LOGIC DESIGN LAB (ECL 203)

Laboratory Manual

PART-II

BTech Semester II

SREE BUDDHA COLLEGE OF ENGINEERING

PATTOR P.O, NOORNAD, ALAPPUZHA, KERALA -690529

Department

Of

Electronics and Communication Engineering

2020-21

TABLE OF CONTENT

Course Objectives:

CO 1	Design and demonstrate the functioning of various combinational and sequential circuits using ICs
CO 2	Apply an industry compatible hardware description language to implement digital circuits
CO 3	Implement digital circuits on FPGA boards and connect external hardware to theboards
CO 4	Function effectively as an individual and in a team to accomplish the given task

Assessment

Mark distribution

Total Marks	CIE	ESE	ESE Duration
150	75	75	2.5 hours

Continuous Internal Evaluation Pattern:

Attendance :15 marks

Continuous Assessment: 30 marks

End Semester Examination Pattern: The following guidelines should be followed regarding awardof marks

(a) Preliminary work	: 15 Marks
(b) Implementing the work/Conducting the experiment	: 10 Marks
(c) Performance, result and inference (usage of equipment and troubleshooting)	: 25 Marks
(d) Viva voce	: 20 marks
(e) Record	: 5 Marks

General instructions: End-semester practical examination is to be conducted immediately after the second series test covering entire syllabus given below. Evaluation is to be conducted under the equal responsibility of both the internal and external examiners. The number of candidates evaluated per day should not exceed 20. Students shall be allowed for the examination only on submitting the duly certified record. The external examiner shall endorse the record.

OVERVIEW OF HDL LAB

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 behaviour 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 hierarchies 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 executes ability that gives HDLs the illusion of being programming languages. Simulators capable of supporting discrete-event and continuous-time (analog) modelling 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 behaviour. 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

1. VHDL (VHSIC HDL)
2. Verilog

VHDL

VHDL (Very High-Speed Integrated Circuit Hardware Description Language) is commonly used as a design-entry language for field-programmable gate arrays and application- specific integrated circuits in electronic design automation of digital circuits. VHDL is a fairly general-purpose language, and it doesn’t require a simulator on which to run the code. There are a lot of VHDL compilers, which build executable binaries. It can read and write files on the host computer, so a VHDL program can be written that generates another VHDL program to be incorporated in the design being developed. Because of this general- purpose nature, it is possible to use VHDL to write a test bench that verifies with the user, and compares results with those expected. This is similar to the capabilities of the Verilog Language.

VHDL is not a case sensitive language. One can design hardware in a VHDL IDE (such as Xilinx) to produce the RTL schematic of the desired circuit. After that, the generated schematic can be verified using simulation software (such as ModelSim) which shows the waveforms of inputs and outputs of the circuit after generating the appropriate test bench. To generate an appropriate test bench for a particular circuit or VHDL code, the inputs have to be defined correctly. For example, for clock input, a loop process or an iterative statement is required.

The key advantage of VHDL when used for systems design is that it allows the behaviour of the required system to be described (modelled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires). When a VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device such as a CPLD or FPGA, then it is the actual hardware being configured, rather than the VHDL code being "executed" as if on some form of a processor chip.

Both VHDL and Verilog emerged as the dominant HDLs in the electronics industry while older and less-capable HDLs gradually disappeared from use. But VHDL and Verilog share many of the same limitations: neither HDL is suitable for analog/mixed-signal circuit simulation. Neither possesses language constructs to describe recursively-generated logic structures.

Verilog

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 behaviour 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).

VHDL vs. Verilog

VHDL	**Verilog**
Strongly typed	Weakly typed
Easier to understand	Less code to write
More natural in use	More of a hardware modelling language
Wordy	Succinct
Non-C-like syntax	Similarities to the C language
Variables must be described by data type	A lower level of programming constructs
Widely used for FPGAs and military	A better grasp on hardware modelling
More difficult to learn	Simpler to learn

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.

HDL Programming using Xilinx ISE design suite

Xilinx ISE means Xilinx® Integrated Software Environment (ISE), i.e programmable logic design tool in electronics industry. This Xilinx ® design software suite allows taking design from design entry through Xilinx device programming. The ISE Project Navigator manages and processes design through several steps in the ISE design flow. These steps are Design Entry, Synthesis, Implementation, Simulation/Verification, and Device Configuration. Xilinx is one of most popular software tools used to synthesize VHDL/Verilog code.

Operators In Verilog

a) Arithmetic Operators

These operators are performing arithmetic operations. The + and −are used as either unary (x) or binary (z−y) operators.

The Operators which are included in arithmetic operation are −

+ (addition), −(subtraction), * (multiplication), / (division), % (modulus)

b) Relational Operators

These operators compare two operands and return the result in a single bit, 1 or 0.

Wire and reg variables are positive. Thus (−3’d001) = = 3’d111 and (−3b001)>3b110.

The Operators which are included in relational operation are −

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

c) Bit-wise Operators

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

The Operators which are included in Bit wise operation are −

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

d) Logical Operators

Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1. They can work on integers or group of bits, expressions and treat all non-zero values as 1. Logical operators are generally, used in conditional statements since they work with expressions.

The operators which are included in Logical operation are −

- ! (logical NOT)
- && (logical AND)
- || (logical OR)

e) Reduction Operators

Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value.

The operators which are included in Reduction operation are −

- •& (reduction AND)
- •| (reduction OR)
- •~& (reduction NAND)
- •~| (reduction NOR)
- •^ (reduction XOR)
- •~^ or ^~ (reduction XNOR)

f) Shift Operators

Shift operators, which are shifting the first operand by the number of bits specified by second operand in the syntax. Vacant positions are filled with zeros for both directions, left and right shifts (There is no use sign extension).

The Operators which are included in Shift operation are −

- •<< (shift left)
- •>> (shift right)

g) Concatenation Operator

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

The operator included in Concatenation operation is − { } (concatenation)

h) Replication Operator

The replication operators are making multiple copies of an item.

The operator used in Replication operation is − {n {item}} (n fold replication of an item)

i) Conditional Operator

Conditional operator synthesizes to a multiplexer. It is the same kind as is used in C/C++ and evaluates one of the two expressions based on the condition.

The operator used in Conditional operation is −

(Condition)? (Result if condition true) −

(Result if condition false)

j) Wires, Regs, and Parameters

Wires, regs and parameters are the data types used as operands in Verilog expressions

Bit-Selection "x [2]" and Part-Selection "x [4:2]"

Bit-selects and part-selects are used to select one bit and multiple bits, respectively, from a wire, reg or parameter vector with the use of square brackets “[ ]”. Bit-selects and part-selects are also used as operands in expressions in the same way that their main data objects are used.

INTRODUCTION TO FPGA (FIELD PROGRAMMABLE GATE ARRAY)

What is FPGA?

Field-programmable gate array (FPGA) is a device that has array of **Configurable logic blocks (CLB)** and can be programmed on-board through dedicated Joint Test Action Group (JTAG) or through any other serial/ parallel non-volatile Memory. FPGA architecture is based on static random-access memory (SRAM) Volatile memory. The Data programmed inside the memory of an FPGA erase, once the board is powered off. In order to configure the data, external EEPROM is attached to FPGA.

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 ARCHITECTURE OF AN FPGA

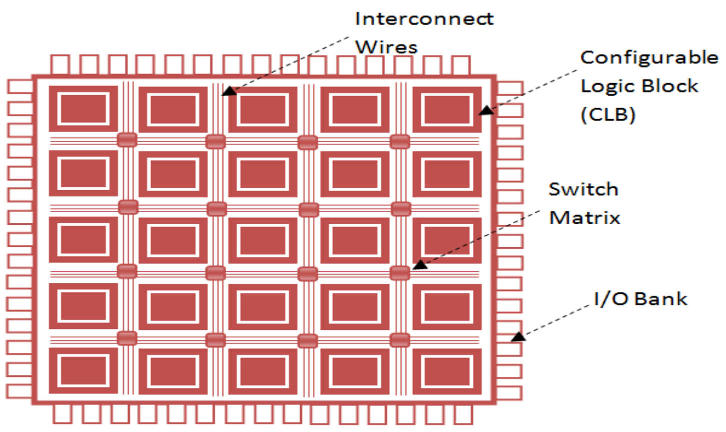
FPGAs can be used to implement an entire System On one Chip (SOC). The FPGA has the advantage of **re-programmable hardware architecture**. In case of microprocessor, it is not possible to implement re-configurable Hardware. FPGA Architecture Features

FPGA Architecture consist of the following features

Configurable logic Block (CLB)

Input/output Block

Switching Matrix Interconnects



In FPGA, each Configurable Logic Block consists of 2 slices. Those slices are further divided in 2 logic elements.

