DIGITAL LAB (CSL 202)

Laboratory Manual

PART-II

BTech Semester IV

SREE BUDDHA COLLEGE OF ENGINEERING

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

Department

Of

Electronics and Communication Engineering

2022-23

TABLE OF CONTENT

Course Objectives:

CO 1	Design and implement combinational logic circuits using Logic Gates (Cognitive Knowledge Level: Apply)
CO 2	Design and implement sequential logic circuits using Integrated Circuits (Cognitive Knowledge Level: Apply)
CO 3	Simulate functioning of digital circuits using programs written in a Hardware Description Language (Cognitive Knowledge Level: Apply)
CO 4	Function effectively as an individual and in a team to accomplish a given task of designing and implementing digital circuits (Cognitive Knowledge Level: Apply)

Assessment

Mark distribution

Total Marks	CIE	ESE	ESE Duration
150	75	75	3 hours

Continuous Internal Evaluation Pattern:

Attendance: 15 marks
Continuous Evaluation in Lab: 30 marks
Continuous Assessment Test: 15 marks
Viva-voce: 15 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 module 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

Most of the 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 ModelSim HDL simulator

ModelSim is a program created by Mentor Graphics used for simulating your VHDL and Verilog designs. It is the most widely use simulation program in business and education. ModelSim simulates behavioural, RTL, and gate-level code - delivering increased design quality and debug productivity with platform-independent compile. Single Kernel Simulator technology enables transparent mixing of VHDL and Verilog in one design. Modelsim Student Edition for free for your personal use.

Experiment 1: Realization of Logic Gates and Familiarization of Verilog

a) Familiarization of the basic syntax of Verilog

b) Development of Verilog modules for basic gates and to verify truth tables.

c) Design and simulate the HDL code to realize three and four variable Boolean functions

Aim1: Familiarization of the basic syntax of Verilog

Module: A Module in Verilog is declared within the pair of keywords module and endmodule. Following the keyword module are the module name and port interface list.

```
module my_module ( a, b, c, d );
 input a, b;
 output c, d;
 ...
endmodule
```

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

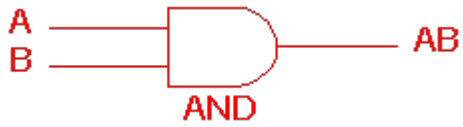
Wires, regs and parameters are the data types used as operands in Verilog expressions

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.

Aim2: Development of Verilog modules for basic gates and to verify truth tables.

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**	**Behavioural Model**	**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 or y) begin
		if (x = = 1'b1 && y = = 1'b1)
		begin
		z =1'b1;
		end
		else
		z=1'b0;
		end
		endmodule
			module anddf_tb;
			reg x,y;
			wire z;
			anddf uut (.x(x),.y(y), .z(z));
			initial begin
			x = 0; y = 0; #50;
			x = 0; y = 1; #50;
			x = 1; y = 0; #50;
			x = 1; y = 1; #50;
			end
			endmodule

OR Gate

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

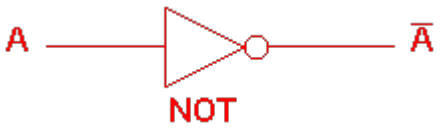
The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.

Structural Model	**Data Flow Model**	**Behavioural Model**	**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 or y) begin
		if (x = = 1'b0 && y = = 1'b0)
		begin
		z =1'b0;
			module ordf_tb;
			reg x,y;
			wire z;
			ordf uut (.x(x),.y(y),.z(z));
			initial begin
			x=0; y=0;#50;
			x=0; y=1;#50;
			x=1; y=0;#50;

		end
		else
		z=1'b1;
		end
		endmodule
			x=1; y=1;#50;
			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	**Data Flow Model**	**Behavioural Model**	**TTEST BENCH:**

module notstr(x,z);
input x;
output z;
not g1(z,x);
endmodule
	module notdf(x,z);
	input x;
	output z;
	assign z=! x;
	endmodule
		module notbeh(x,z);
		input x;
		output z;
		reg z;
		always @(x) begin
		if (x = = 1'b1) begin
		z =1'b0;
		end
		else
		z=1'b1;
		end
		endmodule
			module notdf_tb;
			reg x;
			wire z;
			notdf uut (.x(x),.z(z));
			initial begin
			x=0; #50;
			x=1; #50;
			end
			endmodule

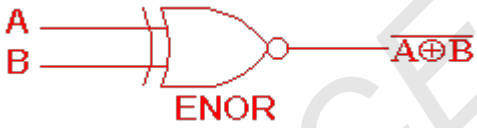
EXOR Gate

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

Structural Model	**Data Flow Model**	**Behavioural Model**	**TEST BENCH:** module

module xorstr(x,y,z);
inputx,y;
output z;
xor g1(z,x,y);
endmodule
	module xordf(x,y,z);
	inputx,y;
	output z;
	assign z=(x^y);
	endmodule
		module xorbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x or y) begin
		if ((x = = 1'b1 && y = = 1'b1)
		|| (x = = 1'b0 && y = = 1'b0))
		begin
		z =1'b0;
		end
		else
		z=1'b1;
		end
		endmodule
			xordf_tb;
			reg x,y;
			wire z;
			xordf uut (.x(x),.y(y),.z(z));
			initial begin
			x=0; y=0; #50;
			x=0; y=1; #50;
			x=1; y=0; #50;
			x=1; y=1; #50;
			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

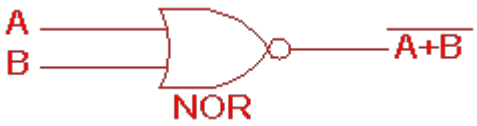
The 'Exclusive-NOR' gate circuit does the opposite to the EXOR gate. It will give a low output if either, but not both, of its two inputs are high. The symbol is an EXOR gate with a small circle on the output. The small circle represents inversion.

