reaction to stimulus and triggering events.

Design verification is often the most time-consuming portion of the design process, due to the disconnect between a device's functional specification, the designer's interpretation of the specification, and the imprecision of the HDL language. The majority of the initial test/debug cycle is conducted in the HDL simulator environment, as the early stage of the design is subject to frequent and major circuit changes. An HDL description can also be prototyped and tested in hardware–programmable logic devices are often used for this purpose. Hardware prototyping is comparatively more expensive than HDL simulation, but offers a real-world view of the design. Prototyping is the best way to check interfacing against other hardware devices, and hardware prototypes, even those running on slow FPGAs, offer much faster simulation times than pure HDL simulation.

INTRODUCTION TO FPGA (FIELD PROGRAMMABLE GATE ARRAY)

FPGA contains a two-dimensional arrays of logic blocks and interconnections between logic blocks. Both the logic blocks and interconnects are programmable. Logic blocks are programmed to implement a desired function and the interconnects are programmed using the switch boxes to connect the logic blocks. To implement a complex design (CPU for instance), the design is divided into small sub functions and each sub function is implemented using one logic block. All the sub functions implemented in logic blocks must be connected and this is done by programming the interconnects.



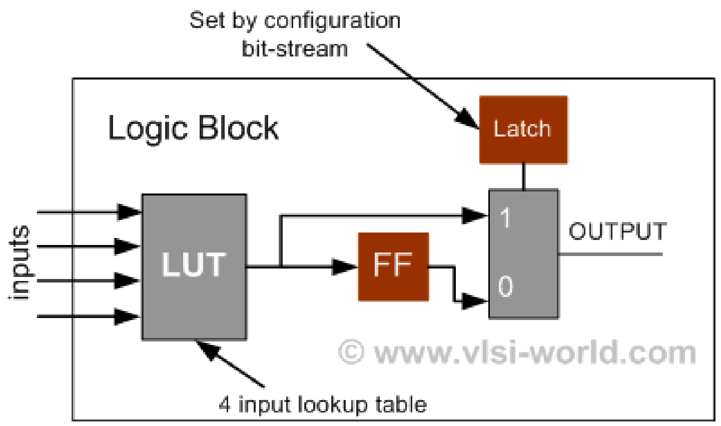
INTERNAL STRUCTURE OF AN FPGA

FPGAs, alternative to the custom ICs, can be used to implement an entire System On one Chip (SOC). The main advantage of FPGA is ability to reprogram. User can reprogram an FPGA to implement a design and this is done after the FPGA is manufactured. This brings the name "Field Programmable."

Custom ICs are expensive and takes long time to design so they are useful when produced in bulk amounts. But FPGAs are easy to implement within a short time with the help of Computer Aided Designing (CAD) tools.

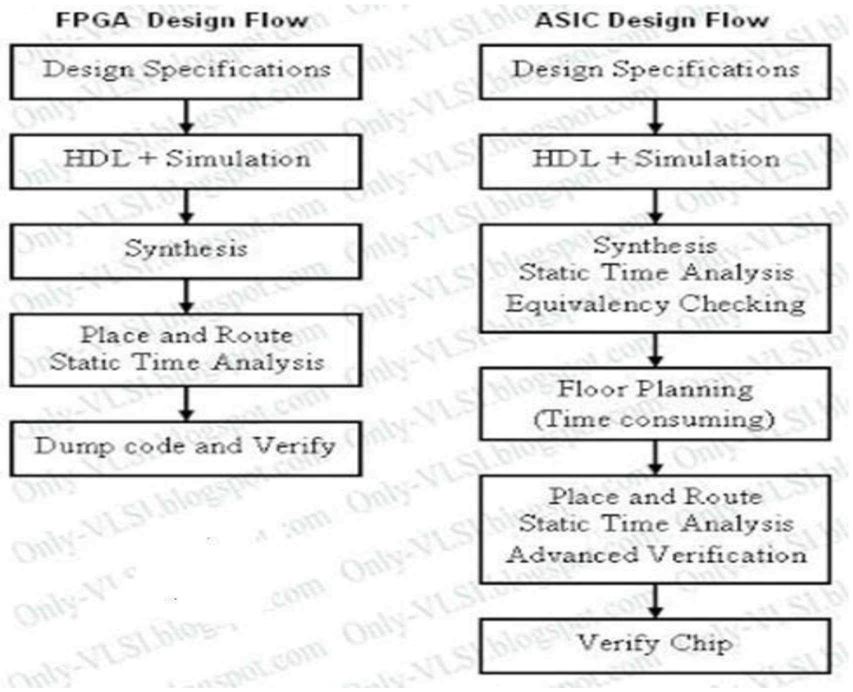
XILINX FPGA

Xilinx logic block consists of one Look Up Table (LUT) and one FlipFlop. An LUT is used to implement number of different functionalities. The input lines to the logic block go into the LUT and enable it. The output of the LUT gives the result of the logic function that it implements and the output of logic block is registered or unregistered output from the LUT.



4-input lut based implementation of logic block.

FPGA/ASIC DESIGN FLOW OVERVIEW



PART – A HDL Experiments Using XILINX

EXPERIMENT NO: 01

NAME OF THE EXPERIMENT: Verilog code to realize all the logic gates

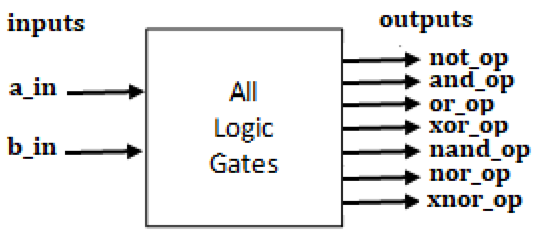
THEORY: A logic gate is an electronic circuit/device which makes the logical decisions alternatively a logic gate performs a logical operation on one or more logic inputs and produces a single logic output. The logic normally performed is Boolean logic and is most commonly found in digital circuits. Logic gates are primarily implemented using diodes or transistors. The logic gates are broadly classified into 3 types:

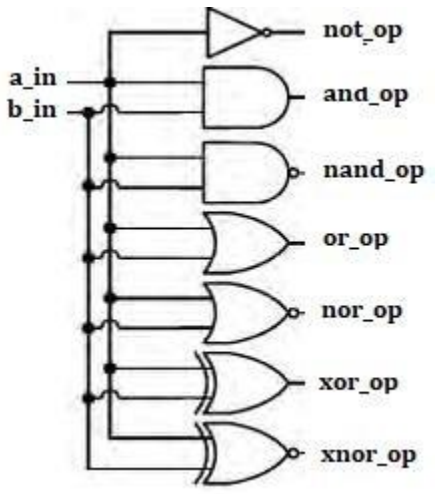
Basic gates::AND, OR, NOT / INVERTER

Universal gates:: NAND, NOR

Special gates:: XOR, XNOR

Logic diagram:





Truth Table:

Inputs		Outputs						
a_in	b_in	not_op	and_op	nand_op	or_op	nor_op	xor_op	xnor_op
0	0	1	0	1	0	1	0	1
0	1	1	0	1	1	0	1	0
1	0	0	0	1	1	0	1	0
1	1	0	1	0	1	0	0	1

Verilog code:

```
module gates(not_op, and_op, nand_op, or_op, nor_op, xor_op, xnor_op, a_in, b_in);
output not_op, and_op, nand_op, or_op, nor_op, xor_op, xnor_op;
input a_in, b_in;


        assign not_op= ~a_in;
        assign and_op=a_in&b_in;
        assign nand_op=~(a_in&b_in);
        assign or_op=a_in|b_in;
        assign nor_op=~(a_in|b_in);
        assign xor_op=a_in^b_in;
        assign xnor_op=~(a_in^b_in);
endmodule
```

Result:

Date:	**Staff's Sign:**

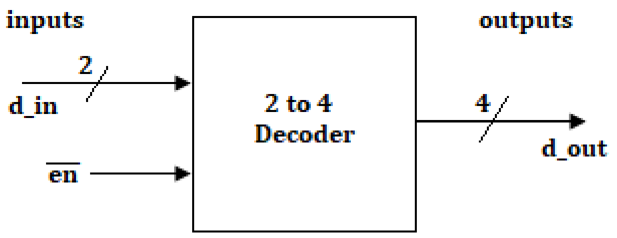
EXPERIMENT NO: 02

NAME OF THE EXPERIMENT: Verilog program for the following combinational designs.

A) 2 to 4 Decoder

Theory: A decoder is a multiple input, multiple output logic circuit that converts coded inputs into coded outputs where the input and output codes are different. The enable inputs must be **ON** for the decoder to function, otherwise its outputs assume a „disabled" output code word. Decoding is necessary in applications such as data multiplexing, seven segment display and memory address decoding. A decoder is a device which does the reverse operation of an encoder, undoing the encoding so that the original information can be retrieved. The same method used to encode is usually just reversed in order to decode. It is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines.

Symbol:



Truth Table:

Inputs			Outputs			
En	d_in [1]	d_in [0]	d_out[3]	d_out[2]	d_out[1]	d_out[0]
1	x	x	z	z	z	Z
0	0	0	0	0	0	1
0	0	1	0	0	1	0
0	1	0	0	1	0	0
0	1	1	1	0	0	0

<u>Verilog code</u>:

```
module decoder (d_op, d_in, en);

output [3:0] d_op;

input [1:0] d_in;

input en;

reg [3:0] d_op;

always @(d_in,en)

begin

        if (en==1)

                d_op=4’bzzzz;

        else

                case (d_in)

                        2'b00: d_op = 4'b0001;

                        2'b01: d_op = 4'b0010;

                        2'b10:d_op = 4'b0100;

                        2'b11: d_op = 4'b1000;

                        default: d_op = 4'bxxxx;

                endcase

end

endmodule
```

Result:

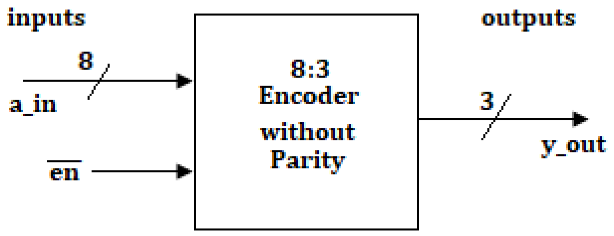
Date:	**Staff's Sign:**

B). 8 to 3 Encoder:

THEORY: An encoder is a digital circuit which performs the inverse of decoder. An encoder has 2^N input lines and N output lines. In encoder the output lines generate the binary code corresponding to input value. The decimal to BCD encoder usually has 10 input lines and 4 output lines. The decoder decimal data as an input for decoder an encoded BCD output is available at 4 output lines. An encoder is a device, circuit, transducer, software program, algorithm or person that converts information from one format or code to another, for the purposes of standardization, speed, secrecy, security or compressions.

