Combinational Logic Design using VHDL

Brihat Ratna Bajracharya

Department of Electronics and Computer Engineering, IOE Central Campus, Pulchowk
Lalitpur, Nepal

070bct513@ioe.edu.np

***Abstract*—VHDL is a language for describing digital electronic systems. VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design, i.e. how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.**

I. Introduction

VHDL stands for VHSIC (Very High Speed Integrated Circuit) Hardware Descriptive Language. It is used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits. VHDL can also be used as a general programming language. VHDL was originally developed at the behest of the US Department of Defence in order to document the behaviour of the ASICs that supplier companies were including in equipment. VHDL is influenced from Ada and Pascal. The initial release of VHDL was designed as per IEEE 1076 standard in 1987. The most commonly used VHDL version supported by CAD tools is IEEE 1076 1993.

A. Features of VHDL

- It is a hardware descriptive language used for design entry and simulation of digital circuits.
- It is an event-driven language: i.e. whenever an event occurs on signals in VHDL, it triggers the execution of a statement.
- It allows both concurrent as well as sequential modelling.
- It gives the flexibility to define data types that are specific to user needs apart from predefined types.
- It supports code reusability and code sharing via packages and user defined libraries.
- It is case-insensitive i.e. it does not differentiate between lowercase and uppercase letters.
- It is strongly typed language i.e. it does not support implicit conversion between data types.

B. Levels of representation and abstraction

A digital system can be represented at different levels of abstraction. This keeps the description and design of complex systems manageable.

1) Behavioral – The highest level of abstraction that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level.

2) Structural – The structural level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system.

C. Lexical Elements

1) Comments – VHDL comments start with two adjacent hyphens ('--') and extend to the end of the line. They have no part in the meaning of a VHDL description.

2) Identifiers – Identifiers in VHDL are used as reserved words and as programmer defined names. They must conform to the rule:

```
letter { [underscore] letter_or_digit}
```

Identifiers are case insensitive but underscore characters in identifiers are significant.

3) Numbers – Literal numbers may be expressed either in decimal or in a base between two and sixteen. If the literal includes a point, it represents a real number, otherwise it represents an integer. Decimal literals are defined by:

```
dec_literal ::= integer[.integer][exponent]
integer ::= digit {[underline]digit}
exponent ::= E [+] integer | E - integer
```

The base and the exponent are expressed in decimal. The exponent indicates the power of the base by which the literal is multiplied. The letters A to F (upper or lower case) are used as extended digits to represent 10 to 15.

4) Characters – Literal characters are formed by enclosing an ASCII character in single-quote marks (' ').

5) Strings – Literal strings of characters are formed by enclosing the characters in double-quote marks (" "). To include a double-quote mark itself in a string, a pair of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters.

6) Bit Strings – VHDL provides a convenient way of specifying literal values for arrays of type bit ('0's and '1's). The syntax is:

```
bit_string_literal ::= base_spec "bit_val"
base_spec ::= B | O | X
bit_val ::= ext_dig {[underscore] ext_dig}
```

Base specifier `B` stands for binary, `O` for octal and `X` for hexadecimal.

D. Expressions and Operators

An expression is a formula combining primaries with operators. Primaries include names of objects, literals, function calls and parenthesized expressions.

The logical operators **and**, **or**, **nand**, **nor**, **xor** and **not** operate on values of type **bit** or **boolean**, and also on one-dimensional arrays of these types. For array operands, the operation is applied between corresponding elements of each array, yielding an array of the same length as the result. For **bit** and **boolean** operands, **and**, **or**, **nand**, and **nor** are 'short-circuit' operators, i.e. they only evaluate their right operand if the left operand does not determine the result. So **and** and **nand** only evaluate the right operand if the left operand is true or '1', and **or** and **nor** only evaluate the right operand if the left operand is false or '0'.

The **relational operators** (=, /=, <, <=, > and >=) must have both operands of the same type, and yield **boolean** results. The **equality operators** (= and /=) can have operands of any type. For composite types, two values are equal if all of their corresponding elements are equal. The remaining operators must have operands which are scalar types or one-dimensional arrays of discrete types.

The **sign operators** (+ and –) and the **addition** (+) and **subtraction** (–) operators have their usual meaning on numeric operands. The **concatenation operator** (&) operates on one-dimensional arrays to form a new array with the contents of the right operand following the contents of the left operand. It can also concatenate a single new element to an array, or two individual elements to form an array. The concatenation operator is most commonly used with strings.

The **multiplication** (*) and **division** (/) operators work on integer, floating point and physical types. The **modulus** (mod) and **remainder** (rem) operators only work on integer types. The **absolute value** (abs) operator works on any numeric type. Finally, the **exponentiation** (**) operator can have an integer or floating point left operand, but must have an integer right operand. A negative right operand is only allowed if the left operand is a floating point number.

E. Sequential Statements

VHDL contains a number of facilities for modifying the state of objects and controlling the flow of execution of models.

1) Variable Assignment – As in other programming languages, a variable is given a new value using an assignment statement. The syntax is:

```
target := expression;
target ::= name | aggregate
```

In the simplest case, the target of the assignment is an object name, and the value of the expression is given to the named object. The object and the value must have the same base type. If the target of the assignment is an aggregate, then the elements listed must be object names, and the value of the expression must be a composite value of the same type as the aggregate.

2) If Statement – The if statement allows selection of statements to execute depending on one or more conditions. The syntax is:

```
if condition then
   sequence_of_statements
{ elsif condition then
   sequence_of_statements }
[ else
   sequence_of_statements ]
end if;
```

The conditions are expressions resulting in boolean values. The conditions are evaluated successively until one found that yields the value true. In that case the corresponding statement list is executed. Otherwise, if the else clause is present, its statement list is executed.

3) Case Statement – The case statement allows selection of statements to execute depending on the value of a selection expression. The syntax is:

```
case expression is
   case_statement_alternative
   { case_statement_alternative }
end case;
case_statement_alternative ::=
   when choices =>
   sequence_of_statements
choices ::= choice { | choice }
choice ::=
   simple_expression
   | discrete_range
   | element_simple_name
   | others
```

The selection expression must result in either a discrete type, or a one-dimensional array of characters. The alternative whose choice list includes the value of the expression is selected and the statement list executed.

4) Loop Statements – VHDL has a basic loop statement, which can be augmented to form the usual while and for loops seen in other programming languages. The syntax of the loop statement is:

```
[ loop_label : ]
   [ iteration_scheme ] loop
     sequence_of_statements
   end loop [ loop_label ] ;
iteration_scheme ::=
   while condition
   | for loop_parameter_specification
parameter_specification ::=
   identifier in discrete_range
```

If the iteration scheme is omitted, we get a loop which will repeat the enclosed statements indefinitely. An example of such a basic loop is:

```
loop
   do_something;
end loop;
```

The while iteration scheme allows a test condition to be evaluated before each iteration. The iteration only proceeds if the test evaluates to true. If the test is false, the loop statement terminates. An example:

```
while index < length and str(index)/=' ' loop
   index := index + 1;
end loop;
```

The for iteration scheme allows a specified number of iterations. The loop parameter specification declares an object which takes on successive values from the given range for each iteration of the loop. Within the statements enclosed in the loop, the object is treated as a constant, and so may not be assigned to. The object does not exist beyond execution of the loop statement. An example:

```
for item in 1 to last_item loop
   table(item) := 0;
end loop;
```

5) Null Statement – The null statement has no effect. It may be used to explicitly show that no action is required in certain cases. It is most often used in case statements, where all possible values of the selection expression must be listed as choices, but for some choices no action is required.

6) Assertions – An assertion statement is used to verify a specified condition and to report if the condition is violated. The syntax is:

```
assert condition
   [ report expression ]
   [ severity expression ];
```

If the report clause is present, the result of the expression must be a string. This is a message which will be reported if the condition is false. If it is omitted, the default message is "Assertion violation". If the severity clause is present the expression must be of the type severity_level. If it is omitted, the default is error. A simulator may terminate execution if an assertion violation occurs and the severity value is greater than some implementation dependent threshold. Usually the threshold will be under user control.