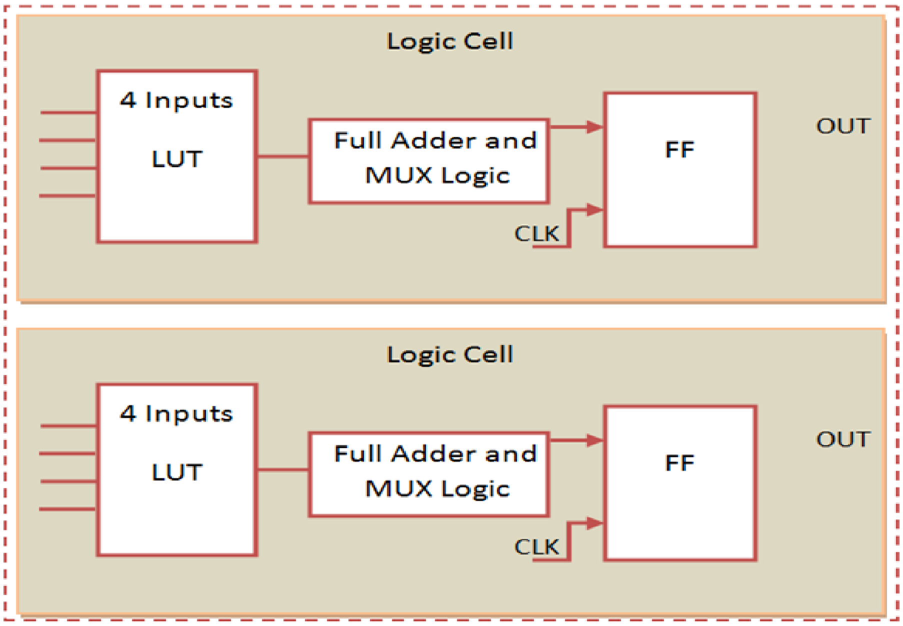
Logic elements consist of

•4 input lookup Table

•Full Adder and Mux logic

•D Flip-flop

Configurable Logic Block



4 input Lookup table LUT is used to implement one of the following functionalities

•Combinational Logic design

•Distributed RAM

•Shift Register

Also, there are various dedicated circuits are present inside FPGA. They are Digital Clock Manager, Multiplier, and Block RAM and so on.

Digital Clock Manager DCM is used to perform Clock Phase shift, De skew, Clock divider and frequency synthesis.

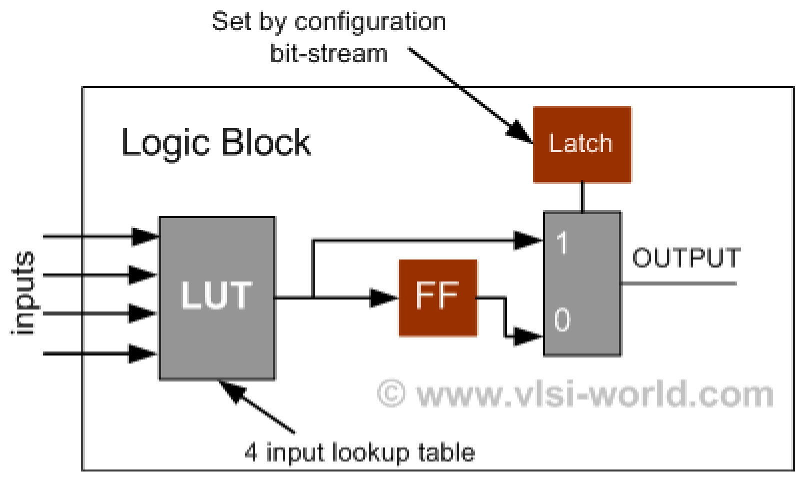
Multiplier Block implement dedicated 18×18 multiplier with Signed and unsigned operation.

Block RAM is a dedicated memory implement dual port 16kb memory.

XILINX FPGA

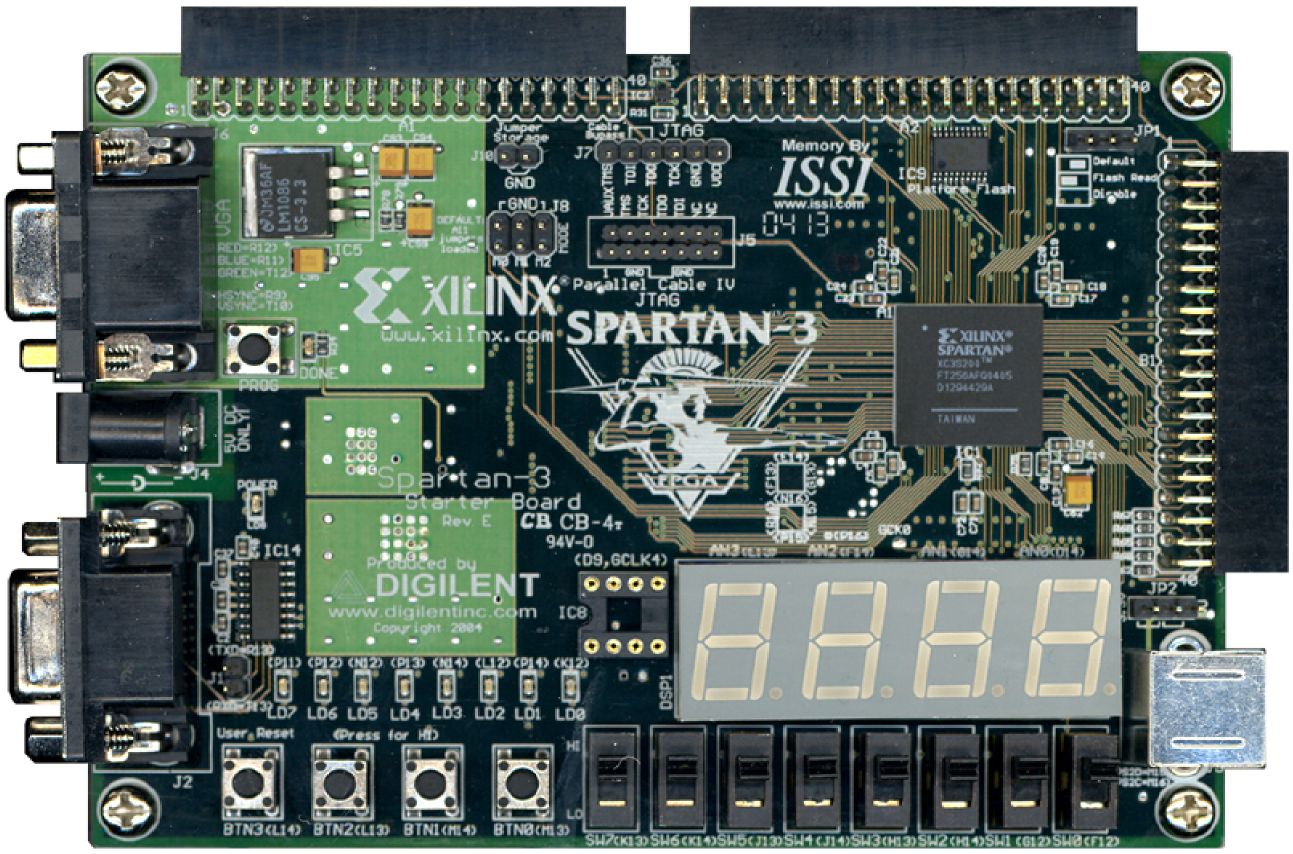
Xilinx logic block consists of one Look up Table (LUT) and one Flip-flop. An LUT is used to implement number of different functionalities. The input lines to the logic block go into the LUT and enable it. The output of the LUT gives the result of the logic function that it implements and the unregistered output from output of logic block is registered or the LUT.Lookup table is implemented using SRAM. A k-input logic function is implemented using 2^k * 1 size SRAM. Number of different possible functions for k input LUT is 2^2^k. Advantage of such an architecture is that it supports implementation of so many logic functions, however the disadvantage is unusually large number of memory cells required to implement such a logic block in case number of inputs is large.

4-INPUT LUT BASED IMPLEMENTATION OF LOGIC BLOCK.



XC3S200A-4FT256C FPGA Board

The Spartan®-3 family of Field-Programmable Gate Arrays is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The eight-member family offers densities ranging from 50,000 to 5,000,000 system gates.
The Spartan-3 family builds on the success of the earlier Spartan-IIE family by increasing the amount of logic resources, the capacity of internal RAM, the total number of I/Os, and the overall level of performance as well as by improving clock management functions. Numerous enhancements derive from the Virtex®-II platform technology. These Spartan-3 FPGA enhancements, combined with advanced process technology, deliver more functionality and bandwidth per dollar than was previously possible, setting new standards in the programmable logic industry.
Because of their exceptionally low cost, Spartan-3 FPGAs are ideally suited to a wide range of consumer electronics applications, including broadband access, home networking, display/projection and digital television equipment.
The Spartan-3 family is a superior alternative to mask programmed ASICs. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs.
It is compatible with the free ISE WebPackTM, so designs can be completed at no extra cost.