i) Without priority:

Symbol:



Truth Table

Inputs									Outputs		
en	a_in[7]	a_in[6]	a_in[5]	a_in[4]	a_in[3]	a_in[2]	a_in[1]	a_in[0]	y_out[3]	y_out[3]	y_out[3]
0	X	X	x	x	x	x	x	x	z	z	z
1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	1	0	0	0	1	0
1	0	0	0	0	1	0	0	0	0	1	1
1	0	0	0	1	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	0	1	1	1

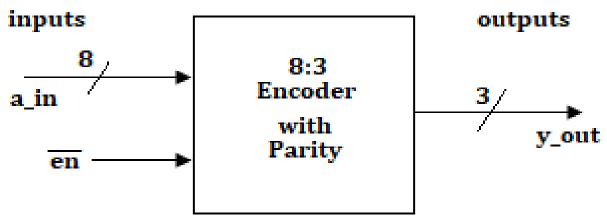
Verilog Code:

```
        module encoder8_3(en, a_in, y_op);
        input en;
        input [7:0] a_in;
        output [2:0] y_op;
        reg [2:0] y_op;
        always @ (a_in,en)
                begin
                if(en==1 )
                        y_op =3’bzzz;
                else
                case (a_in)
                8'b00000001: y_op = 3'b000;
                        8'b00000010: y_op = 3'b001;
                8'b00000100: y_op = 3'b010;
                8'b00001000: y_op = 3'b011;
                8'b00010000: y_op = 3'b100;
                8'b00100000: y_op = 3'b101;
                8'b01000000: y_op = 3'b110;
                8'b10000000: y_op = 3'b111;
                default: y_op =3'bxxx;
                endcase
        end
endmodule
```

Result:

ii) With priority:

Symbol:



Truth Table:

Inputs								Outputs		
a_in[7]	a_in[6]	a_in[5]	a_in[4]	a_in[3]	a_in[2]	a_in[1]	a_in[0]	y_out[3]	y_out[3]	y_out[3]
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	x	0	0	1
0	0	0	0	0	1	x	x	0	1	0
0	0	0	0	1	x	x	x	0	1	1
0	0	0	1	x	x	x	x	1	0	0
0	0	1	x	x	x	x	x	1	0	1
0	1	X	x	x	x	x	x	1	1	0
1	x	X	x	x	x	x	x	1	1	1

Verilog Code:

```
module prio_enco(en, a_in, y_op);

input en;

input [7:0] a_in;

output [2:0] y_op;

reg [2:0] y_op;

always @ (a_in,en)

        begin

        case (a_in)

        8'b00000001: y_op = 3'b000;

        8'b0000001x: y_op= 3'b001;

        8'b000001xx: y_op= 3'b010;

        8'b00001xxx: y_op= 3'b011;

        8'b0001xxxx: y_op= 3'b100;

        8'b001xxxxx: y_op= 3'b101;

        8'b01xxxxxx: y_op= 3'b110;

        8'b1xxxxxxx: y_op= 3'b111;

                default: y_op=3'bxxx;

        endcase

end

endmodule
```

Result:

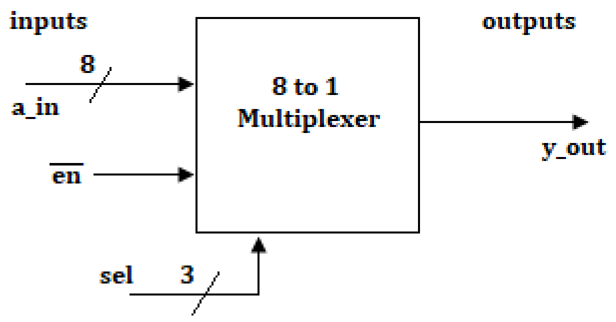
Date:	**Staff's Sign:**

C). Multiplexer:

Theory: Multiplexer is a digital switch. It allows digital information from several sources to be rooted on to a single output line. The basic multiplexer has several data input lines and a single output line. The selection of a particular input line is controlled by a set of selection lines. Normally there are 2^N input lines and N selection lines whose bit combinations determine which input is selected. Therefore multiplexer is many into one and it provides the digital equivalent of an analog selector switch.

i). 8:1 Multiplexer:

Symbol:



Truth Table:

Select Inputs			Inputs								Outputs
sel [2]	sel [1]	sel [0]	a_in [7]	a_in [6]	a_in [5]	a_in [4]	a_in [3]	a_in [2]	a_in [1]	a_in [0]	y_out
0	0	0	0	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	0	0	1
0	1	1	0	0	0	0	1	0	0	0	1
1	0	0	0	0	0	1	0	0	0	0	1
1	0	1	0	0	1	0	0	0	0	0	1
1	1	0	0	1	0	0	0	0	0	0	1
1	1	1	1	0	0	0	0	0	0	0	1

Verilog Code:

```
module mux8_1(i_in, sel, y_out);

input [7:0] a_in;

input [2:0] sel;

output y_out;

regy_out;

always@ (i_in,sel )

        begin

        case (sel)

                3'b000:y_out=i_in[0];

                3'b001:   y_out=i_in[1];

                3'b010:   y_out=i_in[2];

                3'b011:   y_out=i_in[3];

                3'b100:   y_out=i_in[4];

                3'b101:   y_out=i_in[5];

                3'b110:   y_out=i_in[6];

                3'b111:   y_out=i_in[7];

                default: y_out =3'b000;

        endcase

end

endmodule
```

Result:

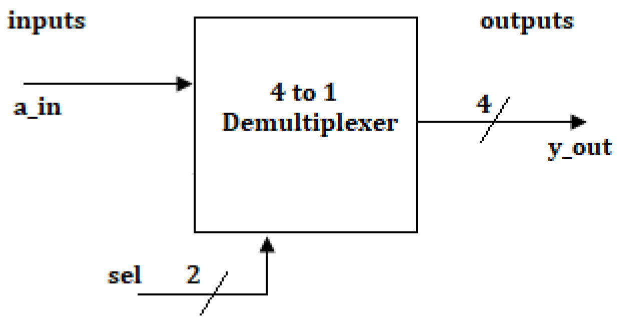
Date:	**Staff's Sign:**

D). De-Multiplexer:

Theory: A De-multiplexer is a circuit that receives information on a single line and transmits this information on one of 2^N output lines. The selection of specific output lines is controlled by the value of N selection lines. The single input variable din as a path to all 4 outputs but the input information is directed to only one of the output lines.

i).1:4 De-Multiplexer:

Symbol:



Truth Table:

Inputs			Outputs			
sel[1]	sel[0]	a_in	y-_out[3]	y-_out[2]	y-_out[1]	y-_out[0]
0	0	1	0	0	0	1
0	1	1	0	0	1	0
1	0	1	0	1	0	0
1	1	1	1	0	0	0

Verilog Code:

```
module demux1_4(a_in, sel, y_out);

input a_in;

input [1:0] sel;

output [3:0] y_out;

reg [3:0] y_out;

always @(a_in, sel)

begin

        case (sel)

                2’b00:begin    y_out[0]=a_in; y_out[1]= 1’b0;

                        y_out[2]= 1’b0;y_out[3]=1’b0; end

                2'b01: begin y_out[0]= 1’b0;y_out[1]=a_in;

                        y_out[2]= 1’b0;y_out[3]=1’b0; end

                2'b10: begin y_out[0]= 1’b0;y_out[1]=1’b0;

                        y_out[2]=a_in; y_out[3]=1’b0; end

                2'b11: begin y_out[0]= 1’b0; y_out[1]= 1’b0;

                        y_out[2]=1’b0;y_out[3]=a_in; end

                                default: y_out=3'b000;

        endcase

end

endmodule
```

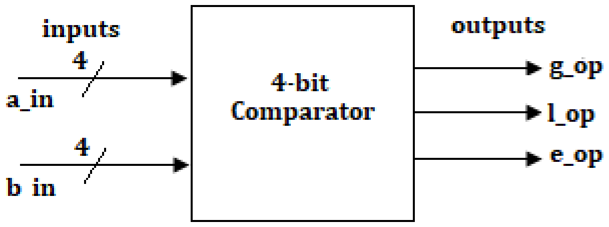
Result:

Date:	**Staff's Sign:**

E). 4-Bit Comparator:

Theory: Comparator is a special combinational circuit designed primarily to compare the relative magnitude of 2 binary numbers. It receives 2N bit numbers A and B as inputs and the outputs are A>B, A=B and A<B. Depending upon the relative magnitudes of the 2 numbers one the outputs will be high.

Symbol:



Truth Table:

Inputs		Outputs		
		a_in>b_in	a_in = b_in	a_in<b_in
a_in	b_in	g_op	e_op	l_op
1100	0011	1	0	0
0110	0110	0	1	0
1000	1110	0	0	1

Verilog Code:

```
module comparator(a_in, b_in, L_op,g_op,e_op);

input [3:0] a_in;

input [3:0] b_in;

output L_op;

output g_op;

output e_op;

regL_op,g_op,e_op;

always @ (a_in,b_in)

        begin

                if (a_in<b_in)

                        L_op=1'b1;

                else

                        L_op=1'b0;

                if (a_in>b_in)

                        g_op=1'b1;

                else

                        g_op=1'b0;

                if (a_in==b_in)

                        e_op=1'b1;

                else

                        e_op=1'b0;

        end

endmodule
```

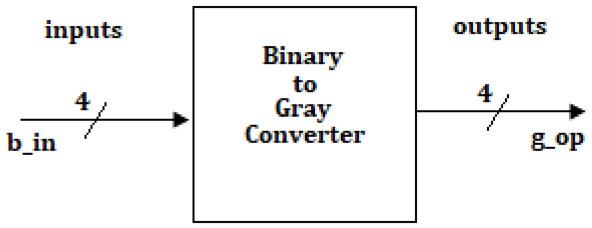
Result:

Date:	**Staff's Sign:**

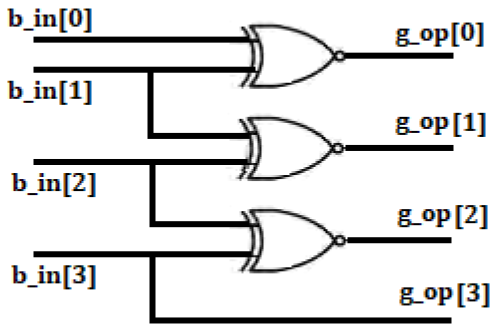
F). 4-Bit Binary to Gray Converter:

Theory: The Gray code is unweighted and is not an arithmetic code: that is there are no specific weights assigned to the bit positions. The important feature of the gray code is that it exhibits only a single bit change from one code word to the next in sequence. This property is important in many applications, such as shaft position encoders, where error susceptibility increases with the number of bit changes between adjacent numbers in a sequence.