F. VHDL Structure

1) Entity Declaration

The entity declaration defines the **name** of the entity and lists the input and output ports. The general form is:

```
entity NAME_OF_ENTITY is
   port (signal_names: mode type;
         signal_names: mode type;
          :             :    :
         signal_names: mode type);
end [NAME_OF_ENTITY] ;
```

mode is one of **in**, **out**, **buffer**, **inout** that indicate the signal direction and **type** is a built-in or user-defined signal type. Some signal types are **bit**, **bit_vector**, **std_logic**, **std_ulogic**, **std_logic_vector**, **time**, **real**, **std_ulogic_vector**, **boolean**, **integer**, **character.**

2) Architecture declaration

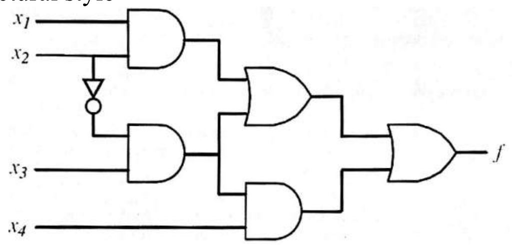
The architecture body specifies how the circuit operates and how it is implemented. The architecture body looks as follows:

```
architecture arch_name of entity is
   -- Declarations
   -- components declarations
   -- signal declarations
   -- constant declarations
   -- type declarations
begin
   -- Statements
   :
end architecture_name;
```

II. Activity I

Write VHDL code to implement the logic circuit shown in figure, which has 4 inputs (x1, x2, x3, x4) and one output (f). Provide the following architectural styles:

Dataflow style
Behavioral style
Structural style



Write a VHDL test bench to verify the operation of the logic circuit. Provide a simulation waveform depicting all possible input cases

(a) Dataflow style

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY small_ckt IS PORT (
    A, B, C, D: IN STD_LOGIC;
    F: OUT STD_LOGIC
);
END small_ckt;

ARCHITECTURE dataflow OF small_ckt IS
BEGIN
    F <= (((A AND B) OR (NOT B AND C)) OR
          ((NOT B AND C) AND D));
END dataflow;
```

(b) Behavioral style

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY small_ckt IS PORT (
    A, B, C, D: IN STD_LOGIC;
    F: OUT STD_LOGIC
);
END small_ckt;

ARCHITECTURE behavorial OF small_ckt IS
SIGNAL F1, F2, F3, F4: STD_LOGIC;
```

```
BEGIN
example: PROCESS(A,B,C,D,F1,F2,F3,F4)

BEGIN
    F1 <= A AND B;
    F2 <= NOT B AND C;
    F3 <= F1 OR F2;
    F4 <= F2 AND D;
    F <= F3 OR F4;
END PROCESS example;

END behavorial;
```

(c) Structural style

[and1.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY and1 IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END and1;

ARCHITECTURE dataflow OF and1 IS
BEGIN
    o1 <= i1 AND i2;
END dataflow;
```

[or1.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY or1 IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END or1;

ARCHITECTURE dataflow OF or1 IS
BEGIN
    o1 <= i1 OR i2;
END dataflow;
```

[andnot.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY andnot IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END andnot;

ARCHITECTURE dataflow OF andnot IS
BEGIN
    o1 <= NOT i1 AND i2;
END dataflow;
```

[small_ckt_structural.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY small_ckt IS PORT (
    A, B, C, D: IN STD_LOGIC;
    F: OUT STD_LOGIC
);
END small_ckt;

ARCHITECTURE structural OF small_ckt IS
    SIGNAL F1, F2, F3, F4: STD_LOGIC;
COMPONENT and1 IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END COMPONENT;

COMPONENT andnot IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END COMPONENT;

COMPONENT or1 IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END COMPONENT;

BEGIN
    C1: and1 PORT MAP (i1 => A, i2 => B,
         o1 => F1);
    C2: andnot PORT MAP (i1 => B, i2 =>
         C, o1 => F2);
    C3: or1 PORT MAP (i1 => F1, i2 => F2,
         o1 => F3);
    C4: and1 PORT MAP (i1 => F2, i2 => D,
         o1 => F4);
    C5: or1 PORT MAP (i1 => F3, i2 => F4,
         o1 => F);
END structural;
```

(d) Test bench

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY small_ckt_tb IS
END small_ckt_tb;

ARCHITECTURE behav OF small_ckt_tb IS
COMPONENT small_ckt
PORT(
    A, B, C, D: IN STD_LOGIC;
    F: OUT STD_LOGIC
    );
END COMPONENT;
```

```
    SIGNAL                    input_vector:
STD_LOGIC_VECTOR( 3 DOWNTO 0)  := "0000";
    SIGNAL output: STD_LOGIC;

BEGIN
    uut: small_ckt PORT MAP(
          A => input_vector(3),
          B => input_vector(2),
          C => input_vector(1),
          D => input_vector(0),
          F => output
    );

    stim_proc: PROCESS
    BEGIN
          FOR index IN 0 TO 15 LOOP
                input_vector          <=
std_logic_vector(to_unsigned(index,4));
                WAIT FOR 50 ns;
          END LOOP;
    END PROCESS;
END behavioral;
```

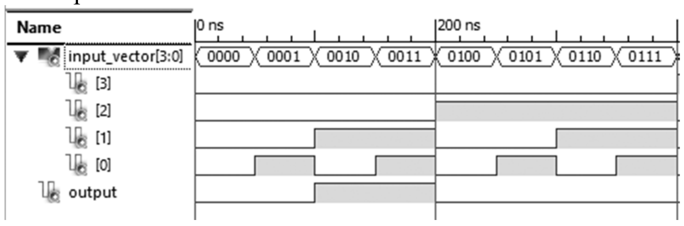
Discussion:

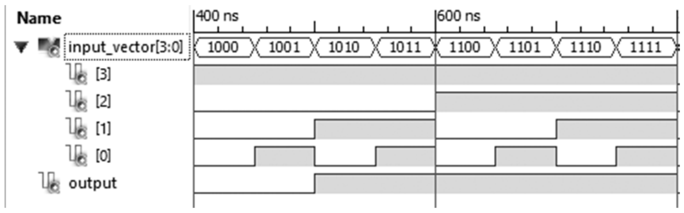
Here, we need to implement given logic circuit in VHDL. In dataflow style, we simply created four input signal along with single output and by using logical operators we simply wrote the expression to produce output.

Behavior style is also similarly done. Only difference in behavioural style is that we called a process to execute the operation and to produce the output.

In structural style, we first created three components namely, **and1**, **or1** and **andnot**. Then these components are used to make structure of the logic circuit. While writing structure PORT MAP is used to connect signals to the port. Components are used to perform each operation separately before producing the output.

Output:





III. Activity II

Write VHDL code to design a logic circuit that implements the truth table of BCD-to-Gray code converter. Use Karnaugh maps to simplify the output functions. Provide the following architectural styles:

Dataflow style
Behavioral style
Structural style

TABLE – I
TRUTH TABLE OF BCD-TO-GRAY CODE CONVERTER

BCD Numbers				Gray Code Numbers			
X3	X2	X1	X0	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
:	:	:	:	:	:	:	:
1	1	1	1	1	0	0	0

Write a VHDL test bench to verify the operation of the logic circuit. Provide a simulation waveform depicting all possible input cases.

(a) Karnaugh Map Simplification

X1X0 →

X3X2↓	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

k-map for Y3
Y3 = X3

X1X0 →

X3X2↓	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

k-map for Y2
Y2 = X3 XOR X2

X1X0 →

X3X2↓	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

k-map for Y1
Y1 = X2 XOR X1

X1X0 →

X3X2↓	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

k-map for Y2
Y0 = X1 XOR X0

(b) Dataflow style

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bcd_gray IS PORT (
    X3, X2, X1, X0: IN STD_LOGIC;
    Y3, Y2, Y1, Y0: OUT STD_LOGIC
);
END bcd_gray;

ARCHITECTURE dataflow OF bcd_gray IS

BEGIN
    Y3 <= X3;
    Y2 <= X2 XOR X3;
    Y1 <= X1 XOR X2;
    Y0 <= X0 XOR X1;
END dataflow;
```

(c) Behavioral style

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bcd_gray IS PORT (
    X3, X2, X1, X0: IN STD_LOGIC;
    Y3, Y2, Y1, Y0: OUT STD_LOGIC
);
END bcd_gray;

ARCHITECTURE behavorial OF bcd_gray IS

BEGIN
PROCESS(X3,X2,X1,X0)

BEGIN
    Y3 <= X3;
    Y2 <= X2 XOR X3;
    Y1 <= X1 XOR X2;
    Y0 <= X0 XOR X1;
END PROCESS

END behavorial;
```

(d) Structural style (using only NOR gates)
[xor_nor.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor_nor IS PORT (
    a, b: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END xor_nor;

ARCHITECTURE behavorial OF xor_nor IS
SIGNAL nota, notb, xnorab: STD_LOGIC;

BEGIN
xor_nor: PROCESS (a,b,nota,notb,xnorab)

BEGIN
    nota <= a NOR a;
    notb <= b NOR b;
    xnorab <= (a NOR notb) NOR
              (nota NOR b);
    o <= xnorab NOR xnorab;

END PROCESS xor_nor;

END behavorial;
```

[bcd_gray_structural.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bcd_gray IS PORT (
    X3, X2, X1, X0: IN STD_LOGIC;
    Y3, Y2, Y1, Y0: OUT STD_LOGIC
);
END bcd_gray;

ARCHITECTURE structural OF bcd_gray IS

COMPONENT xor_nor IS PORT (
    a, b: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END COMPONENT;

BEGIN
    C0: xor_nor PORT MAP (a => X3, b =>
         '0', o => Y3);
    C1: xor_nor PORT MAP (a => X3, b =>
         X2, o => Y2);
    C2: xor_nor PORT MAP (a => X2, b =>
         X1, o => Y1);
    C3: xor_nor PORT MAP (a => X1, b =>
         X0, o => Y0);