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

module xnorstr(x,y,z);
inputx,y;
output z;
xnor g1(z,x,y);
endmodule
	module xnordf(x,y,z);
	inputx,y;
	output z;
	assign z= !(x^y);
	endmodule
		module xnorbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x or y) begin
		if ((x = = 1'b1 && y = =
		1'b1) || (x = = 1'b0 && y =
		= 1'b0)) begin
			module xnordf_tb;
			reg x,y;
			wire z;
			xnordf uut (.x(x),.y(y),.z(z));
			initial begin
			x=0; y=0;#50;
			x=0; y=1;#50;
			x=1; y=0;#50;

		z =1'b1;
		end
		else
		z=1'b0;
		end
		endmodule
			x=1; y=1;#50;
			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**	**Behavioural Model**	**TEST BENCH:**

module norstr(x,y,z);
inputx,y;
output z;
nor g1(z,x,y);
endmodule
	module nordf(x,y,z);
	inputx,y;
	output z;
	assign z= !(x|y);
	endmodule
		module norbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x,y)
		always @(x or y) begin
		if (x = = 1'b0 && y = = 1'b0)
		begin
		z =1'b1;
		end
		else
		z=1'b0;
		end
		endmodule
			module nordf_tb;
			reg x,y;
			wire z;
			nordf uut (.x(x),.y(y),.z(z));
			initial begin
			x=0; y=0;#50;
			x=0; y=1;#50;
			x=1; y=0;#50;
			x=1; y=1;#50;
			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 inputs is in active low state.

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

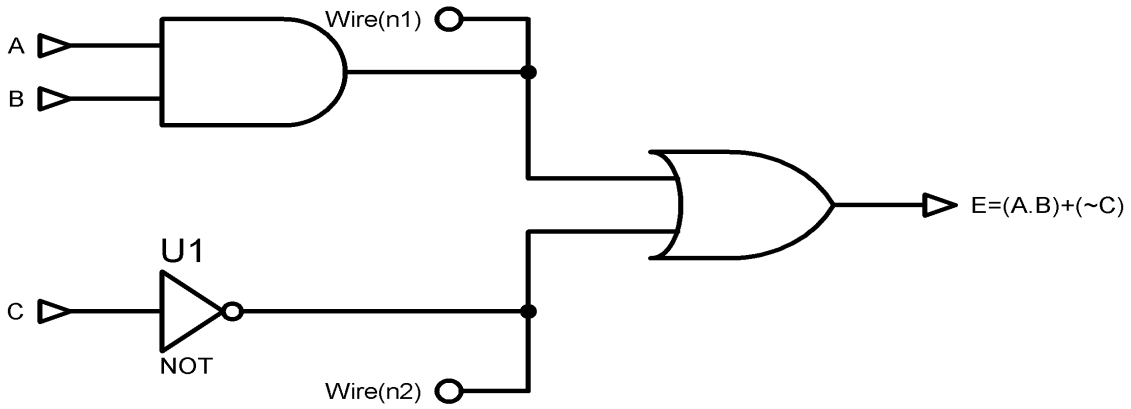
modulenandstr(x,y,z);
inputx,y;
output z;
nand g1(z,x,y);
endmodule
	module nanddf(x,y,z);
	inputx,y;
	output z;
	assign z= !(x&y);
	endmodule
		module nandbeh(x,y,z);
		input x,y;
		output z;
		reg z;
		always @(x or y) begin
		if (x = = 1'b1 && y = = 1'b1) begin
		z =1'b0;
		end
		else
		z=1'b1;
		end
		endmodule
			module nanddf_tb;
			reg x,y;
			wire z;
			ordf uut (.x(x),.y(y),.z(z));
			initial begin
			x=0; y=0;#50;
			x=0; y=1;#50;
			x=1; y=0;#50;
			x=1; y=1;#50;
			end
			endmodule

NOTE: If we want to display the occurrence of the event on the console then following test bench program can be used in above all cases by changing the module mane as required

```
module name_TB;
reg x;
reg y;
wire z;
name uut (.x(x),.y(y),.z(z));
initial begin
x=1'b0;
y=1'b0;
end
always #50 x=!x;
always #100 y=!y;
initial $monitor($time,"----X=%b,Y=%b,Z=%b",x,y,z);
initial #1000 $stop;
endmodule
```

Aim3: Design and simulate the HDL code to realize three and four variable Boolean functions

i) Three variable Boolean Function example



Let's write a data flow model for this

// Three variable Boolean Function example

```
module boolFun1 (A,B,C,E);
input A,B,C;
output E;
wire n1,n2;
assign n1= A & B;
assign n2= !C;
assign E= n1 | n2;
endmodule
```

// Testbench for Three variable Boolean Function

```
module boolFun1_TB;
reg A;
reg B;
reg C;
wire E;
boolFun1 uut (.A(A),.B(B),.C(C),.E(E));
initial begin
A=1'b0;B=1'b0;C=1'b1;#100;
A=1'b1;B=1'b1;C=1'b0;#100;
A=1'b0;B=1'b1;C=1'b0;#100;
```

A=1'b1;B=1'b0;C=1'b1;#100;