Symbol:



Logic diagram Boolean equation:



g_op[3]=b_in[3]
g_op[2]=b_in[3]$\oplus$b_in[2]
g_op[1]=b_in[2]$\oplus$b_in[1]
g_op[0]=b_in[1]$\oplus$b_in[0]

Truth Table:

Inputs					Outputs			
Decimal	Binary				Gray			
	b_in[3]	b_in[2]	b_in[1]	b_in[0]	g_in[3]	g_in[2]	g_in[1]	g_in[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1

6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

Verilog Code:

```
module binary_gray(b_in, g_op);

input [3:0] b_in;

output [3:0] g_op;

assigng_op[3] = b_in[3];

assigng_op[2] = b_in[3] ^ b_in[2];

assigng_op[1] = b_in[2] ^ b_in[1];

assigng_op[0] = b_in[1] ^ b_in[0];

endmodule
```

Result:

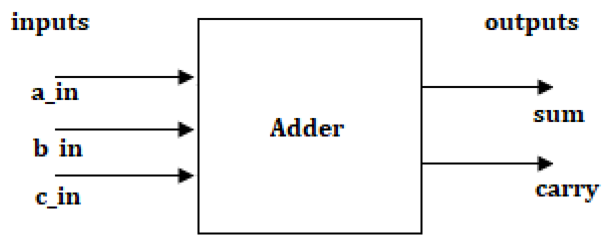
Date:	Staff's Sign:

EXPERIMENT NO 3:

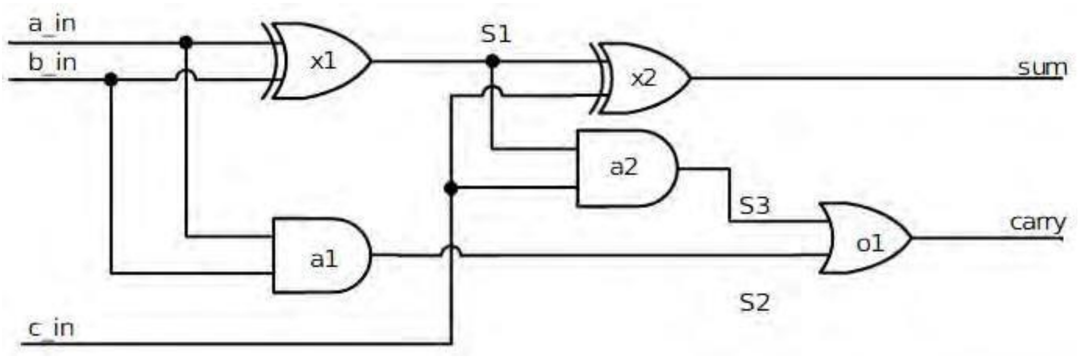
NAME OF THE EXPERIMENT: VHDL & Veilog code to describe the functions of a Full adder.

THEORY: The **full-adder** circuit adds three one-bit binary numbers (C A B) and outputs two one-bit binary numbers, a sum (S) and a carry (C1). The **full-adder**is usually a component in a cascade of **adders**, which add 8, 16, 32, etc. binary numbers.

Symbol:



Logic diagram:

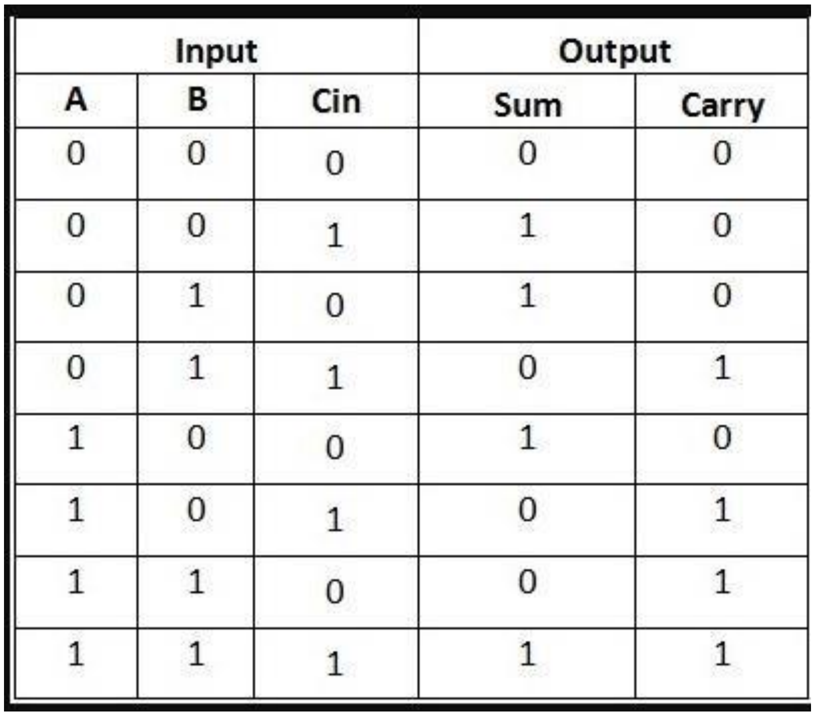


Boolean equations:

sum = a_in⊕b_in⊕c_in

carry= (a_in.b_in)+(b_in.c_in)+(a_in.b_in)

Truth Table:



Input			Output	
A	B	Cin	Sum	Carry
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

VHDL Code:

Data flow:

```
entity fulladder is
port (a_in, b_in, c_in: in std_logic;
sum, carry: out std_logic);
end fulladder;


architecture dataflow of fulladder is
begin
        sum<= a_inxorb_inxorc_in;
        carry<= (a_in and b_in) or (b_in and c_in) or (a_in andb_in);
end dataflow;
```

Behavioural:

```
entity fulladder is
port (abc: instd_logic_vector(2 downto 0);
          sum, carry: out std_logic);
end fulladder;


architecture behavioral of fulladder is
begin
          process(abc)
          begin
                    case (abc) is
                              when”000”=>sum<=’0’; carry<=’0’;
                              when”001”=>sum<=’1’; carry<=’0’;
                              when”010”=>sum<=’1’; carry<=’0’;
                              when”011”=>sum<=’0’; carry<=’1’;
                              when”100”=>sum<=’1’; carry<=’0’;
                              when”101”=>sum<=’0’; carry<=’1’;
                              when”110”=>sum<=’0’; carry<=’1’;
                              when”111”=>sum<=’1’; carry<=’1’;
                              when others=>null;
                    end case;
          end process;
end ;
```

Structural:

```
entity fulladder is

port (a_in, b_in, c_in: in std_logic;

sum, carry: out std_logic);

end fulladder;


architecture structural of fulladder is


component halfadder is

port (p, q: in std_logic;

        r, s: out std_logic);

end component;


signal temp1, temp2, temp3: std_logic;

        begin

                ha1: halfadder port map (a_in, b_in, temp1,temp2);

                ha2: halfadder port map (temp1, c_in, sum, temp3);

                carry<=temp2 or temp3;

end structural;
```

COMPONENT PROGRAM:

```
entity    halfadder    is

port (p, q: in std_logic;

        r, s: out std_logic);

end halfadder;
```

```
architecture dataflow of halfadder is
        begin
                r<= p xor q;
                s<= p and q;
end;
```

Verilog Code:

Data flow:

```
module fulladder(a_in, b_in, c_in, sum, carry);
input a_in, b_in,c_in;
output sum, carry;

        assign sum = a_in^b_in^c_in;
        assign carry = (a_in&b_in)|(b_in&c_in)|(a_in&c_in);
endmodule
```

Behavioural:

```
module fulladder(abc, sum, carry);
input [2:0] abc;
output sum,carry;
reg sum,carry;
always@(abc)
        begin
                case (abc)
                        3'b000:begin sum=1'b0; carry=1'b0;end
```

```
                                        3'b001:begin sum=1'b1; carry=1'b0;end

                                        3'b010:begin sum=1'b1; carry=1'b0;end

                                        3'b011:begin sum=1'b0; carry=1'b1;end

                                        3'b100:begin sum=1'b1; carry=1'b0;end

                                        3'b101:begin sum=1'b0; carry=1'b1;end

                                        3'b110:begin sum=1'b0; carry=1'b1;end

                                        3'b111:begin sum=1'b1; carry=1'b1;end

 endcase

 end

endmodule
```

Structural:

```
Module fulladder(a_in, b_in, c_in, sum, carry);

Input a_in,b_in, c_in;

Output sum,carry;

wire temp1, temp2, temp3;


halfadder ha1 (a_in, b_in, temp1, temp2);

halfadder ha2 (c_in, temp1, sum, temp3);

assign carry= temp3 | temp2;

endmodule


module halfadder(a, b, s, c);

input a, b;
```

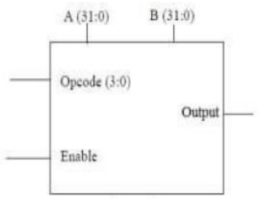
```
output s, c;

        assign s= a^b;

        assign c= a &b;

endmodule
```

Result:

Date:	**Staff's Sign:**

EXPERIMENT NO 4:

NAME OF THE EXPERIMENT: Verilog code to model 32-bit ALU using the schematic diagram shown below.



- ALU should use combinational logic to calculate an output based on the four

bit op-code input.

- ALU should pass the result to the out bus when enable line in high, and tristate

the out bus when the enable line is low.

- ALU should decode the 4 bit op-code according to the example given below.