END structural;
```

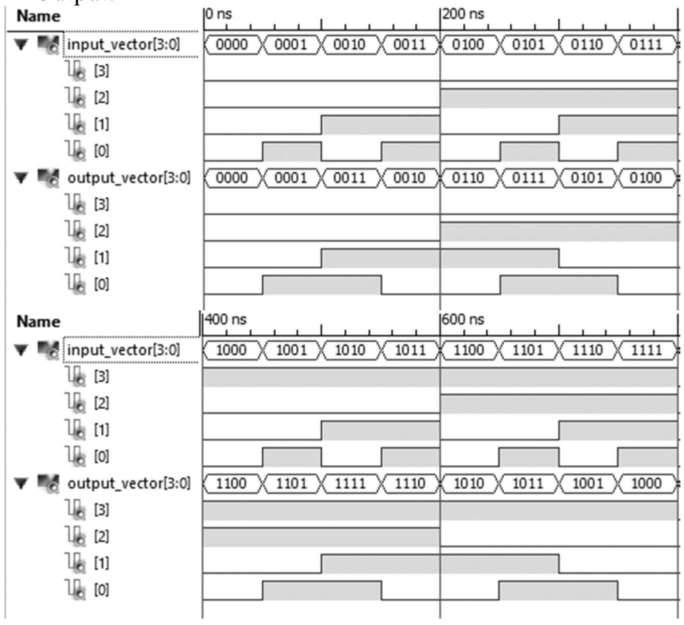
(e) Test Bench

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY bcd_gray_tb IS
END bcd_gray_tb;

ARCHITECTURE behavioral OF bcd_gray_tb IS
    COMPONENT bcd_gray
    PORT(
          X3, X2, X1, X0: IN STD_LOGIC;
          Y3, Y2, Y1, Y0: OUT STD_LOGIC;
    );
    END COMPONENT;

    SIGNAL                 input_vector:
STD_LOGIC_VECTOR( 3 DOWNTO 0) := "0000";
    SIGNAL                output_vector:
STD_LOGIC_VECTOR( 3 DOWNTO 0) := "0000";

BEGIN
    uut: bcd_gray PORT MAP(
            X3 => input_vector(3),
            X2 => input_vector(2),
            X1 => input_vector(1),
            X0 => input_vector(0),

            Y3 => output_vector(3),
            Y2 => output_vector(2),
            Y1 => output_vector(1),
            Y0 => output_vector(0)
        );

        stim_proc: PROCESS

        BEGIN
            FOR index IN 0 TO 15 LOOP
                input_vector          <=
    std_logic_vector(to_unsigned(index,4));
                WAIT FOR 50 ns;
            END LOOP;
        END PROCESS;
    END behavioral;
```

Discussion:

Second activity is a BCD to Gray code converter. Using the given truth table (though incomplete), expression for Y3, Y2, Y1, Y0 is calculated using Karnaugh simplification. This expression was used directly in dataflow as well as behavioural style. But in case of structural style, we were required to use NOR gate for each operation. So, we created XOR gate using NOR gate only and used it in structural description of the converter.

Output:



IV. Activity III

Write VHDL code to implement the logic function (f) with three input variables x1, x2, x3. The function (f) is equal to 1 if and only if two variables are equal to 1; otherwise, it is equal to zero. Draw a truth table for the function (f), and use Karnaugh maps to simplify. Provide the following architectural styles:

Dataflow style

Behavioral style

Structural style (using only NAND gates)

Write a VHDL test bench to verify the operation of the logic circuit. Provide a simulation waveform depicting all possible input cases.

(a) Truth Table

TABLE – II

Truth Table for Activity III

Input			Output
X2	X1	X0	Y3
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) Karnaugh Map Simplification

X1X0 →

X2↓	00	01	11	10
0	0	0	1	0
1	0	1	0	1

k-map for Y

Y = X1X2X3’ + X1X2’X3 + X1’X2X3

(c) Dataflow style

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY func IS PORT (
    X1, X2, X3: IN STD_LOGIC;
    F: OUT STD_LOGIC
);
END func;

ARCHITECTURE dataflow OF func IS
BEGIN
    F <= (X1 AND X2 AND NOT X3) OR (X3
          AND (X1 XOR X2));
END dataflow;
```

(d) Behavioral style

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY func IS PORT (
    X1, X2, X3: IN STD_LOGIC;
    F: OUT STD_LOGIC
);
```

```
END func;

ARCHITECTURE behavioral OF func IS
SIGNAL A1, A2, A3: STD_LOGIC;

BEGIN
PROCESS (X1,X2,X3,A1,A2,A3)

BEGIN
    A1 <= X1 AND X2 AND NOT X3;
    A2 <= X1 XOR X2;
    A3 <= A2 AND X3;
    F <= A1 OR A3;
END PROCESS;

END behavioral;
```

(e) Structural style

[func_and1.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY and1 IS PORT (
    a, b, c: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END and1;

ARCHITECTURE behavorial OF and1 IS
SIGNAL F1, F2, F3, F4: STD_LOGIC;

BEGIN

PROCESS (a,b,c,F1,F2,F3,F4)
BEGIN
    F1 <= a NAND a;
    F2 <= F1 NAND b;
    F3 <= F2 NAND F2;
    F4 <= F3 NAND c;
    o <= F4 NAND F4;
END PROCESS;

END behavorial;
```

[func_or1.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY or1 IS PORT (
    a, b, c: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END or1;

ARCHITECTURE behavorial OF or1 IS
SIGNAL G1, G2, G3, G4, G5: STD_LOGIC;

BEGIN
PROCESS (a,b,c,G1,G2,G3,G4,G5)
BEGIN
    G1 <= a NAND a;
    G2 <= b NAND b;
    G3 <= G1 NAND G2;
    G4 <= G3 NAND G3;
    G5 <= c NAND c;
    o <= G4 NAND G5;
END PROCESS;

END behavorial;
```

[func_structural.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY func IS PORT (
    X3, X2, X1: IN STD_LOGIC;
    F: OUT STD_LOGIC
);
END func;

ARCHITECTURE structural OF func IS
    SIGNAL A1, A2, A3: STD_LOGIC;

    COMPONENT and1 IS PORT (
          a, b, c: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

    COMPONENT or1 IS PORT (
          a, b, c: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

    BEGIN
          N1: and1 PORT MAP (a => X1, b
              => X2, c => X3, o => A1);
          N2: and1 PORT MAP (a => X2, b
              => X3, c => X1, o => A2);
          N3: and1 PORT MAP (a => X3, b
              => X1, c => X2, o => A3);
          N4: or1 PORT MAP (a => A1, b =>
              A2, c => A3, o => F);

END structural;
```

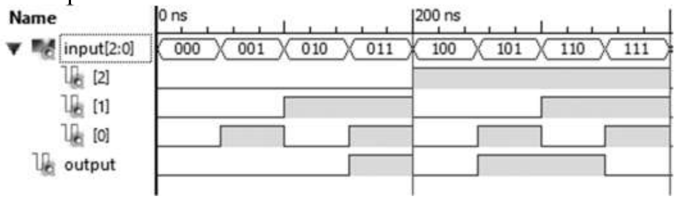
(f) Test Bench

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY func_tb IS
END func_tb;

ARCHITECTURE behavioral OF func_tb IS
    COMPONENT func
    PORT (
            X1, X2, X3: IN STD_LOGIC;
            F: OUT STD_LOGIC
    );
    END COMPONENT;

    SIGNAL   input:   STD_LOGIC_VECTOR(2
DOWNTO 0):= "000";
    SIGNAL output: STD_LOGIC :='0';

    BEGIN
            uut: func PORT MAP (
                    X3 => input(2),
                    X2 => input(1),
                    X1 => input(0),
                    F => output
            );

    stim_proc: PROCESS
BEGIN
    FOR index IN 0 TO 7 LOOP
            input    <=    STD_LOGIC_VECTOR
(TO_UNSIGNED(index,3));
            WAIT FOR 50 ns;
    END LOOP;
END PROCESS;

END behavioral;
```

Discussion:

This activity is similar to previous activity. Before proceeding, we need to make truth table for given problem for all possible cases which we did and then used the expression generated from the truth table in dataflow and behavioural style. In structural style, three basic gates AND, OR, and NOT were made using NAND gate as instructed. NOT gate was not separately made but calculated inside AND and OR gate. These components were then used in structural description of the logic circuit.

Output:



V. Activity IV

Write VHDL code to implement the implicit sum of products (SOP) and product of sum (POS) logic functions

F1(x1,x2,x3,x4) = ∑ (m0,m1,m4,m5,m8,m9,m14,m15)

F2(x1,x2,x3,x4) = ∏ (M0,M1,M5,M8,M9,M13,M15)

Draw the truth tables for the functions, and use Karnaugh maps to simplify. Provide the following architectural styles:

Dataflow style

Behavioral style

Structural style (using only NAND gates)

Write a VHDL test bench to verify the operation of the logic circuit. Provide a simulation waveform depicting all possible input cases.