A=1'b1;B=1'b1;C=1'b1;#100;

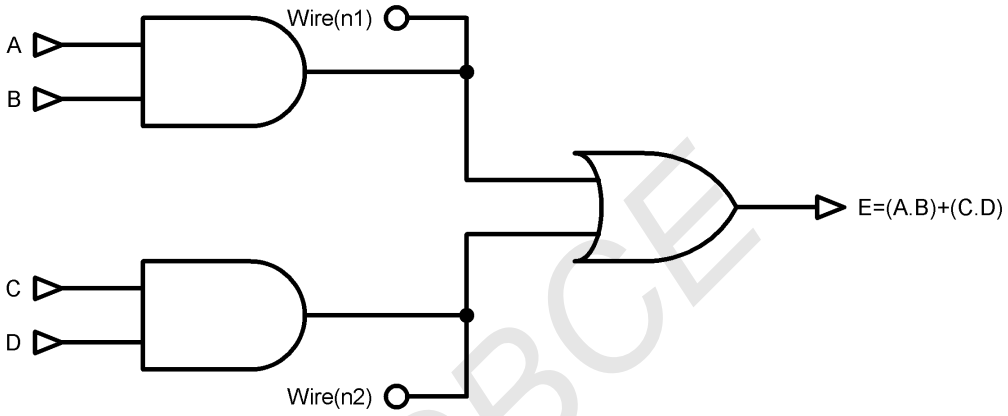
end

initial $monitor($time,"------A=%b,B=%b,C=%b,E=%b",A,B,C,E);

initial #500 $stop;

endmodule

ii) Four variable Boolean Function example



Let's write a data flow model for this

// Four variable Boolean Function example

module boolFun2 (A,B,C,D,E);

input A,B,C,D;

output E;

wire n1,n2;

assign n1= A & B;

assign n2= C & D;

assign E= n1 | n2;

endmodule

// Testbench for Four variable Boolean Function

module boolFun2_TB;

reg A;

reg B;

reg C;

reg D;

```
wire E;
boolFun2 uut (.A(A),.B(B),.C(C),.D(D),.E(E));
initial begin
A=1'b0;B=1'b0;C=1'b1;D=1'b1;#100;
A=1'b1;B=1'b1;C=1'b0;D=1'b0;#100;
A=1'b0;B=1'b1;C=1'b0;D=1'b1;#100;
A=1'b1;B=1'b0;C=1'b1;D=1'b0;#100;
A=1'b1;B=1'b1;C=1'b1;D=1'b1;#100;
end
initial $monitor($time,"------A=%b,B=%b,C=%b,D=%b,E=%b",A,B,C,D,E);
initial #500 $stop;
endmodule
```

Experiment -2: Half adder and full adder

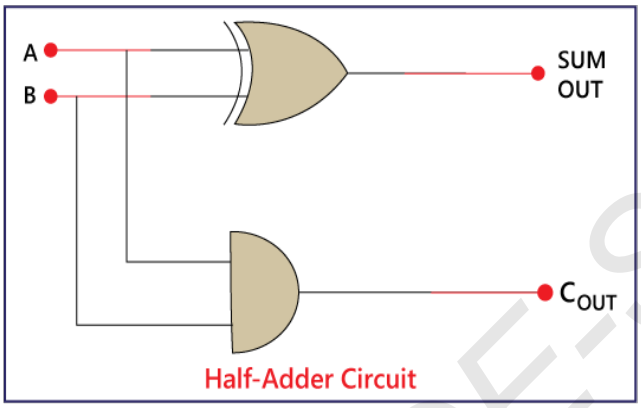
(a) Development of Verilog modules for half adder in 3 modelling styles (dataflow/ structural/behavioural).

(b) Development of Verilog modules for full adder in structural modelling using half adder.

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 producing 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 halfadd(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 halfadd(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 behavioural modelling

```
module halfadd(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
```

// Testbench for Half adder

```
module halfadd_TB;
reg x;
reg y;
```

```
wire sum;
wire carry;
halfadd uut (.x(x),.y(y),.sum(sum),.carry(carry));
initial begin
x=1'b0;y=1'b0;#100;
x=1'b0;y=1'b1;#100;
x=1'b1;y=1'b0;#100;
x=1'b1;y=1'b1;#100;
end
initial $monitor($time,"------X=%b,Y=%b,SUM=%b,CARRY=%b",x,y,sum,carry);
initial #400 $stop;
endmodule
```

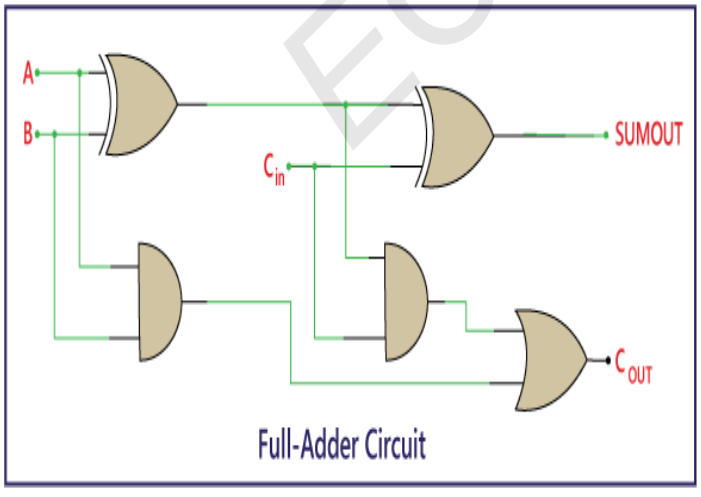
Aim 2: Development of Verilog modules for full adder in dataflow modelling using half adder.