OPCODE	ALU OPERATION
1	A+B
2	A-B
3	A Complement
4	A*B
5	A AND B
6	A OR B
7	A NAND B
8	A XOR B

Theory: An **ALU** is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs). A single CPU, FPU or GPU may contain multiple ALUs

Verilog Code:

```
module alu(a, b, sel,en,y,y_mul);

input [31:0] a;

input [31:0] b;

input en;

input [2:0] sel;

output [31:0] y;

output[63:0]y_mul;

reg [31:0] y;

reg [63:0] y_mul;

always @(a, b , sel)

        begin

        if (en==1)

                case (sel)

                        3'b000:y=a+b;

                        3'b001:y=a-b;

                        3'b010:y=~a;

                        3'b011:y_mul=a*b;

                        3'b100:y= a&b;

                        3'b101:y=a|b;

                        3'b110:y=~(a&b);

                        3'b111:y=a^b;

                default:begin end

                endcase
        else
        begin
                y=32’bz;
                y_mul=64’bz
         end
         end
endmodule
```

Result:

Date:	Staff’s Sign:

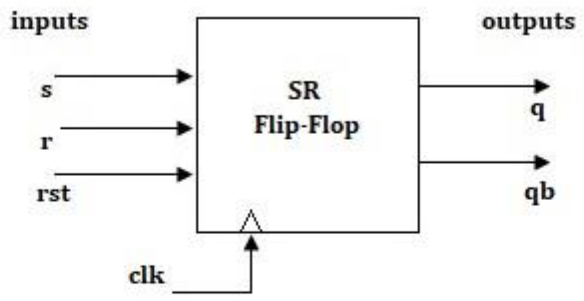
EXPERIMENT NO 5:

NAME OF THE EXPERIMENT: Verilog code for the following flip-flops, SR, D, JK,T.

A). SR Flip-flop:

Theory: A basic NAND gate **SR flip-flop** circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit. Then the **SR flip-flop** actually has three inputs, Set, Reset and its current output Q relating to it's current state or history.

Symbol:



Truth Table:

Inputs				outputs		
rst	Clk	s	r	q	qb	Action
1	↑	x	x	q	qb	No Change
0	↑	0	0	q	qb	No Change
0	↑	0	1	0	1	Reset
0	↑	1	0	1	0	Set
0	↑	1	1	-	-	Illegal

Verilog Code:

```
module sr_ff(sr, clk, rst, q, qb);

input [1:0]sr;

input rst, clk;

output q,qb;

reg q,qb;

always @ (posedgeclk)

        begin

                if (rst==1)

                begin

                        q=0;

                        qb=1;

                end

                else

                case (sr)

                        2'b00: begin q=q; qb=qb; end

                        2'b01: begin q=0; qb=1; end

                        2'b10: begin q=1; qb=0; end

                        2'b11: begin q=1'bx; qb=1'bx; end

                default:begin end

                endcase

        end

endmodule
```

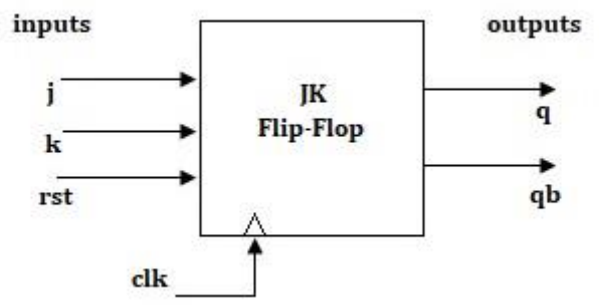
Result:

Date:	**Staff's Sign:**

B). J-K Flip flop:

Theory: The **JK flip flop** is basically a gated SR **flip-flop** with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic level “1”.

Symbol:



Truth Table:

Inputs				outputs		
rst	clk	j	k	q	qb	Action
1	↑	x	x	q	qb	No Change
0	↑	0	0	q	qb	No Change
0	↑	0	1	0	1	Reset
0	↑	1	0	1	0	Set
0	↑	1	1	q'	qb'	Toggle

Verilog Code:

```
module jk_ff(j, k, clk, reset, q, qb);

input [1:0]jk;

input clk,rst;

output q, qb;

reg q, qb;
```

```
          reg [22:0] div;

          reg clkdiv;

          always @ (posedge clk)

                    begin

                              div = div+1'b1;

                              clkdiv = div[22];

                    end

always @ (posedge clkdiv) begin

                              if(rst==1)

                    begin

                              q=0;

                              qb=1;

                    end

                              else

                    case (jk)

                              2'b00: begin q=q; qb=qb; end

                              2'b01: begin q=0; qb=1; end

                    2'b10: begin q=1; qb=0; end

                    2'b11: begin q=~(q); qb=~(qb); end

                    default: begin end

                    endcase

                    end

          endmodule
```

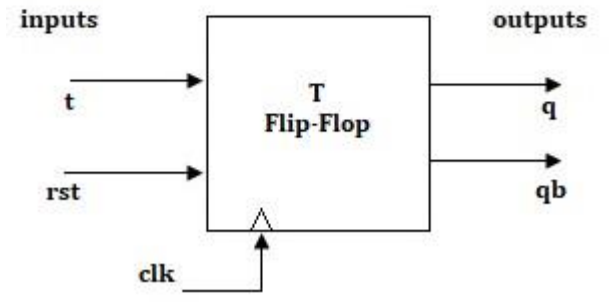
Result:

Date:	**Staff's Sign:**

C). T-Flip flop:

Theory: The **T flip flop** is the modified form of JK**flip flop**. The Q and Q' represents the output states of the **flip-flop**. According to the table, based on the input the output changes its state. But, the important thing to consider is all these can occur only in the presence of the clock signal.

Symbol:



Truth Table:

Inputs			Outputs		
Rst	clk	t	q	qb	Action
1	↑	x	q	qb	No Change
0	↑	0	q	qb	No Change
0	↑	1	q'	qb'	Toggle

Verilog Code:

```
module t_ff(t, clk, rst, q, qb);

input t, clk, rst;

output q, qb;

reg q,qb;

always @ (posedge clk)

        begin
```

```
                div = div+1'b1;

                clkdiv = div[22];

        end


always @ (posedge clkdiv)

        begin

                if (rst==1)

begin

                q=0;

                qb=1;

end

                else

                case ( t)

                        1’b0:begin q=q; qb=qb; end

                        1’b1:begin q=~(q); qb=~(qb); end

                default: begin end

                endcase

        end

endmodule
```

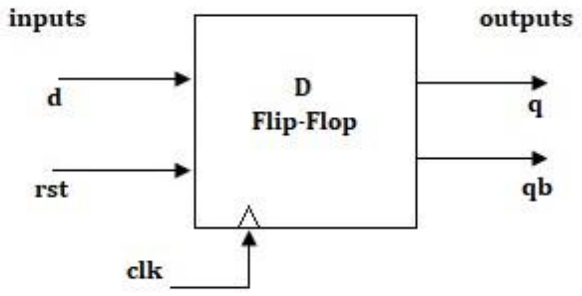
<u>Result:</u>

Date:	**Staff’s Sign:**

D). D-Flip flop:

Theory: The **D flip-flop** tracks the input, making transitions with match those of the input **D**. The **D** stands for "data"; this **flip-flop** stores the value that is on the data line. It can be thought of as a basic memory cell. A D **flip-flop** can be made from a set/reset **flip-flop**by tying the set to the reset through an inverter.

Symbol:



Truth Table:

Inputs			Outputs		
Rst	clk	d	q	qb	Action
1	↑	x	q	qb	No Change
0	↑	0	0	1	q=d
0	↑	1	1	0	q=d

Verilog Code:

```
module d_ff(d, rst, clk, q, qb);

input d;

input   rst;

input   clk;

output   q;

output qb;

reg q,qb;
```

```
always@(posedge clk)
        begin
                if (rst==1)
                begin
                        q=0;
                        qb=1;
                end
                else
                begin
                        q=d;
                        qb=~d;
                end
        end
endmodule
```

Result:

Date:	**Staff's Sign:**

EXPERIMENT NO 6:

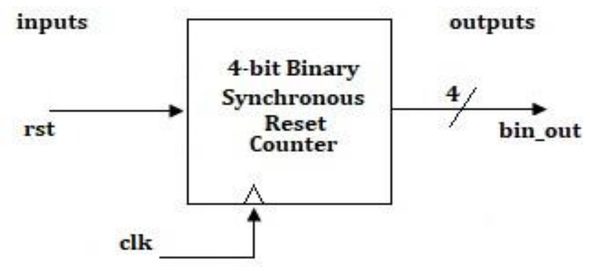
NAME OF THE EXPERIMENT: Counters

AIM: Design a 4 bit binary, BCD counters(Synchronous reset and asynchronous reset) and "any sequence" counters, using verilog code.

A). Binary Synchronous counters:

Theory: A **synchronous** circuit is a **digital** circuit in which the changes in the state of memory elements are synchronized by a clock signal. In a sequential **digital logic** circuit, data is stored in memory devices called flip-flops or latches.(0-15)

Symbol:



i). Up Counter:

Truth Table:

Rst	clk	A	B	C	D
1	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1

0	1	1	1	1	0
0	1	1	1	1	1

Verilog Code:

```
module bin_up(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial

begin

count=4'b0000;

end

always@(posedge clk)

begin

if(rst)

count=4'b0000;

else

count=count+4'b0001;

end

endmodule
```

Result:

ii). Down Counter:

Truth Table:

Rst	clk	A	B	C	D
1	1	1	1	1	1
0	1	1	1	1	0
0	1	1	1	0	1
0	1	1	1	0	0
0	1	1	0	1	1
0	1	1	0	1	0
0	1	1	0	0	1
0	1	1	0	0	0
0	1	0	1	1	1
0	1	0	1	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	0	1
0	1	0	0	0	0

Verilog Code:

```
module bin_down(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial

begin

count=4’b1111;

end

always@(posedge clk)

begin

if(rst)
```

```
count=4'b0000;
else
count=count-4'b0001;
end
endmodule
```

Result:

iii).Up-Down counter:

Verilog Code:

```
module updown-counter(clk,rst,updown,count);
input clk,rst,updown;
output[3:0]count;
reg[3:0]count;
always@(posedge clk )
begin
if(rst)
```

```
count<=4’b0000;
else
if(updown==1)
if(count==4’b1111)
count<=4’b0000;
else
count<=count+4’b0001;
else
count<=4’b1111;
else
count<=count-4’b0001;
end
endmodule
```

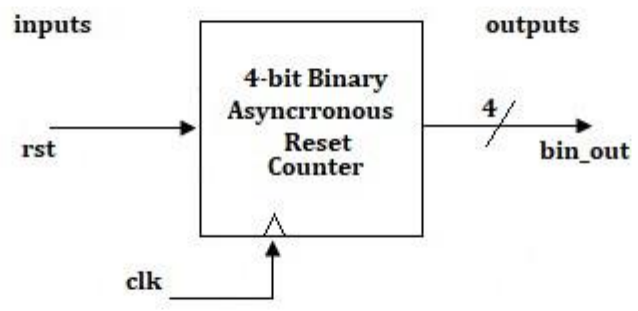
Result:

Date:	**Staff’s Sign:**

B). Binary Asynchronous Counters:

Theory: This type of circuit is contrasted with synchronous **circuits**, in which changes to the signal values in the circuit are triggered by repetitive pulses called a clock signal. Most**digital** devices today use synchronous **circuits.... Asynchronous circuits** are an active area of research in**digital logic** design.(0-15)