(a) Truth Table

TABLE – III

Truth Table for Activity IV

Input				Output	
X3	X2	X1	X0	F1	F2
0	0	0	0	1	0
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	1	1
0	1	0	1	1	0
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	0	1	0	0
1	1	1	0	1	1
1	1	1	1	1	0

(b) Karnaugh Map Simplification

X1X0 →

X3X2↓	**00**	**01**	**11**	**10**
00	1	1	0	0
01	1	1	0	0
11	0	0	1	1
10	1	1	0	0

k-map for F1

F1 = X1'X3' + X2'X3' + X1X2X3

X1X0 →

X3X2↓	**00**	**01**	**11**	**10**
00	0	0	1	1
01	1	0	1	1
11	1	0	0	1
10	0	0	1	1

k-map for F2

F2 = (X3 + X4') (X2 + X3) (X1'+X2'+X4')

(a) Dataflow style (for F1)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY sop IS PORT (
    X1, X2, X3, X4: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END sop;

ARCHITECTURE dataflow OF sop IS

BEGIN
    Y <= (NOT X1 AND NOT X3) OR (NOT X2
      AND NOT X3) OR (X1 AND X2 AND X3);
```

```
END dataflow;
```

(b) Behavioral style (for F1)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY sop IS PORT (
    X1, X2, X3, X4: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END sop;

ARCHITECTURE behavorial OF sop IS

BEGIN

PROCESS(X1,X2,X3,X4)

BEGIN
    Y <= (NOT X1 AND NOT X3) OR (NOT X2
      AND NOT X3) OR (X1 AND X2 AND X3);

END PROCESS;

END behavorial;
```

(c) Structural style (using only NAND gates) (for F1)
[sop_and.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY and1 IS PORT (
    a, b: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END and1;

ARCHITECTURE behavorial OF and1 IS
SIGNAL F: STD_LOGIC;

BEGIN

PROCESS (a,b,F)

BEGIN
    F <= a NAND b;
    o <= F NAND F;

END PROCESS;

END behavorial;
```

[sop_or.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY or1 IS PORT (
    a,b: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END or1;

ARCHITECTURE behavorial OF or1 IS
SIGNAL G1,G2: STD_LOGIC;

BEGIN

PROCESS (a,b,G1,G2)

BEGIN
    G1 <= a NAND a;
    G2 <= b NAND b;
    o <= G1 NAND G2;
END PROCESS;

END behavorial;
```

[sop_not.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY not1 IS PORT (
    a: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END not1;

ARCHITECTURE behavorial OF not1 IS

BEGIN

PROCESS (a)
BEGIN
    o <= a NAND a;
END PROCESS;

END behavorial;
```

[sop_structural.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY sop IS PORT (
    X1, X2, X3, X4: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END sop;

ARCHITECTURE structural OF sop IS
    SIGNAL NX1, NX2, NX3, F: STD_LOGIC;
    SIGNAL I1, I2, I3, I4: STD_LOGIC;

    COMPONENT not1 IS PORT (
          a: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;
    COMPONENT and1 IS PORT (
          a, b: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

    COMPONENT or1 IS PORT (
          a, b: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

BEGIN
    N1: not1 PORT MAP (a => X1, o => NX1);
    N2: not1 PORT MAP (a => X2, o => NX2);
    N3: not1 PORT MAP (a => X3, o => NX3);

    A1: and1 PORT MAP (a => NX1, b => X3,
         o => I1);
    A2: and1 PORT MAP (a => NX2, b =>
         NX3, o => I2);
    A3: and1 PORT MAP (a => X1, b => X2,
         o => I3);
    A4: and1 PORT MAP (a => I3, b => X3,
         o => I4);

    O1: or1 PORT MAP (a => I1, b => I2,
         o => F);
    O2: or1 PORT MAP (a => F, b => I4, o
         => Y);

END structural;
```

(d) Dataflow style (for F2)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY pos IS PORT (
    X1, X2, X3, X4: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END pos;

ARCHITECTURE dataflow OF pos IS

BEGIN
    Y <= (X3 OR NOT X4) AND (X2 OR X3)
     AND (NOT X1 OR NOT X2 OR NOT X4);
END dataflow;
```

(e) Behavioral style (for F2)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY pos IS PORT (
    X1, X2, X3, X4: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END pos;

ARCHITECTURE behavorial OF pos IS

BEGIN

PROCESS(X1,X2,X3,X4)
BEGIN
    Y <= (X3 OR NOT X4) AND (X2 OR X3)
     AND (NOT X1 OR NOT X2 OR NOT X4);
END PROCESS;

END behavorial;
```

(f) Structural style (using only NAND gates) (for F2)
[pos_and.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY and1 IS PORT (
    a, b: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END and1;

ARCHITECTURE behavorial OF and1 IS
SIGNAL F: STD_LOGIC;

BEGIN

PROCESS (a,b,F)
BEGIN
    F <= a NAND b;
    o <= F NAND F;
END PROCESS;

END behavorial;
```

[pos_or.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY or1 IS PORT (
    a, b: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END or1;

ARCHITECTURE behavorial OF or1 IS
SIGNAL G1, G2: STD_LOGIC;

BEGIN

PROCESS (a,b,G1,G2)
BEGIN
    G1 <= a NAND a;
    G2 <= b NAND b;
    o <= G1 NAND G2;
END PROCESS;
END behavorial;
```

[pos_not.vhd]
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY not1 IS PORT (
    a: IN STD_LOGIC;
    o: OUT STD_LOGIC
);
END not1;

ARCHITECTURE behavorial OF not1 IS

BEGIN

PROCESS (a)
BEGIN
    o <= a NAND a;
END PROCESS;

END behavorial;
```

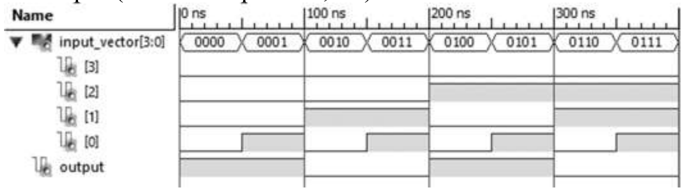
[pos_structural.vhd]
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY pos IS PORT (
    X1, X2, X3, X4: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END pos;

ARCHITECTURE structural OF pos IS
    SIGNAL NX1, NX2, NX4, F: STD_LOGIC;
    SIGNAL I1, I2, I3, I4: STD_LOGIC;

    COMPONENT not1 IS PORT (
          a: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

    COMPONENT and1 IS PORT (
          a, b: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

    COMPONENT or1 IS PORT (
          a, b: IN STD_LOGIC;
          o: OUT STD_LOGIC
    );
    END COMPONENT;

BEGIN
    N1: not1 PORT MAP (a => X1, o => NX1);
    N2: not1 PORT MAP (a => X2, o => NX2);
    N3: not1 PORT MAP (a => X4, o => NX4);

    O1: or1 PORT MAP (a => X3, b => NX4,
         o => I1);
    O2: or1 PORT MAP (a => X2, b => X3,
         o => I2);
    O3: or1 PORT MAP (a => NX1, b => NX2,
         o => I3);
    O4: or1 PORT MAP (a => I3, b => NX4,
         o => I4);

    A1: and1 PORT MAP (a => I1, b => I2,
         o => F);
    A2: and1 PORT MAP (a => F, b => I4,
         o => Y);

END structural;
```

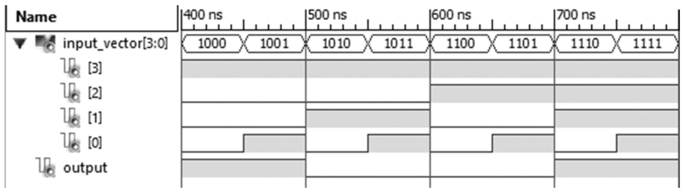
(g) Test Bench
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY sop_tb IS
END sop_tb;

ARCHITECTURE behavioral OF sop_tb IS
    COMPONENT sop
    PORT(
          X1, X2, X3, X4: IN STD_LOGIC;
          Y: OUT STD_LOGIC
    );
    END COMPONENT;

    SIGNAL                    input_vector:
STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
    SIGNAL output: STD_LOGIC:= '0';

BEGIN
    uut: sop PORT MAP(
          X1 => input_vector(3),
          X2 => input_vector(2),
          X3 => input_vector(1),
          X4 => input_vector(0),
          Y => output
    );

    stim_proc: PROCESS

    BEGIN
          FOR index IN 0 TO 15 LOOP
                input_vector          <=
std_logic_vector(to_unsigned(index,4));
                WAIT FOR 50 ns;
          END LOOP;
    END PROCESS;

END behavioral;
```

Discussion:

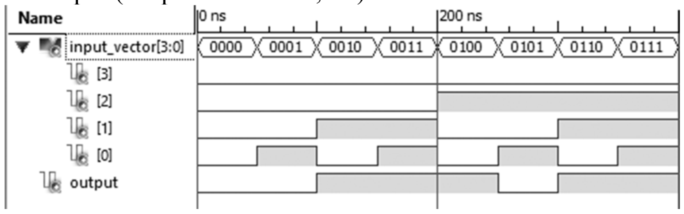
Instead of logic circuit, this question provided us the sum of product and product of sum function. In this activity, we first drew the truth table for each function and using Karnaugh Map, simplified to shorter expression. This expression was directly used in dataflow and behavioural style using logical operators. In structural style, basic gates were made as components using NAND gate only. These components were then used in structural description of the SOP and POS function.

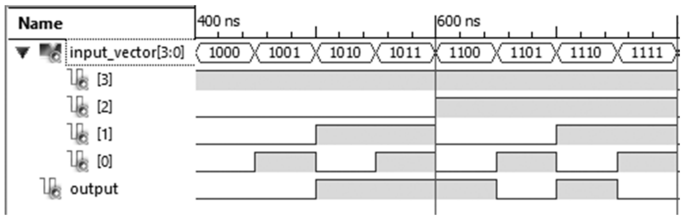
Output (for sum of product, F1):





Output (for product of sun, F2):





VI. ACTIVITY V

Write VHDL code to implement a 2:1 MUX having inputs x1 and x2, select line s and output y.

`TABLE – IV
TRUTH TABLE FOR 2:1 MULTIPLEXER

Select Line	Input Lines		Output Line
S	X2	X1	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Provide the following architectural implementation:
Using WITH-SELECT statement
Using WHEN-ELSE statement
Using IF-THEN-ELSE statement
Write a VHDL test bench to verify the operation of the logic circuit. Provide a simulation waveform depicting all possible input cases.

(a) Using WITH-SELECT statement

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux2to1 IS PORT (
    S, X1, X2: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END mux2to1;

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
    WITH (NOT S AND X1) OR (S AND X2)
    SELECT
          Y <=  '1' WHEN '1',
                '0' WHEN '0',
                '0' WHEN OTHERS;
END dataflow;
```