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.

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.



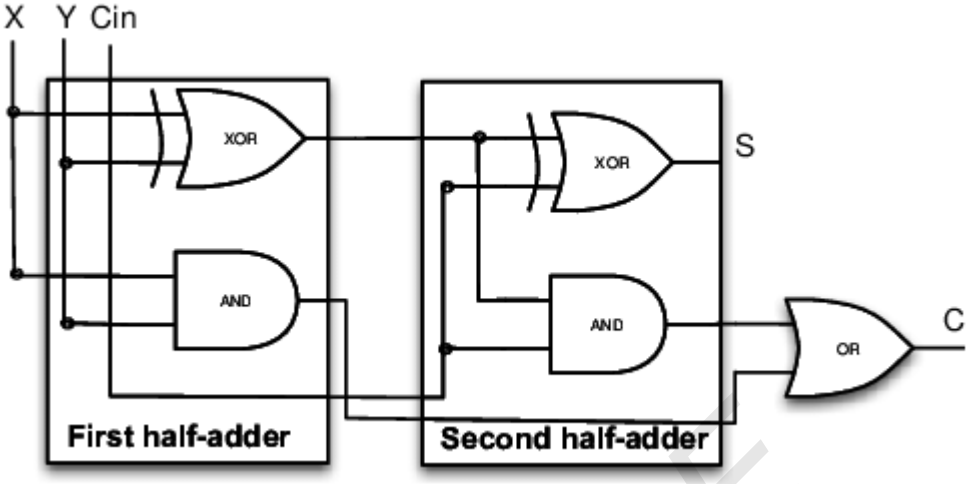
Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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 structural model

//half adder module

```
module halfadd(x, y, sum, carry);
input x,y;
output sum;
output carry;
xor(sum,x,y);
and(carry,x,y);
endmodule
```

//fulladder

```
module fulladd (x,y,cin,sumout,carryout);
input x,y,cin;
output sumout;
output carryout;
//Internal connections
wire a;
wire b;
wire c;
wire d;
```

//Instantiate the half adder 1

```
halfadd ha1 (.x(x),.y(y),.sum(a),.carry(b));
```

//Instantiate the half adder 2

```
halfadd ha2 (.x(a),. y(cin),.sum(c),.carry(d));
assign sumout= c;
assign carryout = d | b;
endmodule
```

// Testbench for Fulladder using two half adders using a structural model

```
module fulladd_TB;
reg x;
reg y;
reg cin;
wire sumout;
wire carryout;
fulladd uut (.x(x),.y(y),.cin(cin),.sumout(sumout),.carryout(carryout));
initial begin
x=1'b0;y=1'b0;cin=1'b0;#100;
x=1'b0;y=1'b0;cin=1'b1;#100;
x=1'b0;y=1'b1;cin=1'b0;#100;
x=1'b0;y=1'b1;cin=1'b1;#100;
x=1'b1;y=1'b0;cin=1'b0;#100;
x=1'b1;y=1'b0;cin=1'b1;#100;
x=1'b1;y=1'b1;cin=1'b0;#100;
x=1'b1;y=1'b1;cin=1'b1;#100;
end
initial $monitor($time,"------ X = %b,Y = %b, CIN = %b, SUMOUT =%b, CARRYOUT= %b ", x,
y, cin, sumout, carryout);
initial #800 $stop;
endmodule
```

Experiment -3: Design of code converters Design and simulate the HDL code for

(a) 4- bit binary to Gray code converter

(b) 4- bit Gray to binary code converter

Aim1: Development of Verilog modules for a 4- bit binary to Gray code converter

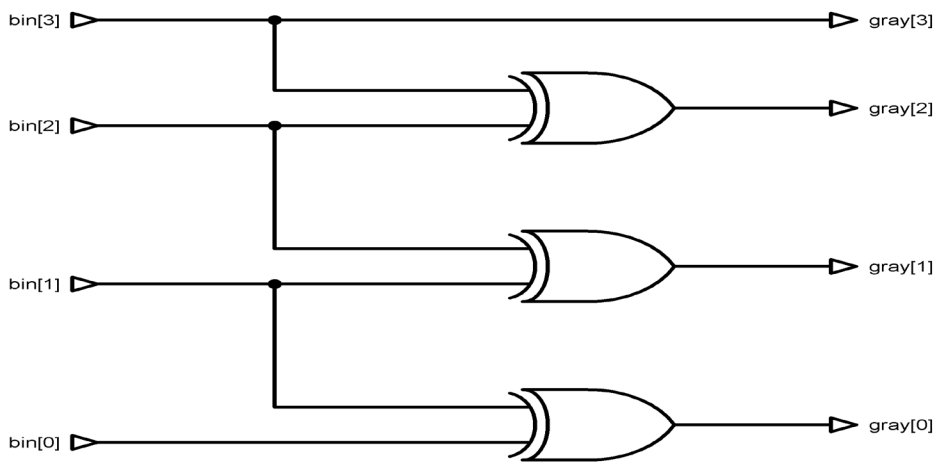
Binary to Gray code converter:

The logical circuit which converts the binary code to equivalent gray code is known as the binary to gray code converter. An n-bit gray code can be obtained by reflecting an n-1 bit code about an axis after 2^{n-1} rows and putting the MSB (Most Significant Bit) of 0 above the axis and the MSB of 1 below the axis.