Feature

• Low-cost, high-performance logic solution for high-volume, consumer-oriented applications
• Densities up to 74,880 logic cells
• SelectIO™ interface signalling
• Up to 633 I/O pins
• 622+ Mb/s data transfer rate per I/O
• 18 single-ended signal standards
• 8 differential I/O standards including LVDS, RSDS
• Termination by Digitally Controlled Impedance
• Signal swing ranging from 1.14V to 3.465V
• Double Data Rate (DDR) support
• DDR, DDR2 SDRAM support up to 333 Mb/s
• Logic resources
• Abundant logic cells with shift register capability
• Wide, fast multiplexers
• Fast look-ahead carry logic
• Dedicated 18 x 18 multipliers
• JTAG logic compatible with IEEE 1149.1/1532
• SelectRAM™ hierarchical memory
• Up to 1,872 Kbits of total block RAM
• Up to 520 Kbits of total distributed RAM
• Digital Clock Manager (up to four DCMs)
• Clock skew elimination

• Frequency synthesis
• High resolution phase shifting

Slide Switches

The Spartan-3 Starter Kit board has eight slide switches. These switches are labelled SW7 through SW0. Switch SW7 is the left-most switch, and SW0 is the right- most switch. The switches connect to an associated FPGA pin, as shown below.

Switch	SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
FPGA Pin	K13	K14	J13	J14	H13	H14	G12	F12

Push Button Switches

The Spartan-3 Starter Kit board has four momentary-contact push button switches. These push buttons are located along the lower edge of the board, toward the right edge. The switches are labelled BTN3 through BTN0. Push button switch BTN3 is the left-most switch, BTN0 the right-most switch, as shown below.

Push Button	BTN3 (User Reset)	BTN2	BTN1	BTN0
FPGA Pin	L14	L13	M14	M13

LEDs

The Spartan-3 Starter Kit board has eight individual surface-mount LEDs located above the push button switches. The LEDs are labelled LED7 through LED0. LED7 is the left-most LED, LED0 the right-most LED.

LED	LD7	LD6	LD5	LD4	LD3	LD2	LD1	LD0
FPGA Pin	P11	P12	N12	P13	N14	L12	P14	K12

FOUR-DIGIT, SEVEN-SEGMENT LED DISPLAY

FPGA Connections to Seven-Segment Display (Active Low)

Segment	FPGA Pin
A	E14
B	G13
C	N15
D	P15
E	R16
F	F13

G	N16
DP	P16

Digit Enable (Anode Control) Signals (Active Low)

Anode Control	AN3	AN2	AN1	AN0
FPGA Pin	E13	F14	G14	D14

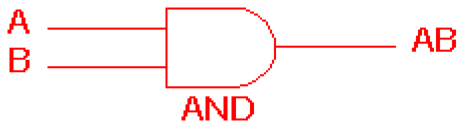
Display Characters and Resulting LED Segment Control Values

Characte r	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1
2	0	0	1	0	0	1	0
3	0	0	0	0	1	1	0
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	0
6	0	1	0	0	0	0	0
7	0	0	0	1	1	1	1
8	0	0	0	0	0	0	0
9	0	0	0	0	1	0	0
A	0	0	0	1	0	0	0
b	1	1	0	0	0	0	0
C	0	1	1	0	0	0	1
d	1	0	0	0	0	1	0
E	0	1	1	0	0	0	0
F	0	1	1	1	0	0	0

EXPERIMENT-1: Familiarization of FPGAs and Realization of Logic Gates

Aim2: The purpose of this experiment is to simulate the behaviour of basic logic gate so that several logic gates can be connected together to create a simple digital model.

AND Gate



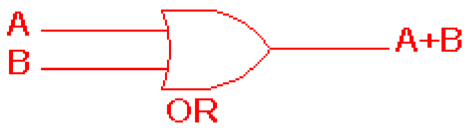
2 Input AND gate		
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

The AND gate is an electronic circuit that gives a high output (1) only if all its inputs are high. A dot (.) is used to show the AND operation i.e., A.B. Bear in mind that this dot is sometimes omitted i.e., AB.

Structural Model	**Data Flow Model**	**BehaviouralModel**	**TEST BENCH:**

module andstr(x,y,z);
input x,y;
output z;
and g1(z,x,y);
endmodule
	module anddf(x,y,z);
	input x,y;
	output z;
	assign z=(x&y);
	endmodule
		module andbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x,y)
		z=x&y;
		endmodule
			module anddf_tb;
			reg x,y;
			wire z;
			anddf uut (.x(x), y(y), z(z));
			initial begin
			x=0; y=0;#5;
			x=0; y=1;#5;
			x=1; y=0;#5;
			x=1; y=1;#5;
			end
			endmodule

OR Gate



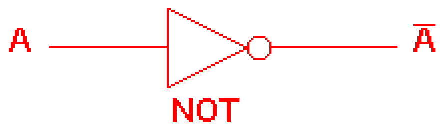
2 Input OR gate		
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.

Structural Model	**Data Flow Model**	**BehaviouralModel**	**TEST BENCH :**

module orstr(x,y,z);
inputx,y;
output z;
or g1(z,x,y);
endmodule
	module ordf(x,y,z);
	inputx,y;
	output z;
	assign z=(x|y);
	endmodule
		module orbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x,y)
		z=x|y;
		endmodule
			module ordf_tb;
			reg x,y;
			wire z;
			ordf uut (.x(x),.y(y),.z(z));
			initial begin
			x=0; y=0;#5;
			x=0; y=1;#5;
			x=1; y=0;#5;
			x=1; y=1;#5;
			end
			endmodule

NOT Gate



NOT gate	
A	$\overline{A}$
0	1
1	0

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top,

Structural Model

```
module notstr(x,z);
input x;
output z;
not g1(z,x);
endmodule
```

Data Flow Model

```
module notdf(x,z);
input x;
output z;
assign z= !x;
endmodule
```

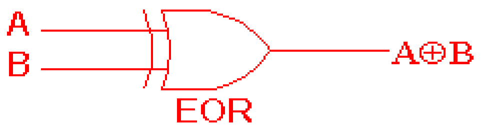
BehaviouralModel

```
module notbeh(x,z);
input x;
output z;
reg z;
always @(x)
z=!x;
endmodule
```

TTEST BENCH:

```
module notdf_tb;
reg x;
wire z;
notdf uut (.x(x),.z(z));
initial begin
x=0; #5;
x=1; #5;
end
endmodule
```

EXOR Gate



2 Input EXOR gate		
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Structural Model

```
module xorstr(x,y,z);
inputx,y;
output z;
xor g1(z,x,y);
endmodule
```

Data Flow Model

```
module xordf(x,y,z);
inputx,y;
output z;
assign z=(x^y);
endmodule
```

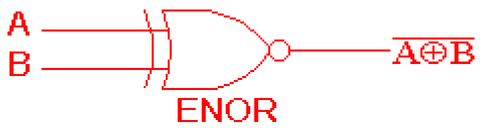
Behavioural Model

```
module xorbeh(x,y,z);
input x,y;
output z;
reg z;
always @(x,y)
z=x^y;
endmodule
```

TEST BENCH:

```
module xordf_tb;
reg x,y;
wire z;
xordf uut (.x(x),.y(y),.z(z));
initial begin
x=0; y=0;#5;
 x=0; y=1;#5;
 x=1; y=0;#5;
 x=1; y=1;#5;
end
endmodule
```

EXNOR Gate

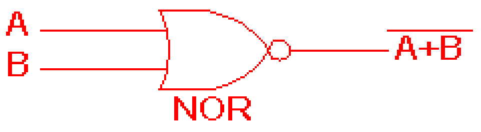


2 Input EXNOR gate		
A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

The 'Exclusive-NOR' gate circuit does the opposite to the EXOR gate. It will give a low output if either, but not both, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

Structural Model	**Data Flow Model**	**BehaviouralModel**	**TEST BENCH :**
module xnorstr(x,y,z);	module xnordf(x,y,z);	module xnorbeh(x,y,z);	module xnordf_tb;
inputx,y;	inputx,y;	input x,y;	reg x,y;
output z;	output z;	output z;	wire z;
xnor g1(z,x,y);	assign z= !(x^y);	reg z;	xnordf uut (.x(x),.y(y),.z(z));
endmodule	endmodule	always @(x,y)	initial begin
		z=!(x^y);	x=0; y=0;#5;
		endmodule	x=0; y=1;#5;
			x=1; y=0;#5;
			x=1; y=1;#5;
			end
			endmodule

NOR Gate

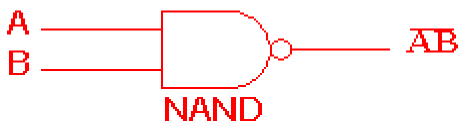


Input		Output
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

The output is active high only if both the inputs are in active low state.

Structural Model	**Data Flow Model**	**BehaviouralModel**	**TEST BENCH:**
module norstr(x,y,z);	module nordf(x,y,z);	module norbeh(x,y,z);	module nordf_tb;
inputx,y;	inputx,y;	input x,y;	reg x,y;
output z;	output z;	output z;	wire z;
nor g1(z,x,y);	assign z= !(x|y);	reg z;	nordf uut (.x(x),.y(y),.z(z));
endmodule	endmodule	always @(x,y)	initial begin
		z=!(x|y);	x=0; y=0;#5;
		endmodule	x=0; y=1;#5;
			x=1; y=0;#5;
			x=1; y=1;#5;
			end
			endmodule

NAND Gate



Input		Output
A	B	Y= $\overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

The output is active high only if any one of the input is in active low state.