Symbol:



i). Up-Counter:

Truth Table:

Rst	Clk	A	B	C	D
1	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1
0	1	1	0	1	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	0	1
0	1	1	1	1	0
0	1	1	1	1	1

Verilog Code:

```
module bin_up(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial

begin

count=4’b0000;

end

always@(posedge clk or posedge rst)

begin

if(rst)

count=4’b0000;

else

count=count+4’b0001;

end

endmodule
```

Result:

ii). Down-Counter:

Truth Table:

Rst	Clk	A	B	C	D
1	1	1	1	1	1
0	1	1	1	1	0
0	1	1	1	0	1
0	1	1	1	0	0
0	1	1	0	1	1
0	1	1	0	1	0
0	1	1	0	0	1
0	1	1	0	0	0
0	1	0	1	1	1
0	1	0	1	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	0	1
0	1	0	0	0	0

Verilog Code:

```
module bin_down(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial

begin

count=4’b1111;

end

always@(posedge clk or posedge rst)

begin

if(rst)

count=4’b0000;
```

```
else
count=count-4’b0001;
end
endmodule
```

Result:

iii). Up-Down Counter:

Verilog Code:

```
module updown-counter(clk,rst,updown,count);
input clk,rst,updown;
output[3:0]count;
reg[3:0]count;
always@(posedge clk or posedge rst)
begin
if(rst)
count<=4’b0000;
else
```

```
if(updown==1)
if(count==4'b1111)
count<=4'b0000;
else
count<=count+4'b0001;
else
count<=4'b1111;
else
count<=count-4'b0001;
end
endmodule
```

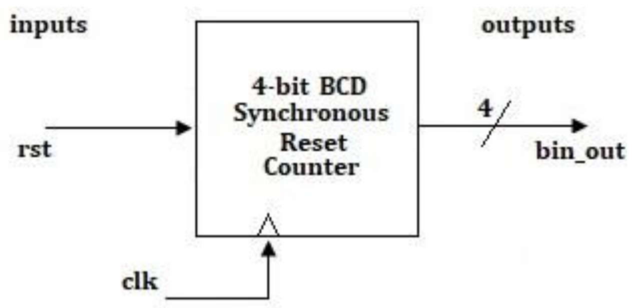
Result:

Date:	**Staff's Sign:**

C). BCD Synchronous Counter:

Theory: : A **synchronous** circuit is a **digital** circuit in which the changes in the state of memory elements are synchronized by a clock signal. In a sequential **digital logic** circuit, data is stored in memory devices called flip-flops or latches.(0-9)

Symbol



i). Up-Counter:

Truth Table:

Rst	clk	A	B	C	D
1	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1

Verilog Code:

```
module bdc_up(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial
```

```
begin
count=4’b0000;
end
always@(posedge clk)
begin
if(rst)
count=4’b0000;
else if
(count<=4’b1001)
count=count+4’b0001;
else
count=4’b0000;
end
endmodule
```

Result:

ii). Down-Conter:

Truth Table:

Rst	Clk	A	B	C	D
1	1	1	0	0	1
0	1	1	0	0	0
0	1	0	1	1	1
0	1	0	1	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	0	1
0	1	0	0	0	0

Verilog Code:

```
module bdc_down(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial

begin

count=4’b1001;

end

always@(posedge clk)

begin

if(rst)

count=4’b0000;

else if

(count<=4’b1001)

count=count-4’b0001;
```

```
else
count=4’b1001;
end
endmodule
```

Result:

iii). Up-Down Counter:

Verilog Code:

```
module updown-counter(clk,rst,updown,count);
input clk,rst,updown;
output[3:0]count;
reg[3:0]count;
always@(posedge clk )
begin
if(rst)
```

```
count<=4’b0000;
else
if(updown==1)
if(count==4’b1001)
count<=4’b0000;
else
count<=count+4’b0001;
else
if(count==0)
count<=4’b1001;
else
count<=count-4’b0001;
end
endmodule
```

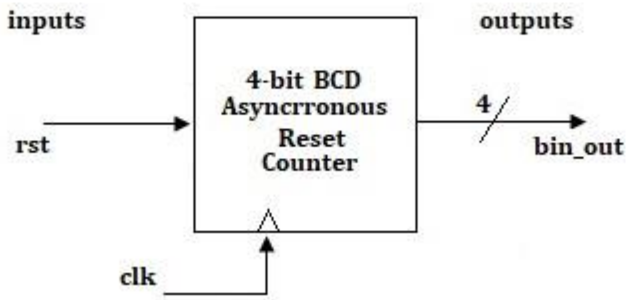
<u>Result:</u>

Date:	**Staff’s Sign:**

D) BCD Asynchronous Counter:

Theory: This type of circuit is contrasted with synchronous **circuits**, in which changes to the signal values in the circuit are triggered by repetitive pulses called a clock signal.
Most **digital** devices today use synchronous **circuits... Asynchronous circuits** are an active area of research in **digital logic** design.(0-9).

Symbol :



i). Up-Counter:

Truth Table:

Rst	Clk	A	B	C	D
1	1	0	0	0	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	1	0	0
0	1	0	1	0	1
0	1	0	1	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	0	1

Verilog Code:

```
module bdc_up(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;
```

```
initial
begin
count=4’b0000;
end
always@(posedge clk or posedge rst)
begin
if(rst)
count=4’b0000;
else if
(count<=4’b1001)
count=count+4’b0001;
else
count=4’b0000;
end
endmodule
```

Result:

ii) Down Counter:

Truth Table:

Rst	clk	A	B	C	D
1	1	1	0	0	1
0	1	1	0	0	0
0	1	0	1	1	1
0	1	0	1	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	0	1	0
0	1	0	0	0	1
0	1	0	0	0	0

Verilog Code:

```
module bdc_down(rst,clk,count)

input clk,rst;

output[3:0] count;

reg[3:0]count;

initial

begin

count=4'b1001;

end

always@(posedge clk or posedge rst)

begin

if(rst)

count=4'b0000;

else if

(count<=4'b1001)

count=count-4'b0001;
```

```
else
count=4'b1001;
end
endmodule
```

Result:

iii) Up-Down Counter:

Verilog Code:

```
module updown-counter(clk,rst,updown,count);
input clk,rst,updown;
output[3:0]count;
reg[3:0]count;
always@(posedge clk or posedge rst )
begin
if(rst)
count<=4'b0000;
else
```

```
if(updown==1)
if(count==4'b1001)
count<=4'b0000;
else
count<=count+4'b0001;
else
if(count==0)
count<=4'b1001;
else
count<=count-4'b0001;
end
endmodule
```

Result:

Date:	**Staff's Sign:**

E). Any sequence:

Verilog Code:

```
module any-seq(rst,clk,updown,load,din,count);
input rst,clk,updown,load;
input[3:0]din;
output[3:0]count;
reg[3:0]count;
initial
begin
count=4’b0000;
end
always@(posedge clk)
if(rst)
count=4’b0000;
else if(load)
count=din;
else if(updown)
count=count+4’b0001;
else
count=count-4’b0001;
end
endmodule
```

Result:

Date:	**Staff's Sign:**

PART-B
HDL Experiments using Spartan

EXPERIMENT NO 1:

NAME OF THE EXPERIMENT: LCD Display

AIM: HDL Code to display messages on an alpha numeric LCD.

THEORY: A liquid-crystal display (LCD) is a flat-panel display or other electronically modulated optical device that uses the light-modulating properties of liquid crystals. Liquid crystals do not emit light directly, instead using a backlight or reflector to produce images in color or monochrome

CODE:

```
module LCD_DEMO(
 input P_Clk,
 output reg LCD_RS,LCD_EN,
 output reg [7:0] P_LCD
);
parameter Length = 53;
reg [32:0] delay = 32'hFFFFFFFF;
integer pointer = 0;
wire [8:0] memory[0:Length-1];
assign      memory[0] = {1'b0,8'h38};
assign memory[1] = {1'b0,8'h06};
assign memory[2] = {1'b0,8'h0C};
assign memory[3] = {1'b0,8'h01};
assign memory[20] ={1'b1,"*"};
assign memory[21] = {1'b1,"*"};
assign memory[22] = {1'b1," "};
assign memory[23] = {1'b1,"W"};
assign  memory[24] = {1'b1,"E"};
assign  memory[25] = {1'b1,"L"};
assign  memory[26] = {1'b1,"C"};
assign  memory[27] = {1'b1,"O"};
assign memory[28] = {1'b1," M"};
assign memory[29] = {1'b1," E"};
assign memory[30] = {1'b1," "};
assign  memory[31] = {1'b1,"T"};
assign  memory[32] = {1'b1,"O"};
assign memory[33] = {1'b1," "};
assign memory[34] = {1'b1,"*"};
assign memory[35] = {1'b1,"* "};
// Shift to second Line of LCD
assign memory[36] = {1'b0,8'hC0};
// Character that should be displayed on the LCD.
assign memory[37] ={1'b1,"B"};
assign  memory[38] = {1'b1,"G"};
assign  memory[39] = {1'b1,"S"};
assign  memory[40] = {1'b1,"I"};
assign memory[41] = {1'b1,"T"};
assign memory[42] = {1'b1," "};
assign memory[43] = {1'b1," "};
assign  memory[44] = {1'b1,"B"};
assign  memory[45] = {1'b1,"G"};
assign memory[46] = {1'b1," "};
assign memory[47] = {1'b1," N"};
assign  memory[48] = {1'b1,"A"};
assign  memory[49] = {1'b1,"G"};
assign  memory[50] = {1'b1,"A"};
assign  memory[51] = {1'b1,"R"};
assign memory[52] = {1'b1," "};
always @(posedge P_Clk)
 begin
    counter = counter + 1;
    if(pointer > Length)
                    LCD_EN = 'b0;
              else
               LCD_EN = counter[15];
 end
always @(negedge LCD_EN)
  begin
    LCD_RS = memory[pointer][8];
    P_LCD = memory[pointer][7:0];
pointer = pointer + 1;
  end
```

```
endmodule
```

Result:

Date:	**Staff's Sign:**

EXPERIMENT NO: 2

NAME OF THE EXPERIMENT: HEX KEYPAD

AIM: HDL Code to interface Hex key pad and display the key code on seven segment display.