The 4 bit binary to gray code conversion table is given below:

Decimal Number	4 bit Binary Number ABCD	4 bit Gray Code $G_1G_2G_3G_4$
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

The binary to gray code converter circuit is shown below:



// Binary to Gray code conversion

```
module bin_gray (bin,gray);
input [3:0] bin;
output[3:0] gray;
assign gray[3]=bin[3];
assign gray[2]=bin[3] ^ bin[2];
assign gray[1]=bin[2] ^ bin[1];
assign gray[0]=bin[1] ^ bin[0];
endmodule
```

// Test bench for Binary to Gray code conversion

```
module bin_gray_TB;
  reg [3:0] bin;
  wire [3:0] gray;
  bin_gray uut(.bin(bin),.gray(gray));
 initial begin
      bin <= 0; #100;
      bin <= 1; #100;
      bin <= 2; #100;
      bin <= 3; #100;
      bin <= 4; #100;
      bin <= 5; #100;
      bin <= 6; #100;
      bin <= 7; #100;
      bin <= 8; #100;
      bin <= 9; #100;
      bin <= 10;#100;
      bin <= 11;#100;
      bin <= 12;#100;
      bin <= 13;#100;
      bin <= 14;#100;
      bin <= 15;#100;
  end
```

initial $monitor($time,"----BINARY = %d, GRAY = %d",bin,gray);

initial #1600 $stop;

endmodule

Aim2: Development of Verilog modules for a 4- bit Gray to binary code converter.

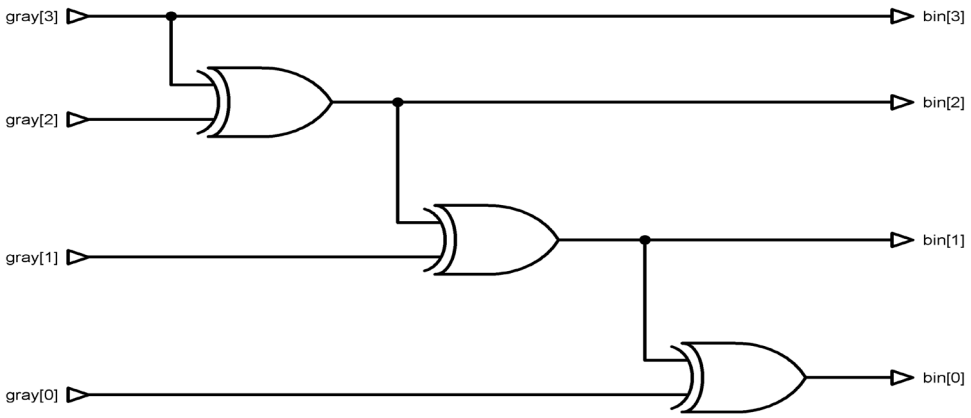
Gray to Binary Code Converter

In a gray to binary code converter, the input is gray code and the output is its equivalent binary code.

Let us consider a 4-bit gray to binary code converter. To design a 4-bit gray to binary code converter, we first have to draw a gray code conversion table, as shown below:

4 bit Gray Code	4 bit Binary Code
A B C D	B_4 B_3 B_2 B_1
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 1	0 0 1 0
0 0 1 0	0 0 1 1
0 1 1 0	0 1 0 0
0 1 1 1	0 1 0 1
0 1 0 1	0 1 1 0
0 1 0 0	0 1 1 1
1 1 0 0	1 0 0 0
1 1 0 1	1 0 0 1
1 1 1 1	1 0 1 0
1 1 1 0	1 0 1 1
1 0 1 0	1 1 0 0
1 0 1 1	1 1 0 1
1 0 0 1	1 1 1 0
1 0 0 0	1 1 1 1

The gray code to binary converter circuit is shown below:



// Gray to Binary code conversion

module gray_bin (gray, bin);

```
input [3:0] gray;
output[3:0] bin;
assign bin[3]= gray[3];
assign bin[2]= gray[3] ^ gray[2];
assign bin[1]= (gray[3] ^ gray[2]) ^ gray[1];
assign bin[0]= ((gray[3] ^ gray[2]) ^ gray[1])^ gray[0];
endmodule
```

//Testbench for Gray to Binary code conversion

```
module gray_bin_TB;
  reg [3:0] gray;
  wire [3:0] bin;
  gray_bin uut(.gray(gray),.bin(bin));
 initial begin
      gray <= 0; #100;
      gray <= 1; #100;
      gray <= 2; #100;
      gray <= 3; #100;
      gray <= 4; #100;
      gray <= 5; #100;
      gray <= 6; #100;
      gray <= 7; #100;
      gray <= 8; #100;
      gray <= 9; #100;
     gray  <=  10;  #100;
     gray  <=  11,  #100;
     gray   <=   12;#100;
     gray   <=   13;#100;
     gray   <=   14;#100;
     gray   <=   15;#100;
  end
initial $monitor($time,"----GRAY = %d, BINARY = %d ",gray,bin);
endmodule
```

Experiment -4: Mux and Demux in Verilog

(a) Development of Verilog modules for a 4x1 MUX.