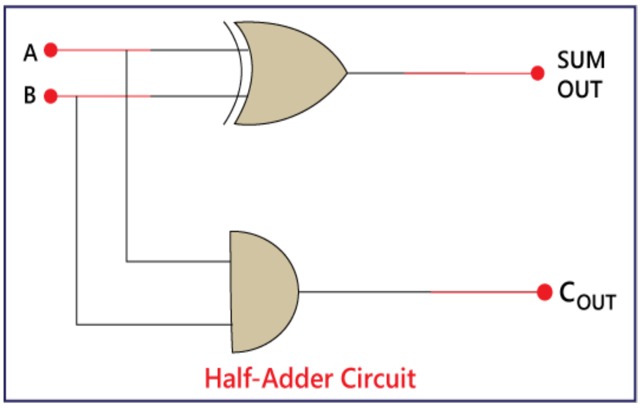
Structural Model	**Data Flow Model**	**BehaviouralModel**	**TEST BENCH:**
modulenandstr(x,y,z);	module nanddf(x,y,z);	module nandbeh(x,y,z);	module nanddf_tb;
inputx,y;	inputx,y;	input x,y;	reg x,y;
output z;	output z;	output z;	wire z;
nand g1(z,x,y);	assign z= !(x&y);	reg z;	ordf uut (.x(x),.y(y),.z(z));
endmodule	endmodule	always @(x,y)	initial begin
		z=!(x&y);	x=0; y=0;#5;
		endmodule	x=0; y=1;#5;
			x=1; y=0;#5;
			x=1; y=1;#5;
			end
			endmodule

EXPERIMENT-2: Adders in Verilog

Aim1: Development of Verilog modules for half adder in 3 modelling styles (dataflow/structural/ behavioural).

Half adder:

The Half-Adder is a basic building block of adding two numbers as two inputs and produce out two outputs. The adder is used to perform OR operation of two single bit binary numbers. The augment and addend bits are two input states, and 'carry' and 'sum 'are two output states of the half adder.



Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Sum= A XOR B (A+B)

Carry= A AND B (A.B)

Dataflow Modelling of Half Adder:

```
//half adder dataflow
module halfadd1(x,y,sum,carry);
input x,y;
output sum;
output carry;
assign sum = x ^ y;
assign carry = x & y;
endmodule
```

Structural Modelling of Half Adder:

```
//halfadder using structural model
module halfadd1(x,y,sum,carry);
input x,y;
output sum;
output carry;
xor(sum,x,y);
```

```
and(carry,x,y);
endmodule
```

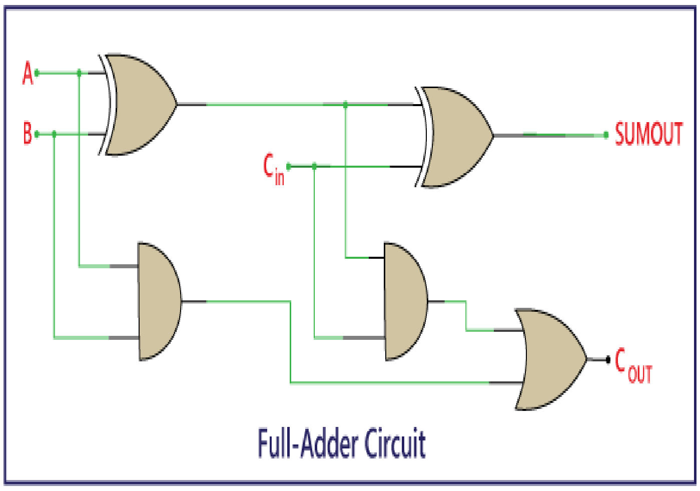
Behavioural Modelling of Half Adder:

```
//Half adder behavioral modelling
module halfadd1(x,y,sum,carry);
input x,y;
output reg sum;
output reg carry;
always @(x or y) begin
if (x == 1'b1 && y == 1'b1) begin
sum=1'b0;
carry =1'b1;
end
else if (x == 1'b0 && y == 1'b0) begin
sum=1'b0;
carry =1'b0;
end
else begin
sum=1'b1;
carry =1'b0;
end
end
endmodule
```

Aim 2: Development of Verilog modules for full adder in dataflow modelling using half adder.

Full Adder:

The half adder is used to add only two numbers. To overcome this problem, the full adder was developed. The full adder is used to add three 1-bit binary numbers A, B, and carry C. The full adder has three input states and two output states i.e., sum and carry.



Inputs			Outputs	
A	B	C_{in}	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

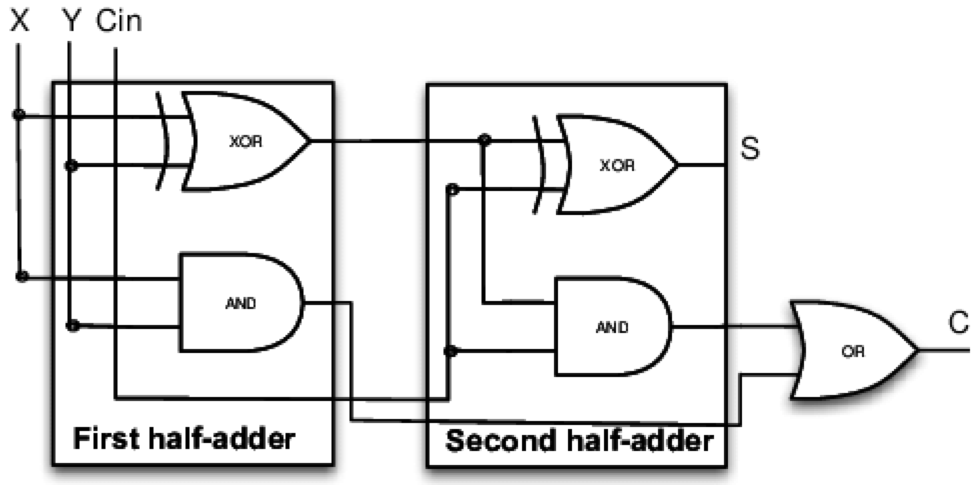
Sum:

- Perform the XOR operation of input A and B.
- Perform the XOR operation of the outcome with carry. So, the sum is (A XOR B) XOR Cin which is also represented as: (A ⊕ B) ⊕ Cin.

Carry:

- Perform the 'AND' operation of input A and B.
- Perform the 'XOR' operation of input A and B.
- Perform the 'OR' operations of both the outputs that come from the previous two steps. So, the 'Carry' can be represented as: A.B + (A ⊕ B).

Dataflow Modelling of Full Adder using two half adders:



```
//fulladder using two half adders using dataflow model
//half adder module
module halfadd1(x,y,sum,carry);
input x,y;
output sum;
output carry;
assign sum = x ^ y;
assign carry = x & y;
endmodule
//fulladder
module fulladd1 (x,y,cin,sumout,carryout);
input x,y,cin;
output sumout;
output carryout;
//Internal connections
```

```
wire a;
wire b;
wire c;
wire d;
wire e;
//Instantiate the half adder 1
halfadd1 ha1 (.x(x),.y(y),.sum(a),.carry(b));
//Instantiate the half adder 2
halfadd1 ha2 (.x(a),.y(cin),.sum(c),.carry(d));
assign sumout= c;
assign carryout = d | b;
endmodule
```

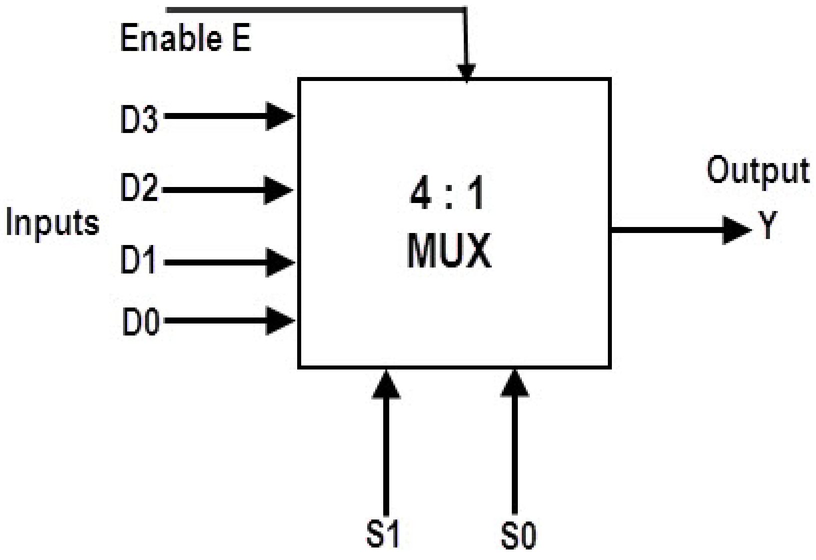
EXPERIMENT-3: Mux and Demux in Verilog

Aim1: Development of Verilog modules for a 4x1 MUX.