(b) Using WHEN-ELSE statement

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux2to1 IS PORT (
    S, X1, X2: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END mux2to1;

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
    Y <=  '1' WHEN ((NOT S AND X1) OR (S
           AND X2)) = '1' ELSE
                `      '0';
END dataflow;
```

(c) Using IF-THEN-ELSE statement

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux2to1 IS PORT (
    S, X1, X2: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END mux2to1;

ARCHITECTURE behavioral OF mux2to1 IS
BEGIN

PROCESS (S,X1,X2)
```

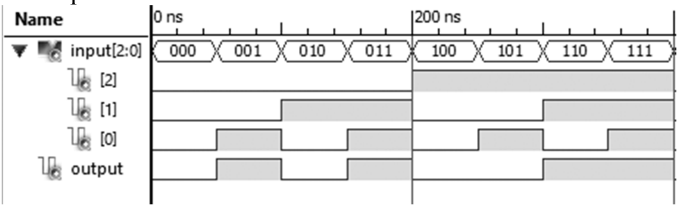
```
BEGIN
    IF ((NOT S AND X1) OR (S AND X2)) =
       '1' THEN Y <= '1';
    ELSE Y <= '0';
    END IF;
END PROCESS;

END behavioral;
```

(d) Test Bench

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY mux2to1_tb IS
END mux2to1_tb;

ARCHITECTURE behavioral OF mux2to1_tb IS
    COMPONENT mux2to1
    PORT (
          S, X1, X2: IN STD_LOGIC;
          Y: OUT STD_LOGIC
    );
    END COMPONENT;

    SIGNAL   input:   STD_LOGIC_VECTOR(2
DOWNTO 0):= "000";
    SIGNAL output: STD_LOGIC;

BEGIN
    uut: mux2to1 PORT MAP (
          S => input(2),
          X2 => input(1),
          X1 => input(0),
          Y => output
    );

stim_proc: PROCESS

BEGIN
    FOR index IN 0 TO 7 LOOP
          input <= STD_LOGIC_VECTOR
                (TO_UNSIGNED(index,3));
          WAIT FOR 50 ns;
    END LOOP;
END PROCESS;

END behavioral;
```

Discussion:
Designing a 2:1 MUX is similar to designing other logic circuits. By the help of truth table given, we can calculate the expression for output and then it can be used to implement it in VHDL. In this activity, we are required to use WITH-SELECT, WHEN-ELSE and IF-THEN-ELSE statement during design. For simplicity, we adapted dataflow style for first two statement and behavioural style for last statement. All outputs were verified by writing a test bench code for all possible cases of inputs.

Output:



VII. Activity VI

Write VHDL code to implement a 4-bit adder/subtracter using four 1-bit full adders

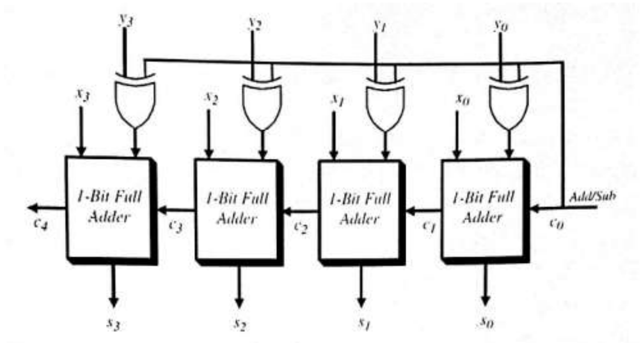


Fig. 4-bit adder/subtracter using 1-bit Full Adder

`TABLE – V
TRUTH TABLE FOR 1-BIT FULL ADDER

Input Bits			Output	
X	Y	Carry (Cin)	Carry (Cout)	Sum (S)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$\boldsymbol{S = X \oplus Y \oplus Cin}$ and $\boldsymbol{Cout = XY + (X \oplus Y)\ Cin}$

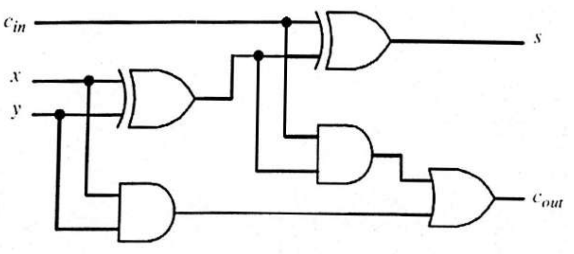


Fig. Logic Circuit for 1-bit Full Adder

Use structural architecture style with hierarchical design approach. Use 1-bit adder as the basic building block. Implement the 4-bit adder/subtracter using four 1-bit full adders. Write a VHDL test bench to verify the operation of the

4-bit adder/subtracter. Provide a simulation waveform depicting all possible input cases.

VHDL Code:
[xor1.vhd]
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY xor1 IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END xor1;

ARCHITECTURE dataflow OF xor1 IS
BEGIN
    o1 <= i1 XOR i2;
END dataflow;
```

[full_adder.vhd]
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY full_adder IS PORT (
    i1, i2, cin: IN STD_LOGIC;
    sum, cout: OUT STD_LOGIC
);
END full_adder;

ARCHITECTURE dataflow OF full_adder IS
BEGIN
    sum <= i1 XOR i2 XOR cin;
    cout <= (i1 AND i2) OR ((i1 XOR i2)
            AND cin);
END dataflow;
```

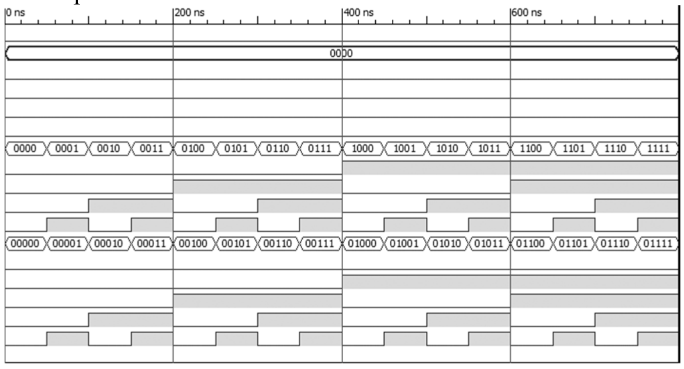
[add_sub_structural.vhd]
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY add_sub IS PORT (
    X3, X2, X1, X0: IN STD_LOGIC;
    Y3, Y2, Y1, Y0, A_S: IN STD_LOGIC;
    S4, S3, S2, S1, S0: OUT STD_LOGIC
);
END add_sub;

ARCHITECTURE structural OF add_sub IS
    SIGNAL F0, F1, F2, F3: STD_LOGIC;
    SIGNAL C1, C2, C3 : STD_LOGIC;

COMPONENT xor1 IS PORT (
    i1, i2: IN STD_LOGIC;
    o1: OUT STD_LOGIC
);
END COMPONENT;

COMPONENT full_adder IS PORT (
    i1, i2, cin: IN STD_LOGIC;
    sum, cout: OUT STD_LOGIC
);
END COMPONENT;

BEGIN
    A0: xor1 PORT MAP (i1 => A_S, i2 =>
         Y0, o1 => F0);
    A1: full_adder PORT MAP (i1 => X0, i2
=> F0, cin => A_S, sum => S0, cout => C1);

    A2: xor1 PORT MAP (i1 => A_S, i2 =>
         Y1, o1 => F1);
    A3: full_adder PORT MAP (i1 => X1, i2
=> F1, cin => C1, sum => S1, cout => C2);

    A4: xor1 PORT MAP (i1 => A_S, i2 =>
         Y2, o1 => F2);
    A5: full_adder PORT MAP (i1 => X2, i2
=> F2, cin => C2, sum => S2, cout => C3);

    A6: xor1 PORT MAP (i1 => A_S, i2 =>
         Y3, o1 => F3);
    A7: full_adder PORT MAP (i1 => X3, i2
=> F3, cin => C3, sum => S3, cout => S4);

END structural;
```

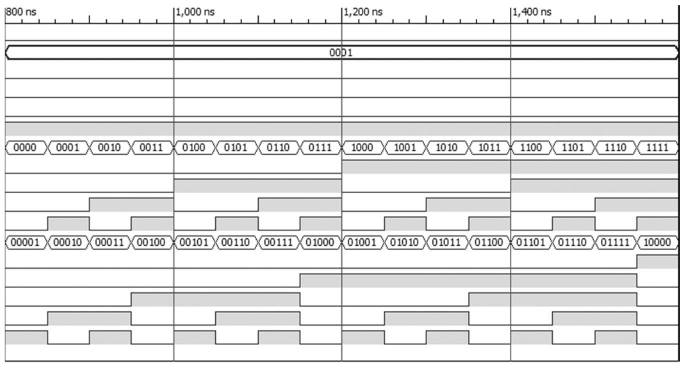
Test Bench
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY add_sub_tb IS
END add_sub_tb;