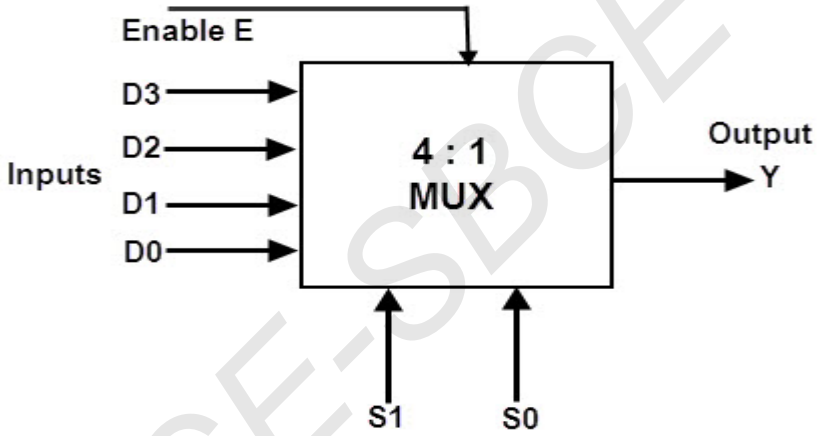
(b) Development of Verilog modules for a 1x4 DEMUX.

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

4-to-1 Multiplexer

A 4-to-1 multiplexer consists of 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 to the output line. The particular input combination on select lines selects one of the inputs (D0 through D3) to the output.

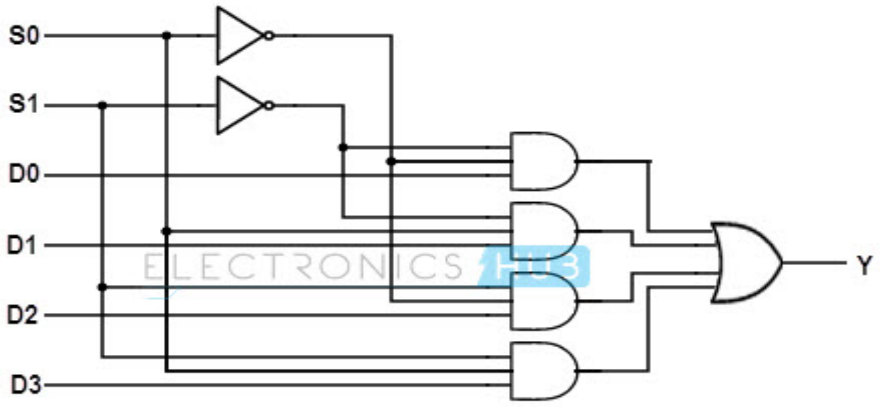
The figure below shows the block diagram of a 4-to-1 multiplexer in which the multiplexer decodes the input through a select line.



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
```

//Test bench for4:1 Multiplexer using data flow model

```
module mux4_1_TB;
// Inputs
reg [3:0] in;
reg [1:0] sel;
// Outputs
wire out;
// Instantiate the Unit Under Test (UUT)
mux4_1 uut (.out(out),.in(in),.sel(sel));
initial begin
in=4'b0000;
```

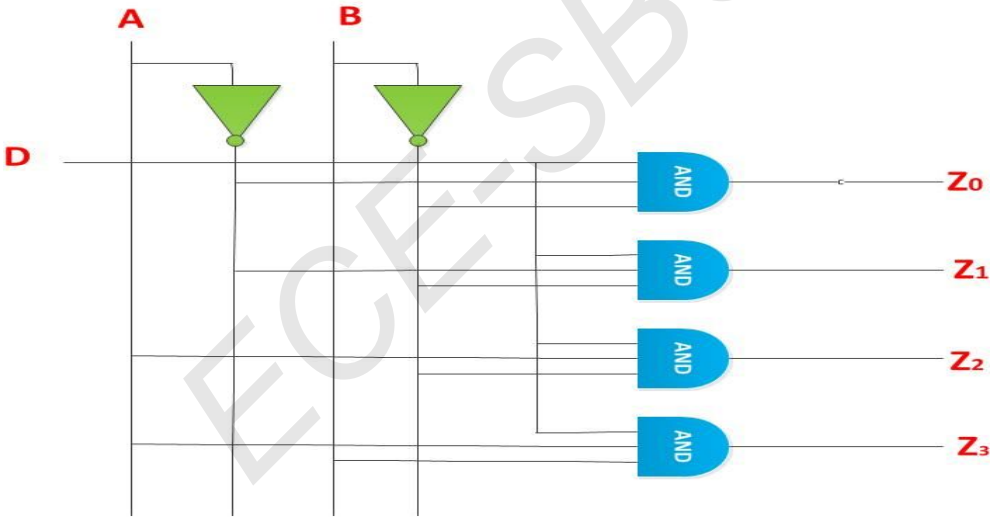
```
sel=2'b00;
end
always #25 sel=sel+1;
always #100 in=in+1;
initial $monitor($time,"------in[3]=%b, in[2]=%b, in[1]=%b, in[0]=%b, sel[1]=%b, sel[0]=%b,
out=%b",in[3],in[2],in[1],in[0],sel[1],sel[0],out);
endmodule
```

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

1-to-4 Demultiplexer

A 1-to-4 demultiplexer consists of 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
```

//Test bench for 1:4 Demultiplexer using data flow model

```
module demux1_4_TB;
// Inputs
reg  in;
reg [1:0]sel;
// Outputs
wire [3:0]Y;
// Instantiate the Unit Under Test (UUT)
demux1_4 uut (.Y(Y),.in(in),.sel(sel));
initial begin
in=1'b0;
sel=2'b00;
end
always #25 sel=sel+1;
always #100 in=in+1;
initial $monitor($time,"------in=%b, sel[1]=%b, sel[0]=%b, Y[0]=%b, Y[1]=%b, Y[2]=%b, Y[3]=%b",in,sel[1],sel[0],Y[0],Y[1],Y[2],Y[3]);
endmodule
```