4-to-1 Multiplexer

A 4-to-1 multiplexer consists four data input lines as D0 to D3, two select lines as S0 and S1 and a single output line Y. The select lines S1 and S2 select one of the four input lines to connect the output line. The particular input combination on select lines selects one of input (D0 through D3) to the output.

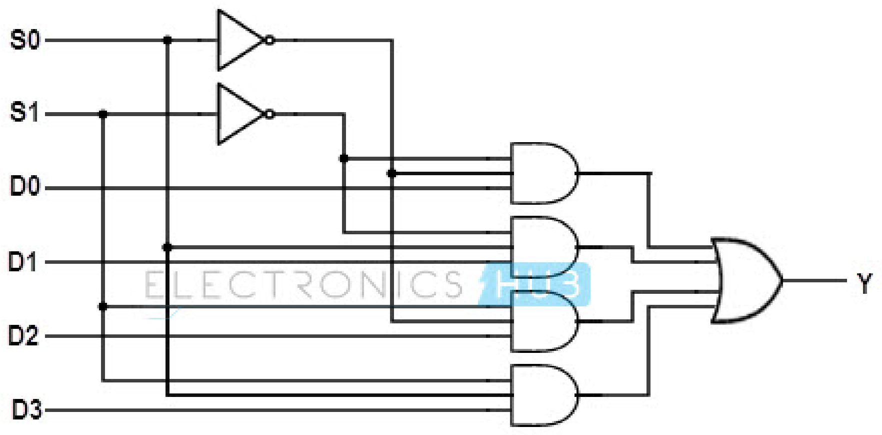
The figure below shows the block diagram of a 4-to-1 multiplexer in which the multiplexer decodes the input through select line.



Select Data Inputs		Output
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

$$Y = D0\,\overline{S1}\,\overline{S0} + D1\,\overline{S1}\,S0 + D2\,S1\,\overline{S0} + D3\,S1\,S0$$

From the above expression of the output, a 4-to-1 multiplexer can be implemented by using basic logic gates. The below figure shows the logic circuit of 4:1 MUX which is implemented by four 3-inputs AND gates, two 1-input NOT gates, and one 4-inputs OR gate.



Dataflow Modelling of 4:1 Multiplexer:

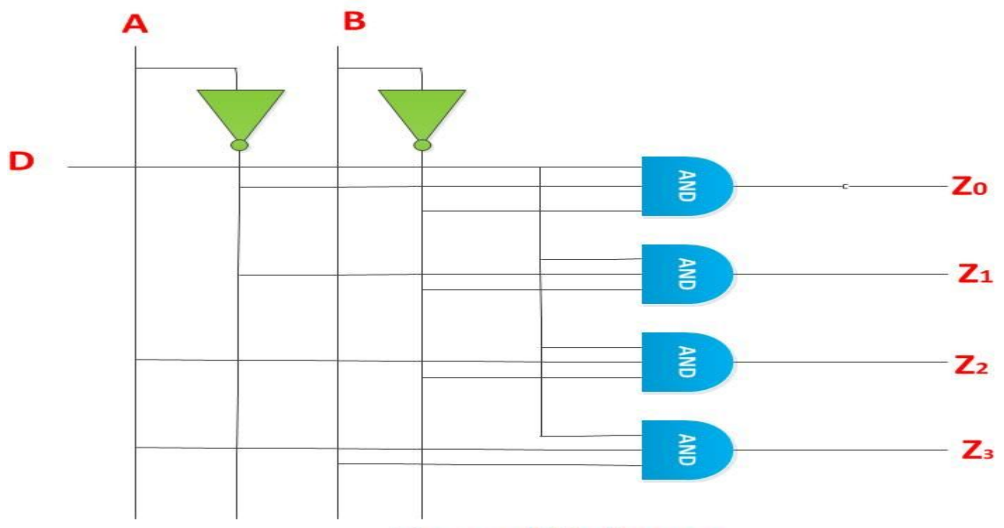
```
// 4:1 Multiplexer using data flow model
module mux4_1(out,in,sel);
input [3:0]in;
input [1:0]sel;
output out;
wire n0, n1, n2, n3, n4, n5;
assign n0 = ~ sel[0];
assign n1 = ~ sel[1];
assign n5 = in[0] & n0 & n1;
assign n4 = in[1] & n1 & sel[0];
assign n3 = in[2] & sel[1] & n0;
assign n2 = in[3] & sel[0] & sel[1];
assign out = n2 | n3 | n4 | n5;
endmodule
```

Aim 2: Development of Verilog modules for a 1x4 DEMUX.

1-to-4 Demultiplexer

A 1-to-4 demultiplexer consists one data input line as D, two select lines as S0 and S1 and four output line Z0 to Z3. The number of the output signal is always decided by the number of the select signal and vice versa.

The figure shows the block diagram of a 1-to-4 multiplexer in which the demultiplexer decodes the input through select line.



Demultiplexer

INPUTS			OUTPUTS			
A	B	D	Z0	Z1	Z2	Z3
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	1	0
1	1	0	0	0	0	0
1	1	1	0	0	0	1

Dataflow Modelling of 1:4 Demultiplexer:

```
// 1:4 Demultiplexer using data flow model
module demux1_4 (Y,in,sel);
input in;
input [1:0]sel;
output [3:0]Y;
wire n0,n1;
assign n0 = ~ sel[0];
assign n1 = ~ sel[1];
assign Y[0] = in & n0 & n1;
assign Y[1] = in & n1 & sel[0];
assign Y[2] = in & sel[1] & n0;
assign Y[3] = in & sel[0] & sel[1];
endmodule
```

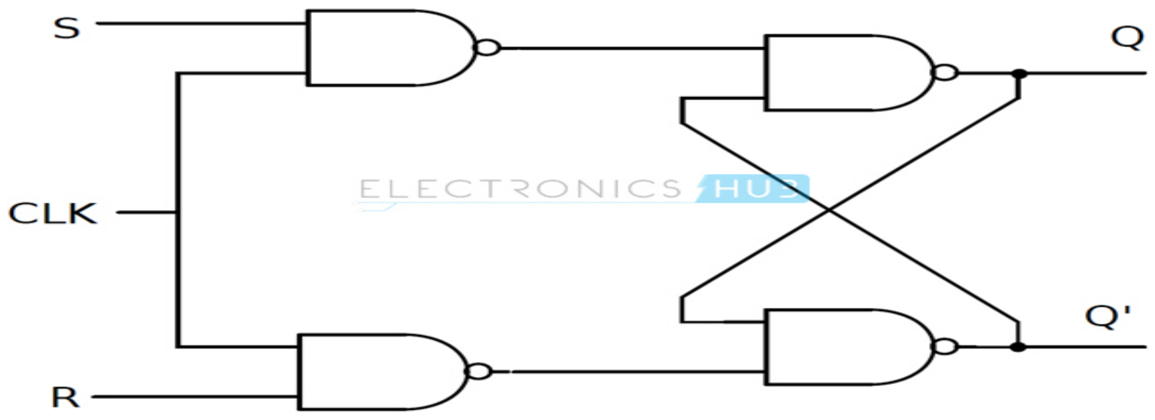
EXPERIMENT-4: Flipflops and counters

Aim 1: Development of Verilog modules for SR, JK and D flipflops.

i) SR Flip Flop

The SR flip flop is a 1-bit memory bistable device having two inputs, i.e., SET and RESET. The SET input 'S' set the device or produce the output 1, and the RESET input 'R' reset the device or produce the output 0. The SET and RESET inputs are labeled as S and R, respectively.

The SR flip flop stands for "Set-Reset" flip flop. The reset input is used to get back the flip flop to its original state from the current state with an output 'Q'. This output depends on the set and reset conditions, which is either at the logic level "0" or "1".



Truth Table for SR Flip Flop

INPUTS			OUTPUT	STATE
CLK	S	R	Q	
X	0	0	No Change	Previous
↑	0	1	0	Reset
↑	1	0	1	Set
↑	1	1	-	Forbidden

Behavioural Modelling of SR Flip Flop:

```
//SR flip-flop using behavioural model
module SR_FF (Q,Qbar,S,R,clk,);
input S,R,clk;
output reg Q;
output reg Qbar;
```

```
always @(posedge clk) begin
case({S,R})
{1'b0,1'b0}: begin Q=Q;Qbar=Qbar; end
{1'b0,1'b1}: begin Q=1'b0;Qbar=1'b1; end
{1'b1,1'b0}: begin Q=1'b1;Qbar=1'b0; end
{1'b1,1'b1}: begin Q=1'bx; Qbar=1'bx; end
endcase
end
endmodule
```

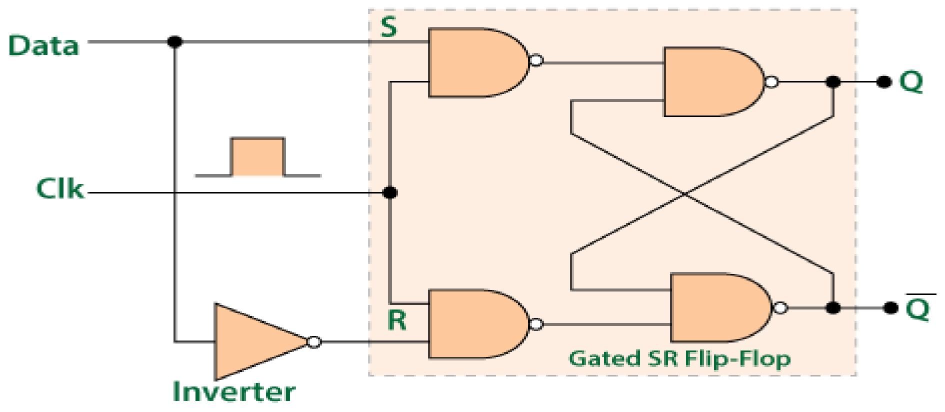
ii) D Flip Flop

In SR NAND Gate Bistable circuit, the undefined input condition of SET = "0" and RESET = "0" is forbidden. It is the drawback of the SR flip flop. This state:

- Override the feedback latching action.
- Force both outputs to be 1.
- Lose the control by the input, which first goes to 1, and the other input remains "0" by which the resulting state of the latch is controlled.