THEORY: The hex keypad is a peripheral that connects to the DE2 through JP1 or JP2 via a 40-pin ribbon cable. It has 16 buttons in a 4 by 4 grid, labeled with the hexadecimal digits 0 to F. An example of this can been seen in Figure 7.1, below. Internally, the structure of the hex keypad is very simple. Wires run in vertical columns (we call them C0 to C3) and in horizontal rows (called R0 to R3). These 8 wires are available externally, and will be connected to the lower 8 bits of the port. Each key on the keypad is essentially a switch that connects a row wire to a column wire. When a key is pressed, it makes an electrical connection between the row and column. The internal structure of the hex keypad is shown in Figure7. 2. The specific mapping of hex keypad wires (C0 to C3 and R0 to R3) to pins is given n Table 7.1.

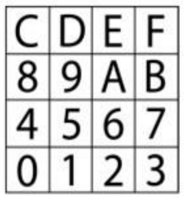


Figure 7.1: Hex keypad layout

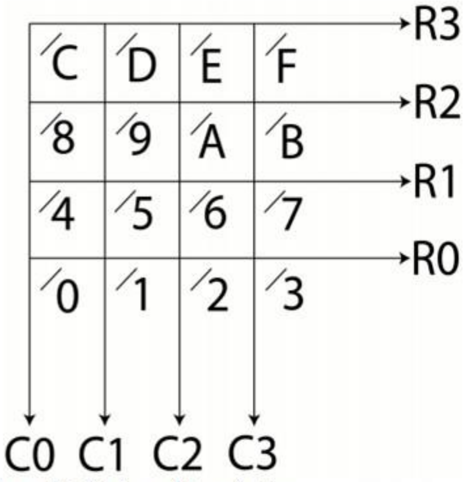
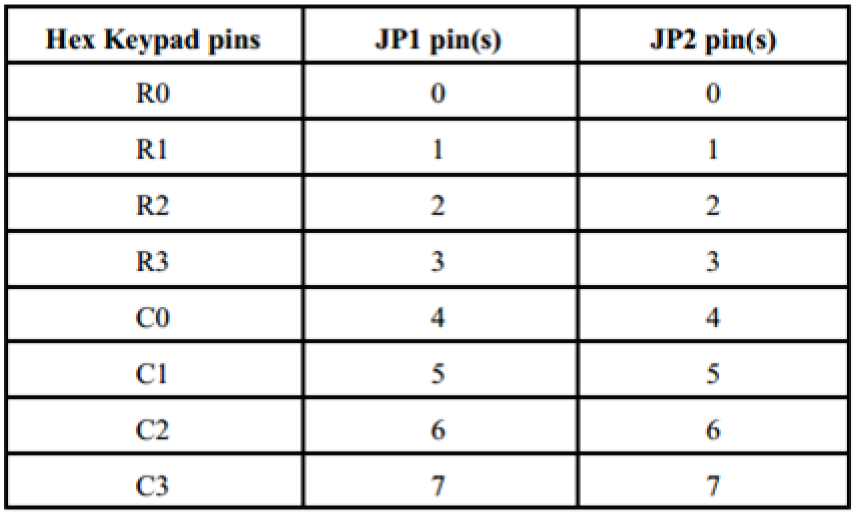


Figure 7.2: Hex keypad internal wiring

Table 7.1: Hex key pad pin details



Hex Keypad pins	JP1 pin(s)	JP2 pin(s)
R0	0	0
R1	1	1
R2	2	2
R3	3	3
C0	4	4
C1	5	5
C2	6	6
C3	7	7

At this point, you may be wondering exactly where the signals on the hex keypad come from. The keys just create a short between a row and column wire when pressed, but the row and column wires all come from JP1 or JP2, rather than connecting to power or ground.

CODE:

```
 module HEX_KEYPAD(
input P_Clk,
input  [3:0] MK_IN,  //Row In
output reg [3:0] MK_OUT, // Col Out
output reg [3:0] P_dig, // anode signals of the 7-segment LED display
output reg [7:0] P_7seg // cathode patterns of the 7-segment LED display
);


reg [2:0] state = 0;
reg [7:0] count = 0;
reg clk = 0;
 always @(posedge P_Clk)
begin
 if(count >= 50)
  begin
            clk = ~clk;
            count = 0;
  end
 else count = count + 1;
end

always @(posedge clk)
begin
            P_dig = 'b0001;

            case (state)
    0: begin
          P_7seg = 'b11111111; //null
                              MK_OUT = 'b1000;
                              state = 1;
                         end

              1: begin
                         if (MK_IN == 'b1000) P_7seg = 'b10001000; //0
```

```
                    if (MK_IN == 'b0100) P_7seg = 'b11101011; //1

                    if (MK_IN == 'b0010) P_7seg = 'b01001100; //2

                    if (MK_IN == 'b0001) P_7seg = 'b01001001; //3


                    MK_OUT = 'b0100;

                    state = 2;

end



    2: begin

              if (MK_IN == 'b1000) P_7seg = 'b00101011; //4

                    if (MK_IN == 'b0100) P_7seg = 'b00011001; //5

                    if (MK_IN == 'b0010) P_7seg = 'b00011000; //6

                    if (MK_IN == 'b0001) P_7seg = 'b11001011; //7


                    MK_OUT = 'b0010;

                    state = 3;

end



    3: begin

              if (MK_IN == 'b1000) P_7seg = 'b00001000; //8

                    if (MK_IN == 'b0100) P_7seg = 'b00001001; //9

                    if (MK_IN == 'b0010) P_7seg = 'b00001010; //A

                    if (MK_IN == 'b0001) P_7seg = 'b00111000; //B
```

```
                        MK_OUT = 'b0001;

                        state = 4;

     end



          4: begin

                   if (MK_IN == 'b1000) P_7seg = 'b10011100; //C

                        if (MK_IN == 'b0100) P_7seg = 'b01101000; //D

                        if (MK_IN == 'b0010) P_7seg = 'b00011100; //E

                        if (MK_IN == 'b0001) P_7seg = 'b00011110; //F

                        state = 0;

     end



  endcase

end


endmodule
```

<u>Result:</u>

Date:	**Staff's Sign:**

EXPERIMENT NO:3(a)

NAME OF THE EXPERIMENT: DC Motor

AIM: HDL Code to control speed, direction of dc motor

THEORY: A DC motor is any of a class of rotary electrical machines that converts direct current electrical energy into mechanical energy. The most common types rely on the forces produced by magnetic fields. Nearly all types of DC motors have some internal mechanism, either electromechanical or electronic, to periodically change the direction of current flow in part of the motor.

CODE:

```
module MOTOR_DC(
input P_Clk,
output reg [1:0] P_DCM, // DC Motor dir control
output reg P_DCMEN //DC Motor enable PWM
);

reg [20:0] count = 0;
reg [16:0] dur = 0;
reg dir = 0;

parameter speed = 100000; //change this value for speed
parameter duration = 1000; //change this value for changing direction timing

always @(posedge P_Clk)
```

```
begin
        count = count + 1;
        if (count == speed)
        begin
                count = 0;

                P_DCMEN = ~P_DCMEN;

                if (dir == 0)
                        P_DCM = 'b01; //Clockwise
                else
                        P_DCM = 'b10; //Anti-Clockwise

                dur = dur + 1;
                if(dur == duration)
                begin
                        dur = 0;
                        dir = ~dir;
                end
        end
end

endmodule
```

Result:

Date:	**Staff's Sign:**

EXPERIMENT NO: 3(b)

NAME OF THE EXPERIMENT: STEPPER MOTOR

AIM: HDL Code to control speed, direction of stepper motor.

THEORY: A **Stepper Motor** or a **step motor** is a brushless, synchronous motor which divides a full rotation into a number of steps. Unlike a brushless DC motor which rotates continuously when a fixed DC voltage is applied to it, a step motor rotates in discrete step angles. The **Stepper Motors** therefore are manufactured with steps per revolution of 12, 24, 72, 144, 180, and 200, resulting in stepping angles of 30, 15, 5, 2.5, 2, and 1.8 degrees per step. The stepper motor can be controlled with or without feedback. The stepper motor which we are using has step angle of **1.8 degree**

Stepper motor "step modes" include Full, Half and Microstep. The type of step mode output of any stepper motor is dependent on the design of the driver. Omegamation™ offers stepper motor drives with switch selectable full and half step modes, as well as microstepping drives with either switch-selectable or software-selectable resolutions.

FULL STEP: Standard hybrid stepping motors have 200 rotor teeth, or 200 full steps per revolution of the motor shaft. Dividing the 200 steps into the 360° of rotation equals a 1.8° full step angle. Normally, full step mode is achieved by energizing both windings while reversing the current alternately. Essentially one digital pulse from the driver is equivalent to one step.

HALF STEP: Half step simply means that the step motor is rotating at 400 steps per revolution. In this mode, one winding is energized and then two windings are energized alternately, causing the rotor to rotate at half the distance, or 0.9°. Although it provides approximately 30% less torque, half-step mode produces a smoother motion than full-step mode.

MICROSTEP: Microstepping is a relatively new stepper motor technology that controls the current in the motor winding to a degree that further subdivides the number of positions between poles. Omegamation microstepping drives are capable of dividing a full step (1.8°) into 256 microsteps, resulting in 51,200 steps per revolution (.007°/step). Microstepping is typically used in applications that require accurate positioning and smoother motion over a wide range of speeds. Like the half-step mode, microstepping provides approximately 30% less torque than full-step mode.

CODE:

```
module MOTOR_STEPPER(
input P_Clk,
output reg [3:0] P_STP // DAC Out
);

reg [20:0] count = 0;
reg [16:0] dur = 0;
reg [3:0] stpval = 'b0001;
reg dir = 0;

parameter speed = 1000000; //change this value for speed
parameter duration = 200; //change this value for changing direction timing

always @(posedge P_Clk)
begin
        count = count + 1;
        if (count == speed)
        begin
                count = 0;
                                P_STP = stpval;

                if (dir == 0)
                begin
                        stpval = stpval << 1;
                        if (stpval == 'b0000) stpval = 'b0001;
                end
                else
                begin
                        stpval = stpval >> 1;
```

```
                    if (stpval == 'b0000) stpval = 'b1000;
                end

                dur = dur + 1;
                if(dur == duration)
                begin
                    dur = 0;
                    dir = ~dir;
                end
        end
end

endmodule
```

Result:

Date:	Staff's Sign:

EXPERIMENT NO:4

NAME OF THE EXPERIMENTS: ADC

AIM: HDL Code to accept analog signal, temperature sensor and display the data on LCD or seven segment display.