Experiment -5: Adder/Subtractor

(a) Write the Verilog modules for a 4-bit adder/subtractor

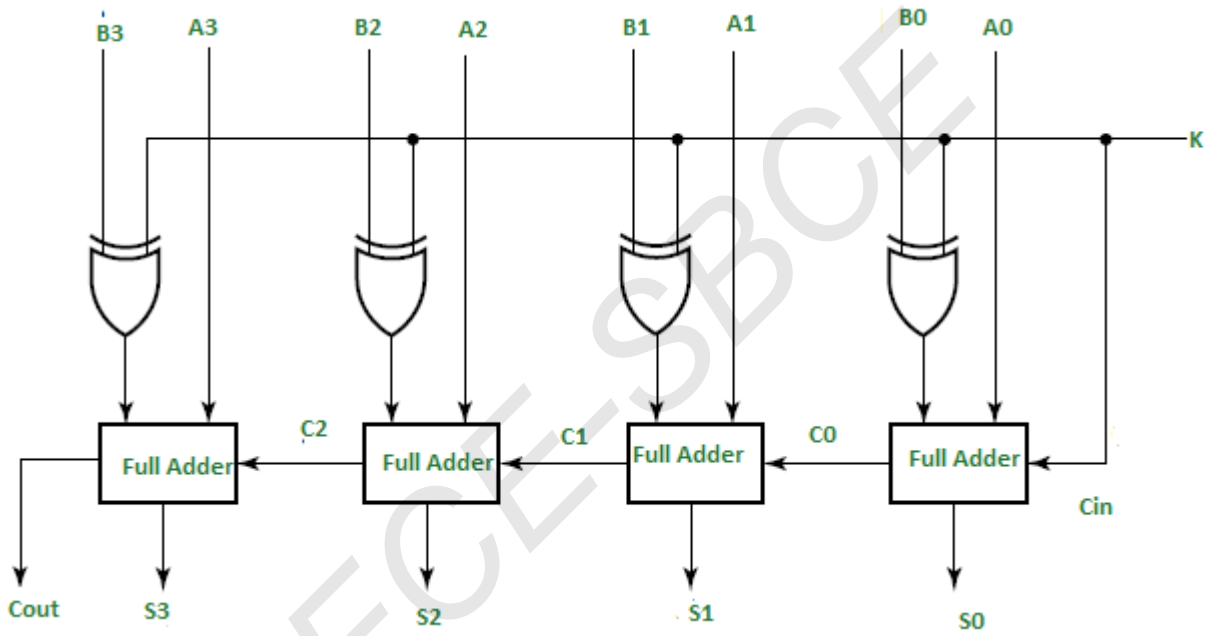
(b) Development of Verilog modules for a BCD adder

Aim 1: Development of Verilog modules for a 4-bit adder/subtractor

4-bit binary Adder-Subtractor

A Binary Adder-Subtractor is one which is capable of both addition and subtraction of binary numbers in one circuit itself. The operation being performed depends upon the binary value the control signal holds. It is one of the components of the ALU (Arithmetic Logic Unit).

This Circuit Requires prerequisite knowledge of Exor Gate, Binary Addition and Subtraction, Full Adder.



// Four Bit Full Adder_Substractor using data flow model

```
// Full adder
module full_adder(a,b,c_in,s,c_out);
input a, b, c_in;
output s, c_out;
wire net1, net2, net3;
assign net1 = a ^ b;
assign net2 = net1 & c_in;
assign net3 = a & b;
assign s = net1 ^ c_in;
assign c_out = net2 | net3;
endmodule
```

```
//Four Bit Full Adder_Subtractor
module FBFA_S (ADD_SUB,C_OUT,A,B,K);
input [3:0]A;
input [3:0]B;
input K;
output [3:0]ADD_SUB;
output C_OUT;
wire [2:0]C;
wire [3:0]N;
assign N[0]= B[0] ^ K;
assign N[1]= B[1] ^ K;
assign N[2]= B[2] ^ K;
assign N[3]= B[3] ^ K;
full_adder fa0(.a(A[0]),.b(N[0]),.c_in(K),.s(ADD_SUB[0]),.c_out(C[0]));
full_adder fa1(.a(A[1]),.b(N[1]),.c_in(C[0]),.s(ADD_SUB[1]),.c_out(C[1]));
full_adder fa2(.a(A[2]),.b(N[2]),.c_in(C[1]),.s(ADD_SUB[2]),.c_out(C[2]));
full_adder fa3(.a(A[3]),.b(N[3]),.c_in(C[2]),.s(ADD_SUB[3]),.c_out(C_OUT));
endmodule
```