ARCHITECTURE behavioral OF add_sub_tb IS
    COMPONENT add_sub
    PORT(
          X3, X2, X1, X0: IN STD_LOGIC;
          Y3, Y2, Y1, Y0: IN STD_LOGIC;
          A_S: IN STD_LOGIC;
          S4,S3,S2,S1,S0: OUT STD_LOGIC
    );
    END COMPONENT;

    SIGNAL addsub: STD_LOGIC := '0';
    SIGNAL                input_vector1:
STD_LOGIC_VECTOR( 3 DOWNTO 0) := "0000";
    SIGNAL                input_vector2:
STD_LOGIC_VECTOR( 3 DOWNTO 0) := "0000";
    SIGNAL                output_vector:
STD_LOGIC_VECTOR( 4 DOWNTO 0) := "00000";

BEGIN
    uut: add_sub PORT MAP(
          A_S => addsub,
          X3 => input_vector1(3),
          X2 => input_vector1(2),
            X1 => input_vector1(1),
            X0 => input_vector1(0),

            Y3 => input_vector2(3),
            Y2 => input_vector2(2),
            Y1 => input_vector2(1),
            Y0 => input_vector2(0),

            S4 => output_vector(4),
            S3 => output_vector(3),
            S2 => output_vector(2),
            S1 => output_vector(1),
            S0 => output_vector(0)
        );

        stim_proc: PROCESS

    BEGIN
        FOR index1 IN 0 TO 15 LOOP
            input_vector1                <=
std_logic_vector(to_unsigned(index1,4));
            FOR index2 IN 0 TO 15 LOOP
                input_vector2            <=
std_logic_vector(to_unsigned(index2,4));
                WAIT FOR 50 ns;
            END LOOP;
        END LOOP;
    END PROCESS;