We need an inverter to prevent this from happening. We connect the inverter between the Set and Reset inputs for producing another type of flip flop circuit called D flip flop, Delay flip flop, D-type Bistable, D-type flip flop.

The D flip flop is the most important flip flop from other clocked types. It ensures that at the same time, both the inputs, i.e., S and R, are never equal to 1. The Delay flip-flop is designed using a gated SR flip-flop with an inverter connected between the inputs allowing for a single input D (Data).



Truth Table for the D-type Flip Flop

Clock	D	Q	Q'	Description
↓ » 0	X	Q	Q'	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

Behavioural Modelling of D Flip Flop:

```
//D Flip flop

module D_FF (Q,Qbar,D,clk);

input D;

input clk;

output reg Q;

output reg Qbar;

always @(posedge clk) begin

Q <= D;

Qbar <= !D;

end

endmodule
```

iii) JK Flip Flop

The SR Flip Flop or Set-Reset flip flop has lots of advantages. But it has the following switching problems:

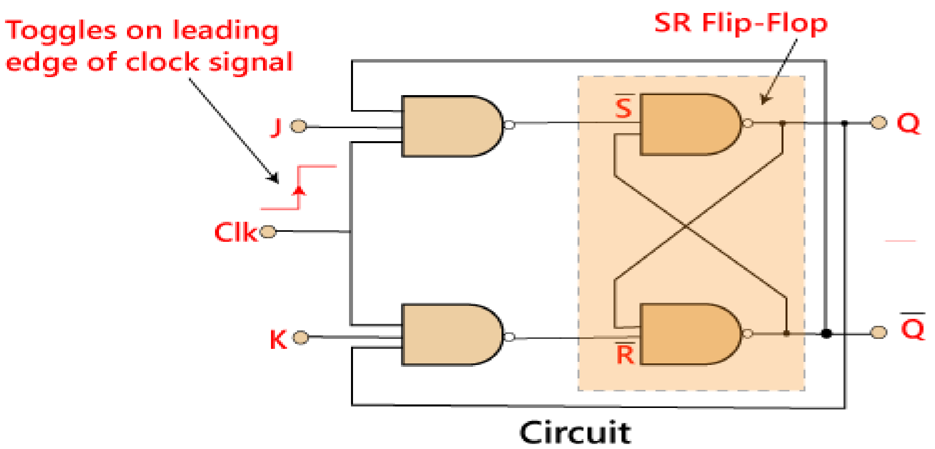
- When Set 'S' and Reset 'R' inputs are set to 0, this condition is always avoided.
- When the Set or Reset input changes their state while the enable input is 1, the incorrect latching action occurs.

The JK Flip Flop removes these two drawbacks of SR Flip Flop.

The JK flip flop is one of the most used flip flops in digital circuits. The JK flip flop is a universal flip flop having two inputs 'J' and 'K'. In SR flip flop, the 'S' and 'R' are the shortened abbreviated letters for Set and Reset, but J and K are not. The J and K are themselves autonomous letters which are chosen to distinguish the flip flop design from other types.

The JK flip flop work in the same way as the SR flip flop work. The JK flip flop has 'J' and 'K' flip flop instead of 'S' and 'R'. The only difference between JK flip flop and SR flip flop is that when both inputs of SR flip flop is set to 1, the circuit produces the invalid states as outputs, but in case of JK flip flop, there are no invalid states even if both 'J' and 'K' flip flops are set to 1.

The JK Flip Flop is a gated SR flip-flop having the addition of a clock input circuitry. The invalid or illegal output condition occurs when both of the inputs are set to 1 and are prevented by the addition of a clock input circuit. So, the JK flip-flop has four possible input combinations, i.e., 1, 0, "no change" and "toggle". The symbol of JK flip flop is the same as SR Bistable Latch except for the addition of a clock input.



Circuit

Same as for SR Latch	Clock	Input		Output		Description
	Clk	J	K	Q	Q’	
	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	¯↓_	0	1	1	0	Reset Q>>0
	X	0	1	0	1	
	¯↓_	1	0	0	1	Set Q>>1
	X	1	0	1	0	
Toggle action	¯↓_	1	1	0	1	Toggle
	¯↓_	1	1	1	0	

Note: Here we need to reduce the clock frequency from 50 MHz to 1 Hz so that the led toggling can be noticed, for this we use a clock divider module.

Behavioural Modelling of JK Flip Flop:

```
//Using Behavioural modelling JK Flipflop
module jkff(j,k,clk,q,qbar);
  input j,k,clk;
  output q,qbar;
```

```
 reg q,qbar;
clockdivider u1(clk, clk_out);
  always@(posedge clk_out)
    begin
      if(j == 0 && k == 0)
        begin
           q <= q;
           qbar<=qbar;
        end
      else if(j == 0 && k == 1)
          begin
           q <= 0;
           qbar<=1;
        end
      else if(j == 1 && k == 0)
           begin
           q <= 1;
           qbar<=0;
        end
      else if(j == 1 && k == 1)
          begin
           q <= !q;
           qbar<=!qbar;
        end
end
endmodule
//Clockdivider Module
module clockdivider(clk_in, clk_out);
input clk_in;
output clk_out;
reg [31:0] counter;
reg out;
initial begin
counter <= 32'b0;
out <= 1'b0;
```

```
end
always @ (posedge clk_in)
begin
counter <= counter + 1'b1;
if (counter > 50000000)
        begin
        out <= !out;
        counter <= 32'b0;
        end
end
assign clk_out = out;
endmodule
```

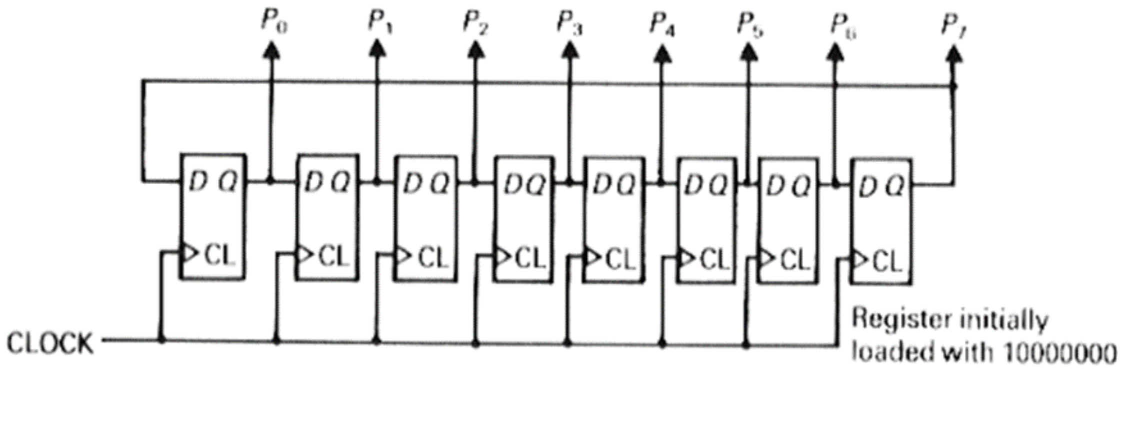
Aim 2: Development of Verilog modules for a binary decade/Johnson/Ring counter.

Ring Counter

A ring counter is a special type of application of the Serial IN Serial OUT Shift register. The only difference between the shift registers and the ring counter is that the last flip flop outcome is taken as the output in the shift register. But in the ring counter, this outcome is passed to the first flip flop as an input. All of the remaining things in the ring counter are the same as the shift register.

In the Ring counter

No. of states in Ring counter = No. of flip-flop used



Truth table of ring counter

The truth table of the 4-bit ring counter is explained below.