THEORY: In electronics, an **analog-to-digital converter** (**ADC**, **A/D**, or **A-to-D**) is a system that converts an analog signal, such as a sound picked up by a microphone or light entering a digital camera, into a digital signal. An ADC may also provide an isolated measurement such as an electronic device that converts an input analog voltage or current to a digital number representing the magnitude of the voltage or current. Typically the digital output is a two's complement binary number that is proportional to the input, but there are other possibilities**.**

CODE:

```
module ADC_POT(
input P_Clk,
input [7:0] P_ADC, // ADC Data
input ADC_INT, // ADC INT
output reg ADC_WR, // ADC WR
output reg [7:0] P_LED
);

reg [4:0] count = 0;
reg await = 0;

always @(posedge P_Clk)
begin
        if (count == 0)
        begin
                        ADC_WR = 0;
                        await = 1;
        end;


        if (count == 5)
        begin
                        ADC_WR = 1;
                        await = 0;
        end

        if(await == 0)
        begin

                if(ADC_INT == 0)
                begin
                        P_LED = P_ADC;
                        await = 1;
                end
        end

        if (await == 1) count = count + 1;

        if (count == 10) count = 0;

end
endmodule
```

<u>Result:</u>

Date:	**Staff's Sign:**

EXPERIMENT NO :5

NAME OF THE EXPERIMENT: Sine, Square, Triangle, Ramp using DAC.

THEORY: In electronics, a digital-to-analog converter (DAC, D/A, D2A, or D-to-A) is a system that converts a digital signal into an analog signal. An analog-to-digital converter (ADC) performs the reverse function.

There are several DAC architectures; the suitability of a DAC for a particular application is determined by figures of merit including: resolution, maximum sampling frequency and others. Digital-to-analog conversion can degrade a signal, so a DAC should be specified that has insignificant errors in terms of the application

A). Sine waveform:

AIM: To generate sine waveform using DAC

CODE:

```
module DAC_SINE(
input P_Clk,
output reg [7:0] P_DAC, // DAC Out
output reg DAC_WR // DAC WR
);

reg [4:0] count = 0;
reg [8:0] i = 0;
reg [7:0] sine[0:255];

initial
begin
$readmemh("DAC_SINE.lst",sine);
end

always @(posedge P_Clk)
begin
        if (count == 0)
        begin
                        i = i + 1;
```

```
                    P_DAC = sine[i];
                    if(i == 255) i = 0;
        end

        if (count == 1) DAC_WR = 1;
        if (count == 3) DAC_WR = 0;

        count = count + 1;
        if (count == 10) count = 0;
end

endmodule
```

Result:

Date:	**Staff's Sign:**

B). Square waveform:

AIM: To generate square waveform using DAC

CODE:

```
module DAC_SQUARE(
input P_Clk,
output reg [7:0] P_DAC, // DAC Out
output reg DAC_WR // DAC WR
);

reg [17:0] count = 0;

always @(posedge P_Clk)
begin
        if (count == 0) P_DAC = 0;
        if (count == 2) DAC_WR = 1;
        if (count == 10) DAC_WR = 0;

        if (count == 50000) P_DAC = 'hFF;
        if (count == 50002) DAC_WR = 1;
        if (count == 50010) DAC_WR = 0;

        count = count + 1;

        if(count == 100000) count = 0;
end
endmodule
```

Result:

Date:	**Staff's Sign:**

C). Triangle waveform:

AIM: To generate triangle waveform using DAC.

CODE:

```
module DAC_TRIANGLE(
input P_Clk,
output reg [7:0] P_DAC, // DAC Out
output reg DAC_WR // DAC WR
);

reg [3:0] count = 0;
reg [7:0] i = 0;
reg dir = 0;

always @(posedge P_Clk)
begin
        if (count == 0)
        begin
                if (dir == 0)
                begin
                        i = i + 1;
                        if(i == 255) dir = 1;
                end
                else
                begin
                        i = i - 1;
                        if(i == 0) dir = 0;
                end

                P_DAC = i;
        end

        if (count == 1) DAC_WR = 1;
        if (count == 3) DAC_WR = 0;

        count = count + 1;
        if (count == 10) count = 0;
end
```

```
endmodule
```

Result:

Date:	**Staff's Sign:**

D). Ramp waveform:

AIM: To generate Ramp waveform using DAC.

CODE:

```
module DAC_SAWTOOTH(
input P_Clk,
output reg [7:0] P_DAC, // DAC Out
output reg DAC_WR // DAC WR
);

reg [3:0] count = 0;

always @(posedge P_Clk)
begin
        if (count == 0)
        begin
                P_DAC = P_DAC + 1;
                if(P_DAC == 255) P_DAC = 0;
        end


        if (count == 1) DAC_WR = 1;
        if (count == 3) DAC_WR = 0;

        count = count + 1;
        if (count == 10) count = 0;
end

endmodule
```

Result:

Date:	**Staff's Sign:**

EXPERIMENT NO:6

NAME OF THE EXPERIMENT: Elevator.

AIM: HDL Code to simulate Elevator operation.

THEORY: An elevator or lift is a type of vertical transportation that moves people or goods between floors (levels, decks) of a building, vessel, or other structure. Elevators are generally powered by electric motors that either drive traction cables and counterweight systems like a hoist, or pump hydraulic fluid to raise a cylindrical piston like a jack.

CODE:

```
Module elevator_fpga(clk,co1,row1,out1);

Input clk;

Input[3:0]co1;

Output reg[3:0]row1;

Output reg[7:0]out1;

Reg[15:0]dely=16’b0000000000000000;

Reg iclk;

Reg test;

Reg[3:0]row;

Reg[3:0]col1;

Reg[3:0]row2;

Reg[3:0]num,pnum;

Reg k;

Reg[7:0]mem[15:0];

Initial

Begin

Mem[0]=8’b00111111;

Mem[1]=8’b00000110;

Mem[2]=8’b01011011;
```

```
Mem[3]=8’b01001111;
Mem[4]=8’b01100110;
Mem[5]=8’b01101101;
Mem[6]=8’b01111101;
Mem[7]=8’b00000111;
Mem[8]=8’b01111111;
Mem[9]=8’b01101111;
Mem[10]=8’b01110111;
Mem[11]=8’b01111100;
Mem[12]=8’b01011000;
Mem[13]=8’b01011110;
Mem[14]=8’b01111001;
Mem[15]=8’b01110001;
Pnum=4’b0000;
End
always@(posedge clk)
begin
delay=delay+1;
iclk=delay[3];
k=delay[15]
end
always@(posedge clk)
begin
if(co1==4’b1110)begin test =1’b1;end
else if(col==4’b1101)begin test =1’b1;end
else if(col==4’b1011)begin test =1’b1;end
else if(col==4'b1111)begin test =1'b1;end
else test=1'b0;
end
always@(posedge test)
begin
col1=col;
row2=row;
end
always@(posedge iclk)
begin
if(row==4'b1110)begin row=4'b1101;end
else if(row==4'b1101)begin row=4'b1011;end
else if(row==4'b1011)begin row=4'b0111;end
else row=4'b1110;
row1=row;
end
always@(posedge test)
begin
if(col1==4'b1110 && row2==4'b1110)
num=0;
else if(col1==4'b1101 && row2==4'b1110)
num=1;
else if(col1==4'b1011 && row2==4'b1110)
num=2;
if(col1==4'b0111 && row2==4'b1110)
num=3;
```

```
else if(col1==4'b1110 && row2==4'b1101)
num=4;
else if(col1==4'b1101 && row2==4'b1101)
num=5;
else if(col1==4'b1011 && row2==4'b1101)
num=6;
else if(col1==4'b0111 && row2==4'b1101)
num=7;
else if(col1==4'b1110 && row2==4'b1011)
num=8;
else if(col1==4'b1101 && row2==4'b1011)
num=9;
else if(col1==4'b1011 && row2==4'b1011)
num=10;
else if(col1==4'b0111 && row2==4'b1011)
num=11;
else if(col1==4'b1110 && row2==4'b0111)
num=12;
else if(col1==4'b1101 && row2==4'b0111)
num=13;
else if(col1==4'b1011 && row2==4'b0111)
num=14;
else if(col1==4'b0111 && row2==4'b0111)
num=15;
end
always@(posedge k)
```

```
begin
if(!(num==pnum))
begin
if(num>pnum)
begin
pnum=pnum+1;
end
else if(num<pnum)
begin
pnum=pnum-1;
end
end
end
always@(pnum)
begin
out1<=mem[pnum];
end
endmodule
```

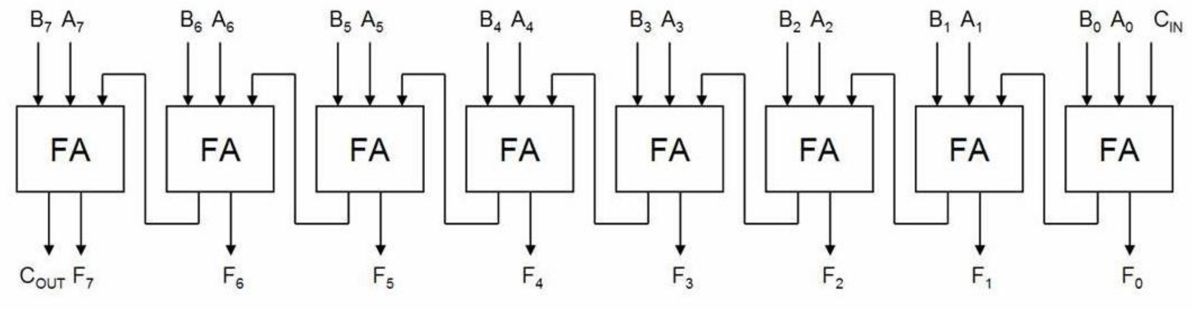
Result:

Date:	**Staff's Sign:**

Beyond Syllabus: 8-BIT RIPPLE CARRY ADDER

Aim: To design and verify the working of an 8-bit ripple carry adder.