    END behavioral;
```

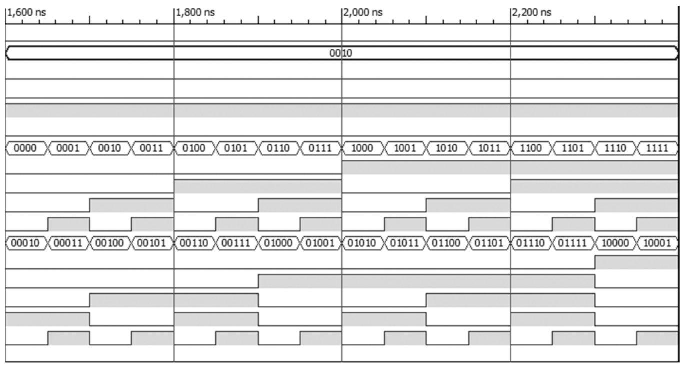
Discussion:

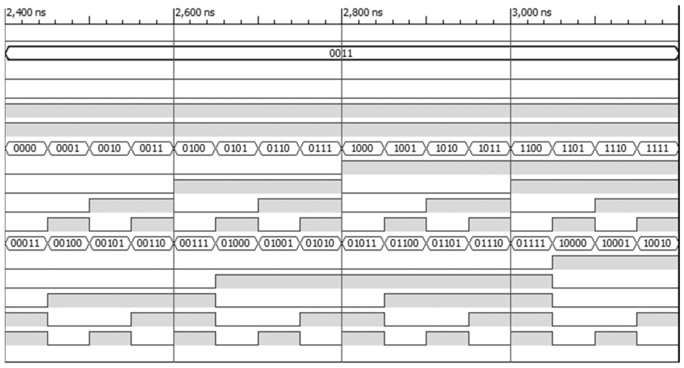
To make a 4-bit adder/subtracter, we need extra input, namely add/sub as given in block diagram above. In block diagram, there are two component, XOR gate and 1-bit full adder. So, while implementing using structural style, two components are first defined. 1-bit full adder is designed using AND, OR and XOR gate as per the logic diagram of full adder. These components are then used in structural description of 4-bit full adder. The test bench code that produces output for all possible cases is made to make sure that our design is complete and accurate. Waveforms depicting all possible input cases are included as output for this activity.

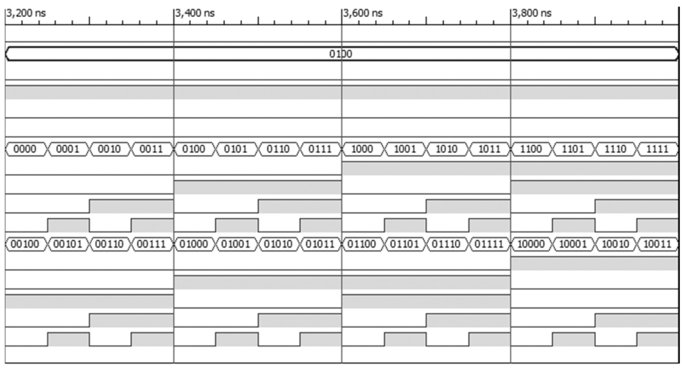
Output:

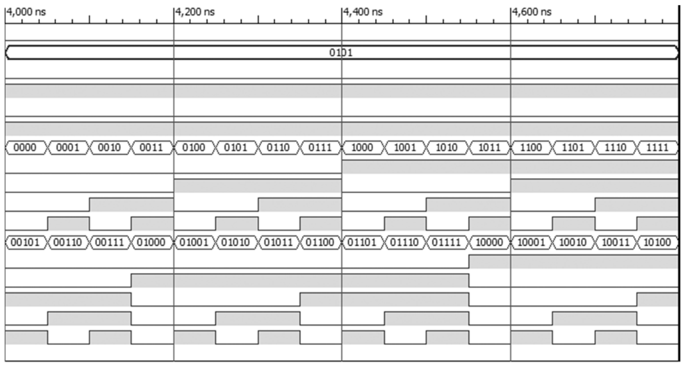


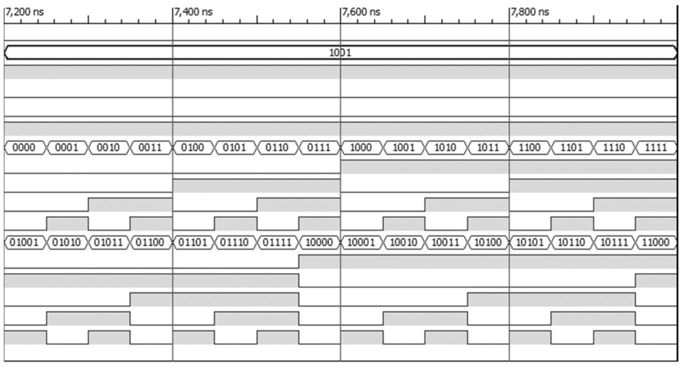


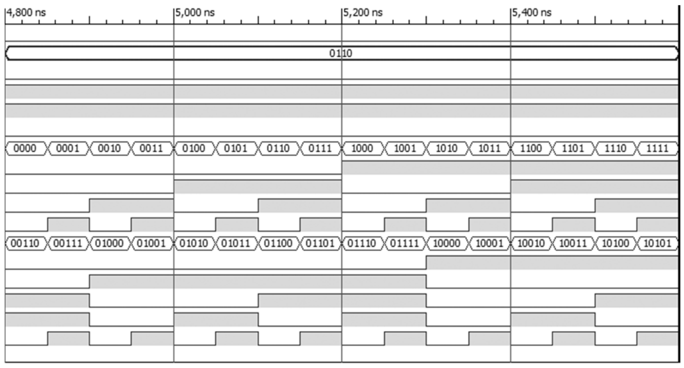


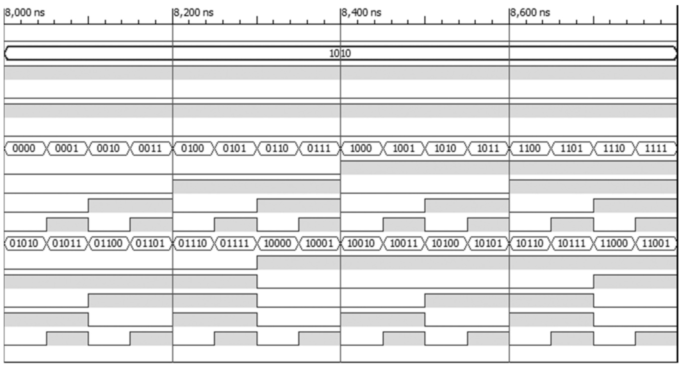


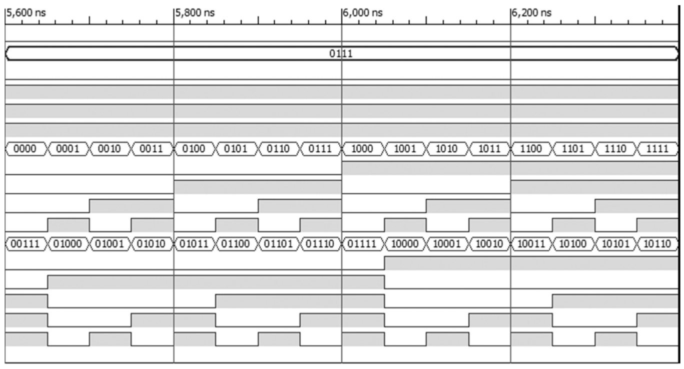


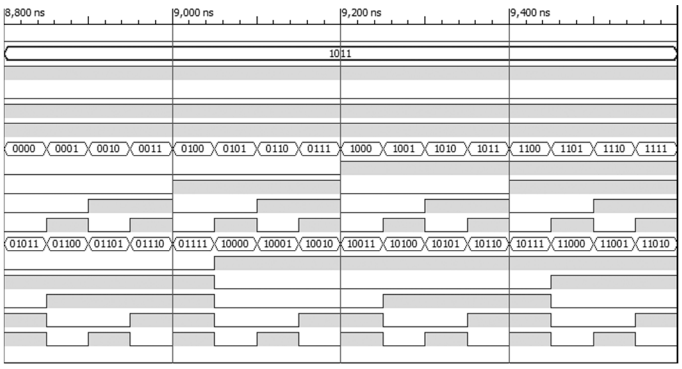

4,000 ns
4,200 ns
4,400 ns
4,600 ns
0101
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
00101 00110 00111 01000 01001 01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100

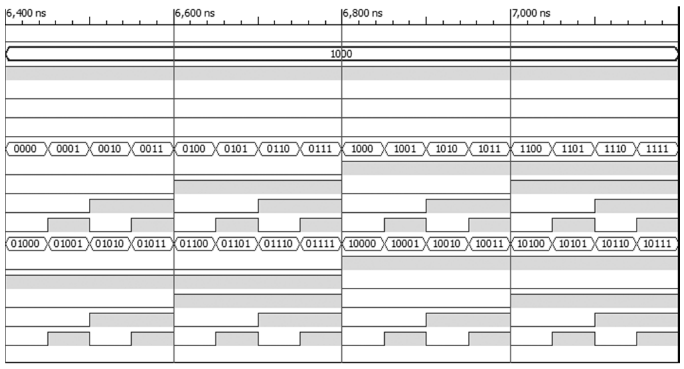

7,200 ns
7,400 ns
7,600 ns
7,800 ns
1001
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
01001 01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111 11000

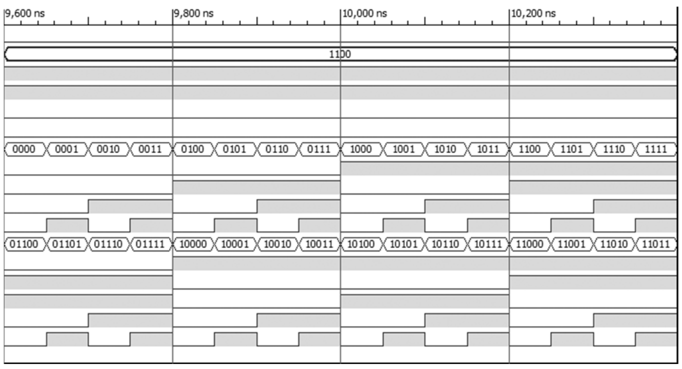

4,800 ns
5,000 ns
5,200 ns
5,400 ns
0110
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
00110 00111 01000 01001 01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101

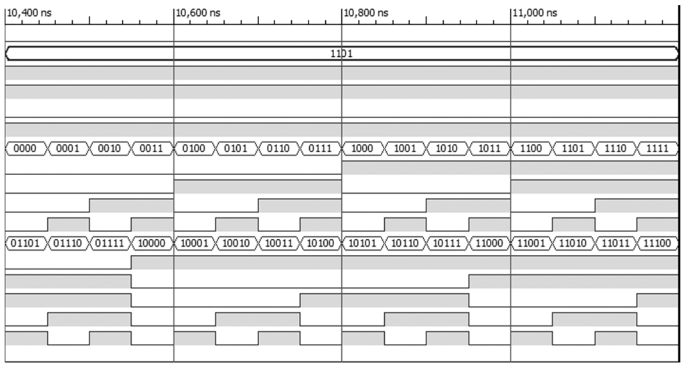

8,000 ns
8,200 ns
8,400 ns
8,600 ns
1010
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111 11000 11001

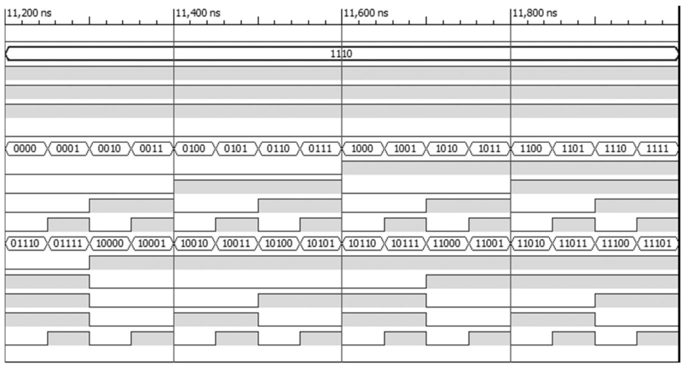

5,600 ns
5,800 ns
6,000 ns
6,200 ns
0111