P0	P1	P2	P3	P4	P5	P6	P7
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0

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

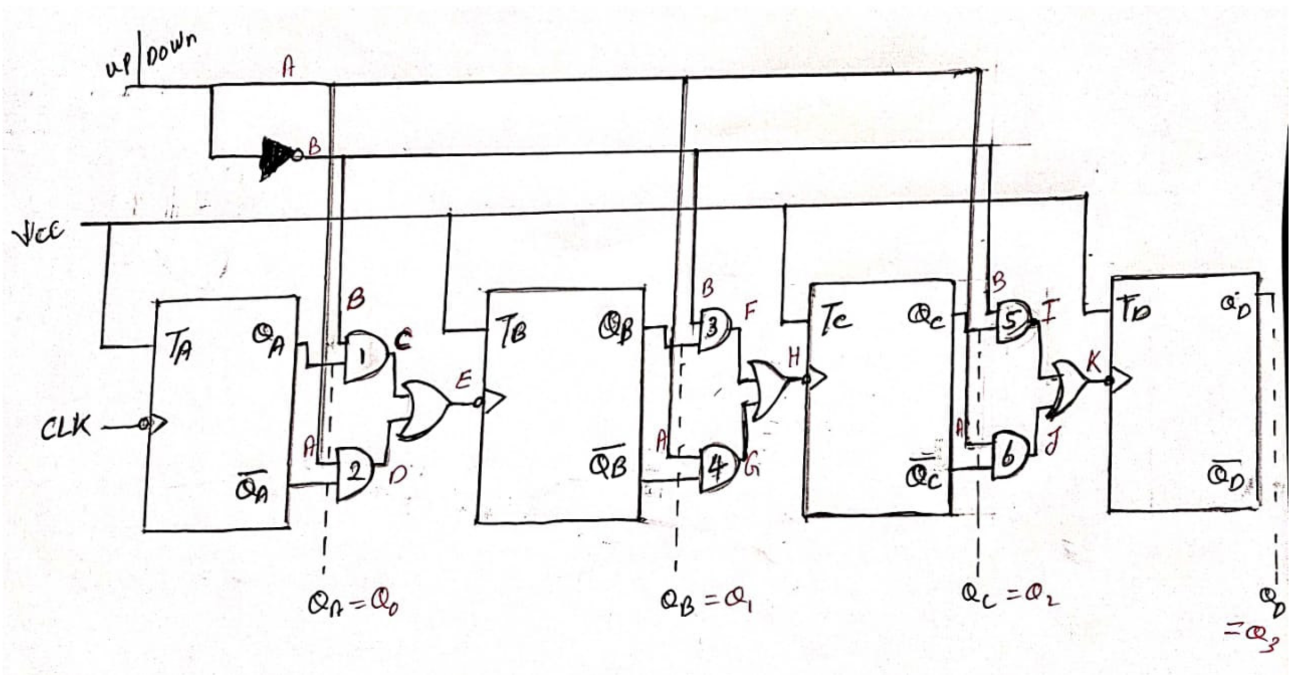
Note: Include the clock divider module

Behavioural Modelling of Ring Counter:

```
// behavioural modelling of ring counter
module ringcount(clk,out);
input clk;
output [7:0]out;
 reg [7:0]q;
clockdivider u1 (clk,clk_out);
always @(posedge clk_out)
begin
 if(q == 8'b00000000)
 q <= 8'b00000001;
 else
 begin
q[0]<=q[7];
q[1]<=q[0];
q[2]<=q[1];
q[3]<=q[2];
q[4]<=q[3];
q[5]<=q[4];
q[6]<=q[5];
q[7]<=q[6];
end
end
 assign out = q;
endmodule
```

EXPERIMENT-5: Asynchronous and Synchronous Counters in FPGA

Aim 1: Make a design of a 4-bit up down ripple counter using T-flip-lops, implement and test them on the FPGA board.



Note: Include the clock divider module

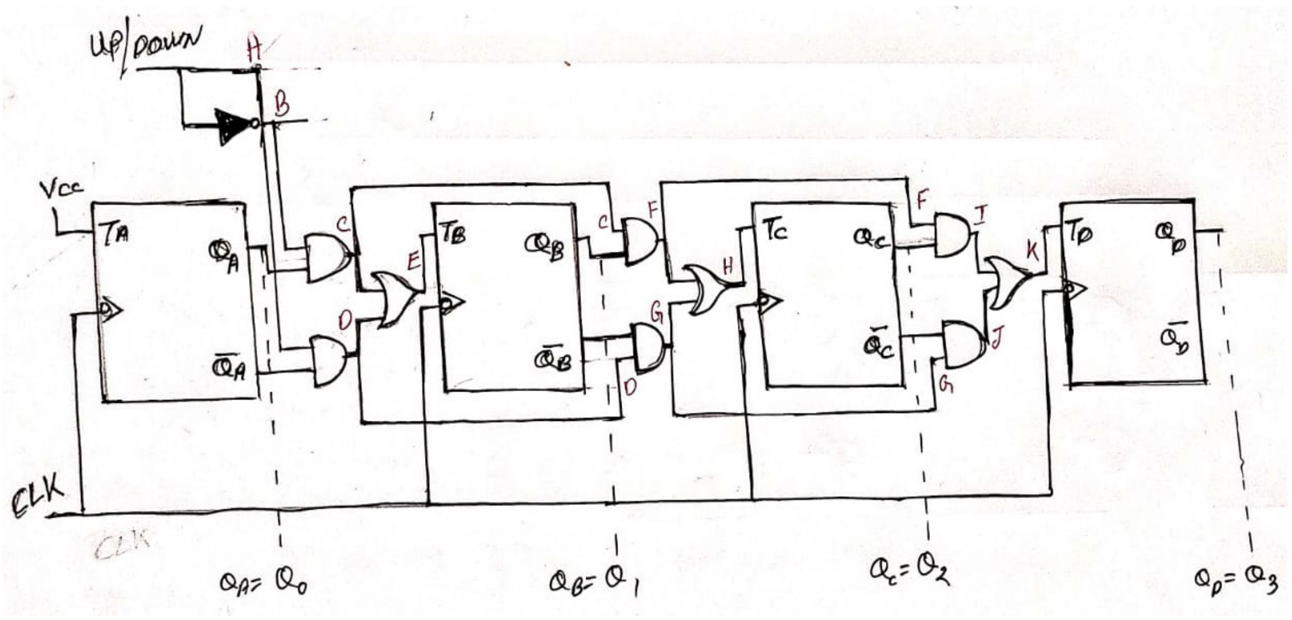
Behavioural Modelling of a 4-bit up down ripple counter using T-flip-lops:

```
//Ripple updown counter using T-Flipflop (Asynchronous)
module tff(T,clk,q,qbar);
        input T;
        input clk;
        output q,qbar;
        reg q,qbar;
initial begin
q = 0;
qbar = 1;
end
 always@(negedge clk)
        begin
    if (T == 1)
     begin
     q = !q;
     end
     else
     begin
                    q = q;
      end
                    qbar=!q;
          end
endmodule
module ripple_updown_counter ( input clk,input updown,output [3:0] out);
wire q0;
wire qn0;
wire q1;
wire qn1;
wire q2;
wire qn2;
wire q3;
wire qn3;
wire A;
wire B;
wire C;
wire D;
wire E;
wire F;
wire G;
wire H;
wire I;
wire J;
wire K;
assign A=updown;
not(B,updown);
clockdivider u1 (clk, clk_out);
  tff  u2 (.T(1'b1),.clk(clk_out),.q(q0),.qbar(qn0));
                   and(C,B,q0);
                   and(D,A,qn0);
                   or(E,C,D);
  tff  u3 (.T(1'b1),.clk(E),.q(q1),.qbar(qn1));
```

```
                and(F,B,q1);
                and(G,A,qn1);
                or(H,F,G);
  tff  u4 (.T(1'b1),.clk(H),.q(q2),.qbar(qn2));
                and(I,B,q2);
                and(J,A,qn2);
                or(K,I,J);
  tff  u5 (.T(1'b1),.clk(K),.q(q3),.qbar(qn3));

  assign out = {q3,q2,q1,q0};
endmodule
```

Aim 2: Make a design of a 4-bit up down synchronous counter using T-flip-lops in the previous experiment, implement and test them on the FPGA board.