Block Diagram:



Theory:

Ripple carry adder can be created by cascading multiple full adders together. Each full adder inputs Cin, which is the Cout of the previous adder. This kind of adder is a Ripple Carry Adder, since each carry bit "ripples" to the next full adder. The first (and only the first) full adder may be replaced by a half adder. The block diagram of 8-bit Ripple Carry Adder is shown above. The corresponding Boolean expressions given here are to construct a ripple carry adder. In the half adder circuit the sum and carry bits are defined as,

$$\text{Sum} = A \oplus B$$

$$\text{Carry} = AB$$

In the full adder circuit the Sum and Carry output is defined by inputs A, B and Carry in (C) as

$$\text{Sum}=ABC + ABC + ABC + ABC$$

$$\text{Sum}= ABC + ABC + ABC + ABC$$

$$= (AB + AB)\,C + (AB + AB)\,C$$

$$= (A \oplus B)\,C + (A \oplus B)\,C$$

$$=A \oplus B \oplus C$$

$$\text{Carry}= ABC + ABC + ABC + ABC$$

$$= AB + (AB + AB)\,C$$

$$= AB + (A \oplus B)\,C$$

Verilog Code:

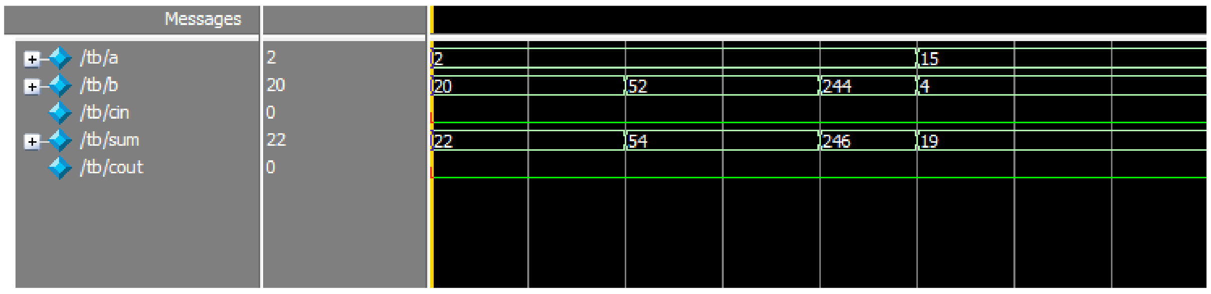
```
module cra(a,b,cin,sum,cout);
input [7:0] a,b;
input cin;
output [7:0] sum;
output cout;
wire c1,c2,c3,c4,c5,c6,c7;
assign sum[0] = a[0] ^ b[0] ^ cin;
assign c1 = a[0] & b[0] | b[0] & cin | a[0] & cin;
assign sum[1] = a[1] ^ b[1] ^ c1;
assign c2 = a[1] & b[1] | b[1] & c1 | a[1] & c1;
assign sum[2] = a[2] ^ b[2] ^ c2;
assign c3 = a[2] & b[2] | b[2] & c2 | a[2] & c2;
assign sum[3] = a[3] ^ b[3] ^ c3;
assign c4 = a[3] & b[3] | b[3] & c3 | a[3] & c3;
assign sum[4] = a[4] ^ b[4] ^ c4;
assign c5 = a[4] & b[4] | b[4] & c4 | a[4] & c4;
assign sum[5] = a[5] ^ b[5] ^ c5;
assign c6 = a[5] & b[5] | b[5] & c5 | a[5] & c5;
assign sum[6] = a[6] ^ b[6] ^ c6;
assign c7 = a[6] & b[6] | b[6] & c6 | a[6] & c6;
assign sum[7] = a[7] ^ b[7] ^ c7;
assign cout = a[7] & b[7] | b[7] & c7 | a[7] & c7;
endmodule
```

Test Bench:

```
module tb;
reg [7:0] a,b;
reg cin;
wire [7:0] sum;
wire cout;
cra R1 (a,b,cin,sum,cout);
initial
begin
a=8'b0000_0010;
b=8'b0001_0100;
cin=0;
#100;
a=8'b0000_0010;
```

```
b=8'b0001_0100;
cin=0;
#100;
a=8'b0000_0010;
b=8'b0011_0100;
cin=0;
#100;
a=8'b0000_0010;
b=8'b0011_0100;
cin=0;
#100;
a=8'b0000_0010;
b=8'b1111_0100;
cin=0;
#100;
a=8'b0000_1111;
b=8'b0000_0100;
cin=0;
#100;
end
endmodule
```

Simulation results:



Date:	Staff's Sign:

PIN PORTS:

```
## Matrix KeyPad IN
#NET "MK_IN[0]" LOC = P62;
#NET "MK_IN[1]" LOC = P61;
#NET "MK_IN[2]" LOC = P58;
#NET "MK_IN[3]" LOC = P57;
#
## Matrix KeyPad OUT
#NET  "MK_OUT[0]"  LOC  =  P51;  #NET
"MK_OUT[1]"     LOC     =     P50;     #NET
"MK_OUT[2]"     LOC     =     P48;     #NET
"MK_OUT[3]" LOC = P46;
#
## 7 SEG BUS
#NET "P_7seg[0]" LOC = P97;
#NET "P_7seg[1]" LOC = P95;
#NET "P_7seg[2]" LOC = P94;
#NET "P_7seg[3]" LOC = P93;
#NET "P_7seg[4]" LOC = P92;
#NET "P_7seg[5]" LOC = P88;
#NET "P_7seg[6]" LOC = P87;
#NET "P_7seg[7]" LOC = P85;
#
## 7 SEG Digit Select Lines
#NET "P_dig[0]" LOC = P6;
#NET "P_dig[1]" LOC = P5;
#NET "P_dig[2]" LOC = P2;
#NET "P_dig[3]" LOC = P1;
#
## LED
#NET "P_LED(0)"    LOC = P98;
#NET "P_LED(1)"LOC = P99;
#NET "P_LED(2)"    LOC = P100;
#NET "P_LED(3)"    LOC = P101;
#NET "P_LED(4)"    LOC = P102;
#NET "P_LED(5)"    LOC = P104;
```

```
#NET "P_LED(6)"   LOC = P105;
#NET "P_LED(7)"   LOC = P111;
#NET "P_LED(8)"   LOC = P112;
#NET "P_LED(9)"   LOC = P114;
#NET "P_LED(10)" LOC = P115;
#NET "P_LED(11)" LOC = P116;
#NET "P_LED(12)" LOC = P117;
#NET "P_LED(13)" LOC = P118;
#NET "P_LED(14)" LOC = P119;
#NET "P_LED(15)" LOC = P120;
#
## ADC
#NET  "P_ADC(0)"  LOC  =  P12;
#NET  "P_ADC(1)"  LOC  =  P14;
#NET  "P_ADC(2)"  LOC  =  P15;
#NET  "P_ADC(3)"  LOC  =  P16;
#NET  "P_ADC(4)"  LOC  =  P17;
#NET  "P_ADC(5)"  LOC  =  P21;
#NET  "P_ADC(6)"  LOC  =  P22;
#NET  "P_ADC(7)"  LOC  =  P23;
#NET  "ADC_WR"  LOC  =  P24;
#NET "ADC_INT" LOC = P26;

# DAC
#NET  "P_DAC(0)"  LOC  =  P80;
#NET  "P_DAC(1)"  LOC  =  P79;
#NET  "P_DAC(2)"  LOC  =  P78;
#NET  "P_DAC(3)"  LOC  =  P75;
#NET "P_DAC(4)"   LOC = P74;
#NET "P_DAC(5)"   LOC = P45; #J5 conn Pin 1
#NET "P_DAC(6)"   LOC = P67;
#NET "P_DAC(7)"   LOC = P66;
#NET "DAC_WR"     LOC = P81;

# LCD
#NET "LCD_RS"     LOC = P7;
#NET "LCD_EN"     LOC = P8;
#NET "P_LCD(0)"   LOC = P97;
#NET "P_LCD(1)"   LOC = P95;
```

```
#NET "P_LCD(2)"    LOC = P94;
#NET  "P_LCD(3)"  LOC  =  P93;
#NET  "P_LCD(4)"  LOC  =  P92;
#NET  "P_LCD(5)"  LOC  =  P88;
#NET  "P_LCD(6)"  LOC  =  P87;
#NET "P_LCD(7)" LOC = P85;

#STEPPER MOTOR
#NET  "P_STP(0)"  LOC  =  P29;
#NET  "P_STP(1)"  LOC  =  P30;
#NET  "P_STP(2)"  LOC  =  P32;
#NET "P_STP(3)" LOC = P33;

#DC MOTOR
NET  "P_DCM(0)"  LOC  =  P82;
NET  "P_DCM(1)"  LOC  =  P83;
NET "P_DCMEN" LOC = P84;

# Main Clock Line
NET "P_Clk" LOC = P56;

#NET "sw1" LOC = P121 ;
#NET "sw2" LOC = P123 ;
#NET "led2" LOC = P120 ;
```

VIVA QUESTIONS:

1. Expand VHDL.What is the difference between VHDL and Verilog?
2. What is the difference between (i) signal and variable (ii) generic & Parameter (iii) function & procedure (iv) task & function (v) always & initial (vi) register & variable (vii) signal & wire
3. What are the different styles of models in VHDL and Verilog?
4. What are the operators in VHDL & Verilog?
5. Which is an operator is having most priority?
6. What is meant by sensitivity list?
7. Give the Following syntax in HDL (i) if, for, function, procedure (ii) while, case.
8. What is the operating frequencyofyour FPGA?
9. Expand FPGA &ASICWhat are the data types in VHDL?
10. What are the data types in Verilog?What is delta time?
11. What is the difference between (0 to 3) & (3 downto 0)?
12. Write the truth table & Excitation table for D filp flop, SR , T, JK
13. What are the file operations in Verilog?
14. What is meant by synthesis?
15. Write the flow chart for synthesis process?
16. What is the difference between combination circuit & sequential circuit?
17. What is the difference between latch & Flip flop?Write a Verilog code to swap contents of two registers with and without a temporary register?
18. In a pure combinational circuit is it necessary to mention all the inputs in sensitivity list? If yes, why?What is the difference between wire and reg?
19. Give only two xor gates one must function as buffer and another as not gate?
20. Build a 4:1 mux using only 2:1 mux?What are shift operators in HDL?
21. What are the logical operators in VHDL & Verilog?
22. What is the gate density of your FPGA?
23. What is data flow model, structural model, behavioral model?How you invoke from VHDL to Verilog and vice versa?
24. What is the difference between SR flip flop & JK flip flop?
25. What is the difference between synchronous reset & Asynchronous reset?
26. What is the difference between stepper motor & DC motor?
27. What is the step size of stepper motor?
28. What is mux and demux?
29. What is encoder and decoder?
30. What is the difference between encoder & priority encoder?
31. What is binding?
32. What is the difference between “bit” and “std_logic”?

33. What are the std_logic values?
34. What are the different types of buffers are in Verilog HDL?
35. What is the difference between dc motor and stepper motor?
36. What are the applications of dc motor and stepper motor?
37. Write the syntax for casex and casez.What is screen time?
38. Draw the simulation waveform for D-latch using signal assignment and variable assignment statements inside the process.
39. What is SRAM?
40. What is mealy model and Moore model?
41. What are user defined types?
42. What are the packages are available in VHDL? And also give the syntax for package
43. How to call procedure and function within the process?
44. Give the syntax for arrays in VHDL and Verilog.
45. What are the VHDL file processing?