//Testbench for Four Bit Full Adder_Subtractor

```
module FBFA_S_TB;
reg [3:0]A;
reg [3:0]B;
reg K;
wire [3:0]ADD_SUB;
wire C_OUT;
FBFA_S uut (.ADD_SUB(ADD_SUB),.C_OUT(C_OUT),.A(A),.B(B),.K(K));
initial begin
A= 4'b1000;B= 4'b0010;K =1'b0;#100;
A= 4'b1000;B= 4'b0010;K =1'b1;#100;
A= 4'b1100;B= 4'b1000;K =1'b0;#100;
A= 4'b1100;B= 4'b1000;K =1'b1;#100;
A= 4'b1000;B= 4'b0100;K =1'b0;#100;
A= 4'b1000;B= 4'b0100;K =1'b1;#100;
A= 4'b1010;B= 4'b1000;K =1'b0;#100;
```

```
A= 4'b1010;B= 4'b1000;K =1'b1;#100;
A= 4'b1000;B= 4'b0010;K =0'b0;#100;
A= 4'b1000;B= 4'b0010;K =0'b1;#100;
A= 4'b1100;B= 4'b1000;K =0'b0;#100;
A= 4'b1100;B= 4'b1000;K =0'b1;#100;
A= 4'b1000;B= 4'b0100;K =0'b0;#100;
A= 4'b1000;B= 4'b0100;K =0'b1;#100;
A= 4'b1010;B= 4'b1000;K =0'b0;#100;
A= 4'b1010;B= 4'b1000;K =0'b1;#100;
end
initial$monitor($time,"---A=%b,B=%b,K=%b,ADD_SUB=%b,C_OUT=%b,",A, B, K, ADD_SUB,
C_OUT);
endmodule
```

Aim 2: Development of Verilog modules for a BCD adder.

BCD adder

Let us begin with an explanation:

BCD stand for binary coded decimal. Suppose we have two 4-bit numbers A and B. The value of A and B can vary from 0(0000 in binary) to 9(1001 in binary) because we are considering decimal numbers.

Now if we consider an example:

Let A = 0111(7) and B = 1000 (8) be two inputs

When we add both we get Y=A+B as

Y = The value of binary sum will be 1111(=15).

But the BCD sum will be Y= 1 0101,

where 1 is 0001 in binary and 5 is 0101 in binary.

Now if we consider another example:

Let A = 0010(2) and B = 0100(4) be two inputs

When we add both we get Y=A+B as

Y = The value of binary sum will be 0110(= 6).

But the BCD sum will be Y= 0 0110,

where 0 is 0000 in binary and 6 is 0110 in binary.

Here we will use a behavioural model to demonstrate this.

1. Let A, B be 4-bit input and Cin be the 1-bit input.
2. Assign a 4-bit variable temp and compute temp =A+B+cin;

3. If value of temp is greater than 9 then perform temp = temp+6, since temp is 4-bit it will generate a carry/overflow. Store this carry in a 1-bit variable carry as 1 and store the temp value in a 4-bit variable sum.
4. Else if value of temp is not greater than 9 then store the temp value in a 4-bit variable sum and carry in a 1-bit variable carry as 0.

//BCD adder using behavioural model

```
module bcd_add (sum,carry,A,B,cin);
input [3:0]A;
input [3:0]B;
input cin;
output[3:0]sum;
output carry;
reg [3:0]sum;
reg carry;
reg [3:0]temp;
always @(A or B or cin)begin
temp =A+B+cin;
if (temp > 9)begin
temp =temp+6;
carry=1;
sum =temp;
end
else begin
carry =0;
sum =temp;
end
end
endmodule
```

//Testbench for BCD adder using behavioural model

```
module bcd_add_TB;
reg [3:0]A;
reg [3:0]B;
reg cin;
wire [3:0]sum;
```

```
wire carry;
bcd_add uut (.sum(sum),.carry(carry),.A(A),.B(B),.cin(cin));
initial begin
A = 0;  B = 0;  cin = 0;   #100;
A = 6;  B = 9;  cin = 0;   #100;
A = 3;  B = 3;  cin = 1;   #100;
A = 4;  B = 5;  cin = 0;   #100;
A = 8;  B = 2;  cin = 0;   #100;
A = 9;  B = 9;  cin = 1;   #100;
end
initial$monitor($time,"----A=%d,B=%d,C_IN=%b,BCD_ADD=%d,CARRY=%b,",A,B,cin,sum,
carry);
endmodule
```

Experiment-6: Magnitude Comparator Development of Verilog modules for a 4-bit magnitude comparator

Aim: Development of Verilog modules for a 4-bit magnitude comparator

4-Bit Magnitude Comparator –

A comparator used to compare two binary numbers each of four bits is called a 4-bit magnitude comparator. It consists of eight inputs each for two four-bit numbers and three outputs to generate less than, equal to and greater than between two binary numbers.

//4 bit Magnitude comparator using behavioural modelling

```
module mag_comp (A,B,larger,equal,lesser);
input [3:0]A;
input [3:0]B;
output reg larger;
output reg equal;
output reg lesser;
always @(A or B)begin
if (A > B)begin
larger = 1'b1;
equal = 1'b0;
lesser = 1'b0;
end
else if (A < B)begin
larger = 1'b0;
equal = 1'b0;
lesser = 1'b1;
end
else begin
larger = 1'b0;
equal = 1'b1;
lesser = 1'b0;
end
end
endmodule
```

//Testbench for 4 bit Magnitude comparator using behavioural modelling

```
module mag_comp_TB;
reg [3:0]A;
reg [3:0]B;
wire larger;
wire equal;
wire lesser;
mag_comp uut (.A(A),.B(B),.larger(larger),.equal(equal),.lesser(lesser));
initial begin
A= 4'b1000;B= 4'b0010;#100;
A= 4'b1000;B= 4'b1010;#100;
A= 4'b1100;B= 4'b1100;#100;
A= 4'b1100;B= 4'b1000;#100;
A= 4'b1000;B= 4'b1000;#100;
A= 4'b1000;B= 4'b0110;#100;
A= 4'b1010;B= 4'b1110;#100;
A= 4'b1010;B= 4'b1010;#100;
end
initial$monitor($time,"----A=%b,B=%b,LARGER=%b,EQUAL=%b,LESSER=%b",A,B,larger,
equal, lesser);
endmodule
```