Note: Include the clock divider module

Behavioural Modelling of a 4-bit up down Synchronous counter using T-flip-lops:

```
//Synchronous updown counter using T-Flipflop
module tff(T,clk,q,qbar);
        input T;
        input clk;
        output q,qbar;
        reg q,qbar;
```

```
initial begin
q = 0;
qbar = 1;
end
 always@(negedge clk)
begin
     if (T == 1)
       begin
       q = !q;
       end
     else
     begin
    q = q;
      end
   qbar=!q;
end
endmodule
module ripple_updown_counter (input clk,input updown,output [3:0] out);
wire q0;
wire qn0;
wire q1;
wire qn1;
wire q2;
wire qn2;
wire q3;
wire qn3;
wire A;
wire B;
wire C;
wire D;
wire E;
wire F;
wire G;
wire H;
wire I;
wire J;
wire K;
assign A=updown;
not(B, updown);
clockdivider u1 (clk, clk_out);
tff  u2 (.T(1'b1),.clk(clk_out),.q(q0),.qbar(qn0));
and(C,B,q0);
and(D,A,qn0);
or(E,C,D);
tff  u3 (.T(E),.clk(clk_out),.q(q1),.qbar(qn1));
and(F,C,q1);
and(G,D,qn1);
or(H,F,G);
 tff  u4 (.T(H),.clk(clk_out),.q(q2),.qbar(qn2));
and(I,F,q2);
and(J,G,qn2);
or(K,I,J);
 tff  u5 (.T(K),.clk(clk_out),.q(q3),.qbar(qn3));
 assign out = {q3,q2,q1,q0};
endmodule
```

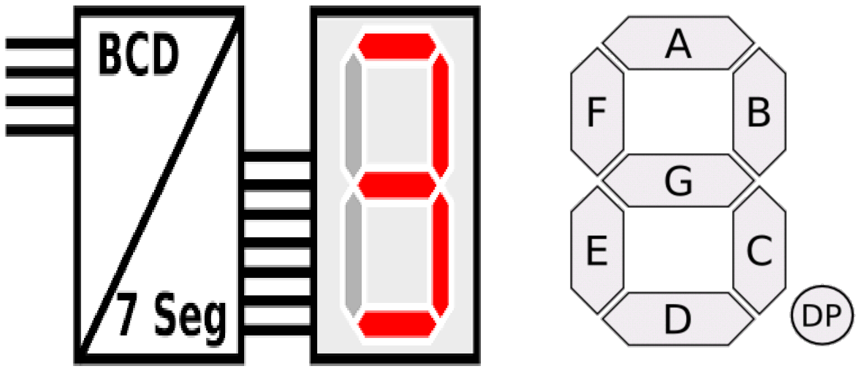
EXPERIMENT-6: BCD to Seven Segment Decoder in FPGA

Aim 1: Make a gate level design of a seven-segment decoder, write to FPGA and test its functionality.

BCD to Seven Segment Display Decoder

The Seven segment display is most frequently used the digital display in calculators, digital counters, digital clocks, measuring instruments, etc. Usually, the displays like LED's as well as LCDs are used to display the characters as well as numerical numbers. But, a seven-segment display is used to display both the numbers and characters.

The decoder is an essential component in BCD to seven segment decoders. A decoder is nothing but a combinational logic circuit mainly used for converting a BCD to an equivalent decimal number. It can be a BCD to seven segment decoders. A combinational logic circuit can be built with logic gates which include inputs as well as outputs. The output of this circuit mainly lies in the current condition of the inputs.



BCD to Seven Segment Display

The circuit design, as well as operation, mainly depends on the concepts of **Boolean algebra** as well as logic gates. A seven segment **LED display circuit** can be built with eight LEDs. The common terminals are either anode otherwise cathode. A general cathode seven segment display includes 8 pins where 7-pins are input pins that are marked with from a to g & 8th-pin is a ground pin.

The Out Put is Given by:

a = X+Z+YW+Y'W'

b = Y'+Z'W'+ZW

c= Y+Z'+W

d = Y'W'+ZW'+YZ'W+Y'Z+X

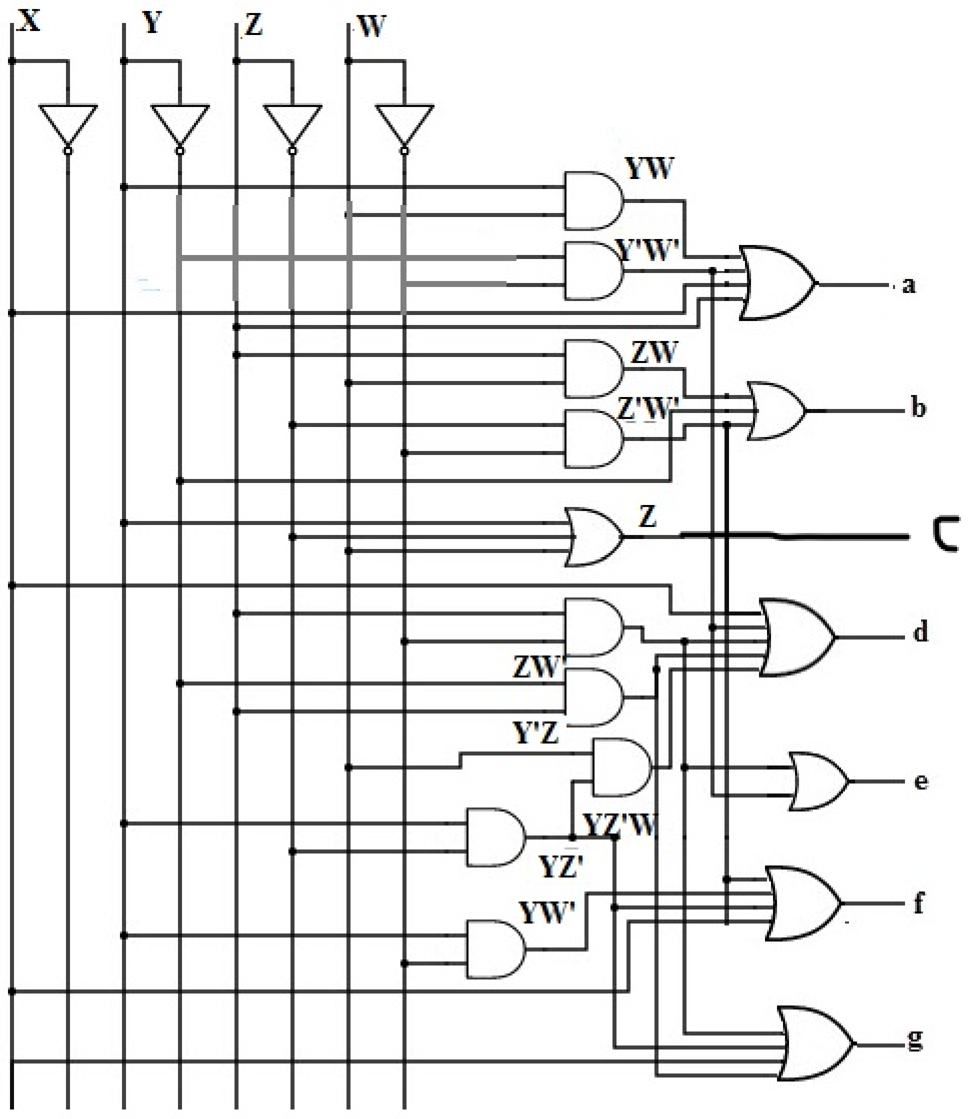
e= Y'W'+ZW'

f= X + Z'W'+YZ'+YW'

g = X+YZ'+Y'Z+ZW'

The Truth Table for a BCD to Seven Segment Display

Digit	X	Y	Z	W	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	1	1	0	0	1	1	1	1
2	0	0	1	0	0	0	1	0	0	1	0
3	0	0	1	1	0	0	0	0	1	1	0
4	0	1	0	0	1	0	0	1	1	0	0
5	0	1	0	1	0	1	0	0	1	0	0
6	0	1	1	0	0	1	0	0	0	0	0
7	0	1	1	1	0	0	0	1	1	1	1
8	1	0	0	0	0	0	0	0	0	0	0
9	1	0	0	1	0	0	0	0	1	0	0



Behavioural Modelling of a BCD to Seven Segment Display Decoder:

```
// BCD to Seven Segment Display Decoder
module bcd1(seg,X,Y,Z,W);
    input X,Y,Z,W;
    output [6:0] seg;
    reg [6:0] seg;
always @ (X,Y,Z,W)
begin
seg[0] = ~(X|Z|(Y&W)|(~Y&~W));
seg[1] = ~(~Y|(~Z&~W)|(Z&W));
seg[2] = ~(Y|~Z|W);
seg[3] = ~((~Y&~W)|(Z&~W)|(Y&~Z&W)|(~Y&Z)|X);
seg[4] = ~((~Y&~W)|(Z&~W));
seg[5] = ~(X|(~Z&~W)|(Y&~Z)|(Y&~W));
seg[6] = ~(X|(Y&~Z)|(~Y&Z)|(Z&~W));
end
endmodule
```

Aim 2: Test it with switches and seven segment display. Use output ports for connection to the display.

Behavioural Modelling of a BCD to Seven Segment Display Decoder:

```
// BCD to Seven Segment Display Decoder
module bcd_to_7seg(seg,bcd);
    input [3:0] bcd;
    output [6:0] seg;
    reg [6:0] seg;
always @(bcd)
 begin
case (bcd) //case statement
        0 : seg = 7'b0000001;
        1 : seg = 7'b1001111;
        2 : seg = 7'b0010010;
        3 : seg = 7'b0000110;
        4 : seg = 7'b1001100;
        5 : seg = 7'b0100100;
        6 : seg = 7'b0100000;
        7 : seg = 7'b0001111;
        8 : seg = 7'b0000000;
        9 : seg = 7'b0000100;
        10: seg = 7'b0001000;//A
        11: seg = 7'b1100000;//b
        12: seg = 7'b0110001;//C
        13: seg = 7'b1000010;//d
        14: seg = 7'b0110000;//E
        15: seg = 7'b0111000;//F
endcase
end
endmodule
```