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
00111 01000 01001 01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110

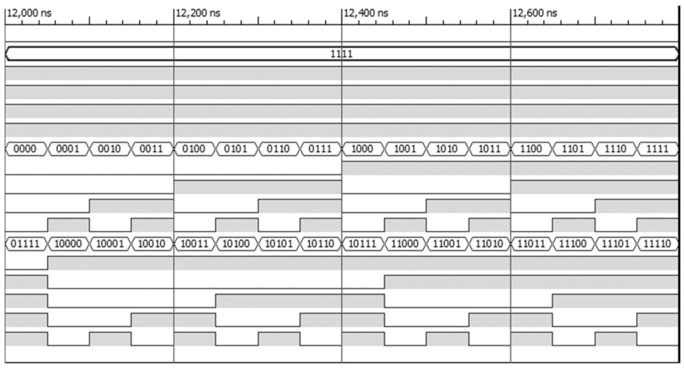

8,800 ns
9,000 ns
9,200 ns
9,400 ns
1011
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111 11000 11001 11010


6,400 ns
6,600 ns
6,800 ns
7,000 ns
1000
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
01000 01001 01010 01011 01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111

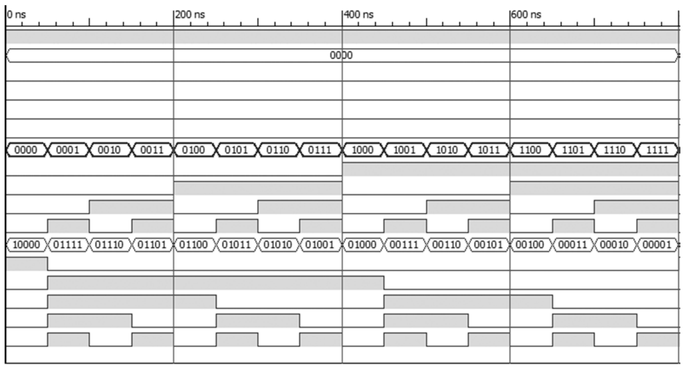

9,600 ns
9,800 ns
10,000 ns
10,200 ns
1100
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111 11000 11001 11010 11011

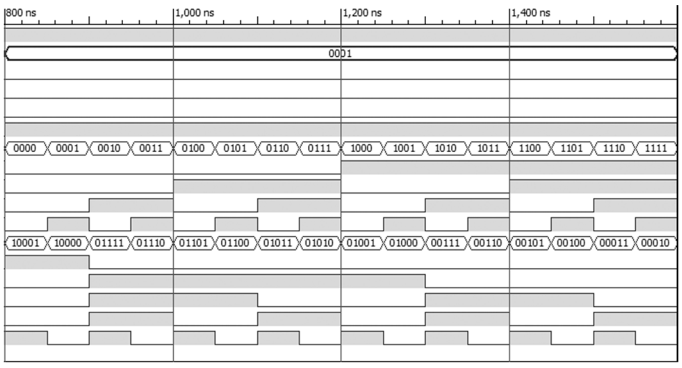


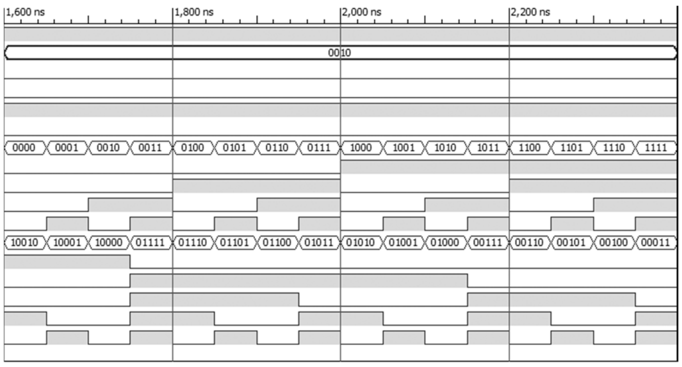


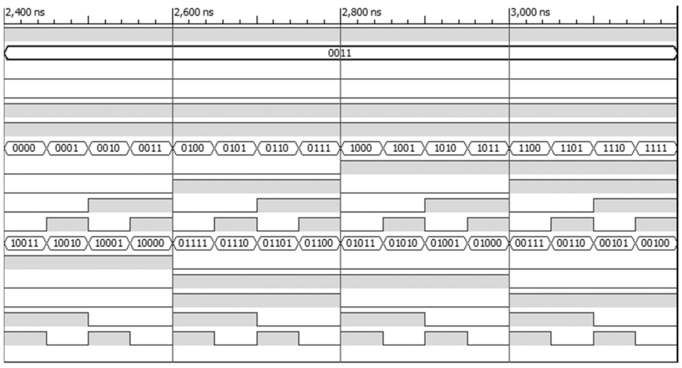


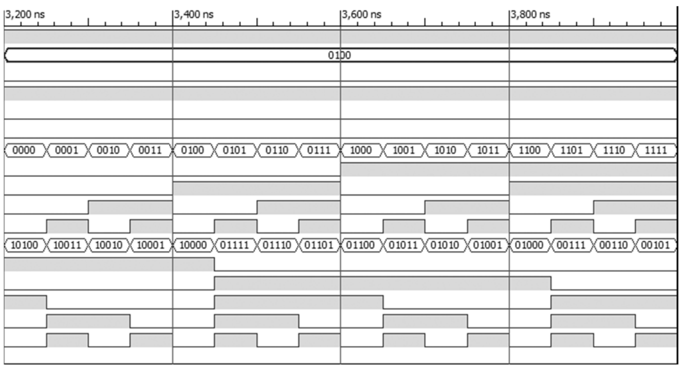
For subtraction:

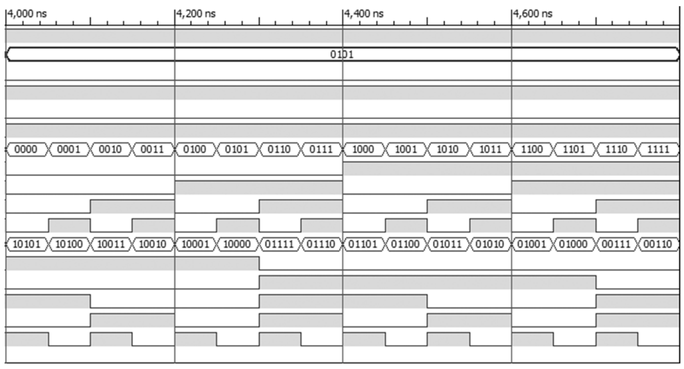


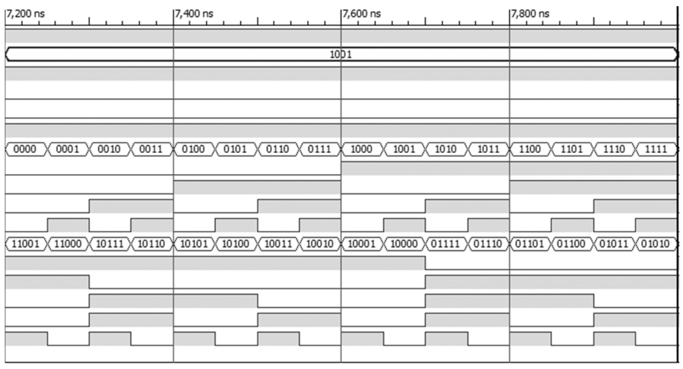


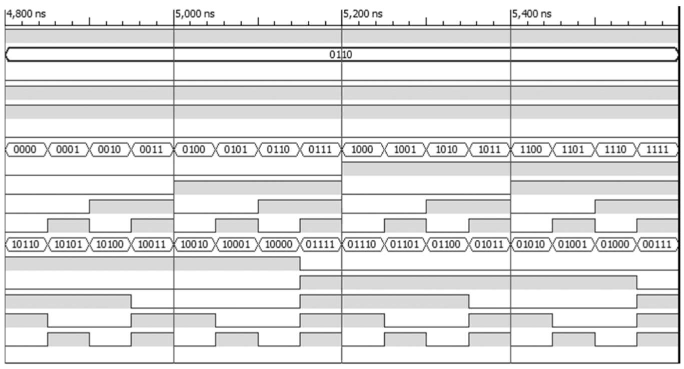


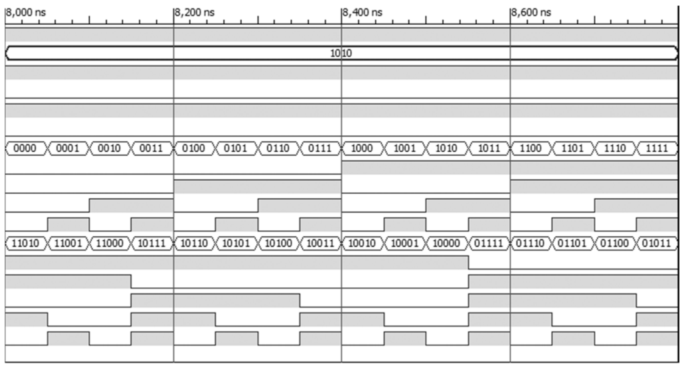


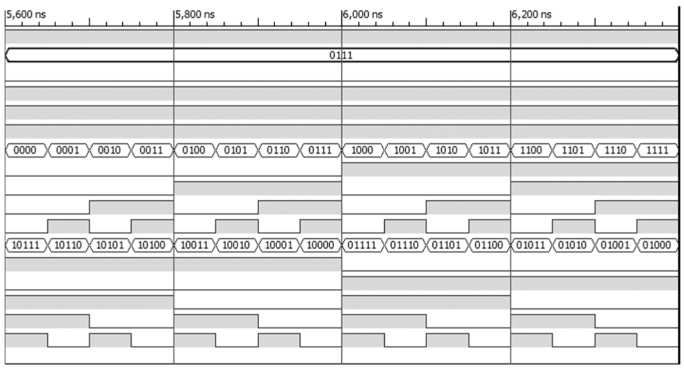


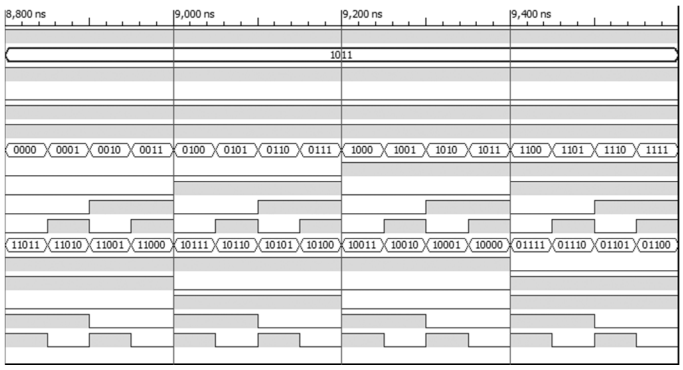

4,000 ns
4,200 ns
4,400 ns
4,600 ns
0101
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
10101 10100 10011 10010 10001 10000 01111 01110 01101 01100 01011 01010 01001 01000 00111 00110

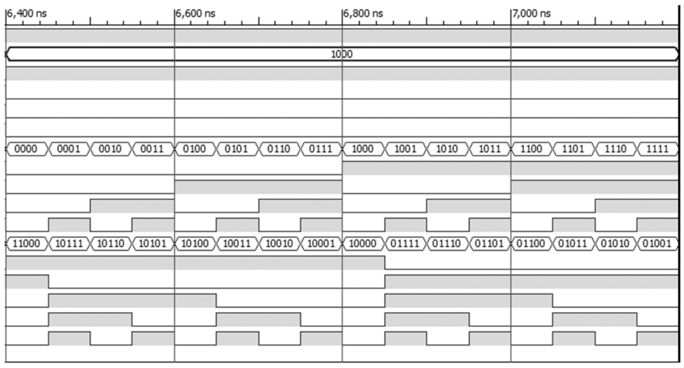

7,200 ns
7,400 ns
7,600 ns
7,800 ns
1001
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
11001 11000 10111 10110 10101 10100 10011 10010 10001 10000 01111 01110 01101 01100 01011 01010

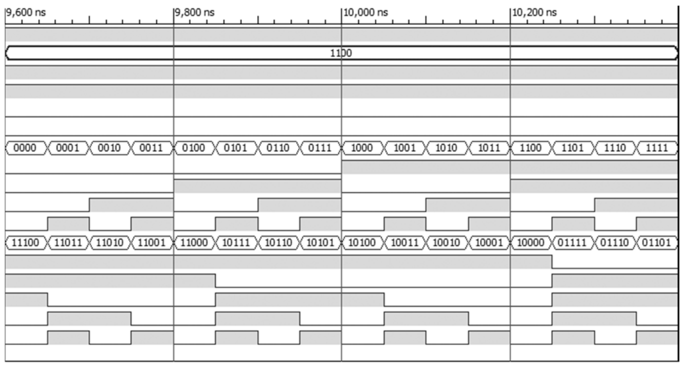

4,800 ns
5,000 ns
5,200 ns
5,400 ns
0110
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
10110 10101 10100 10011 10010 10001 10000 01111 01110 01101 01100 01011 01010 01001 01000 00111

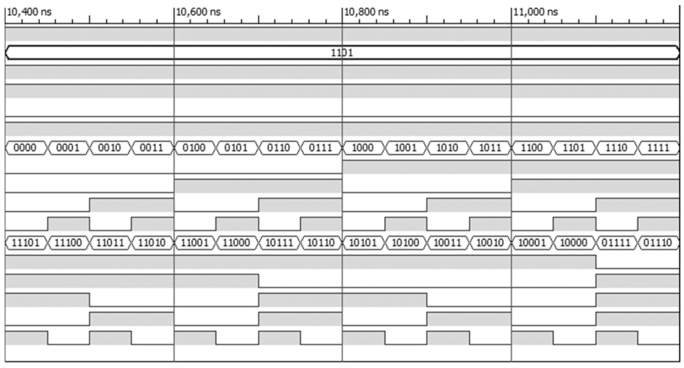

8,000 ns
8,200 ns
8,400 ns
8,600 ns
1010
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
11010 11001 11000 10111 10110 10101 10100 10011 10010 10001 10000 01111 01110 01101 01100 01011

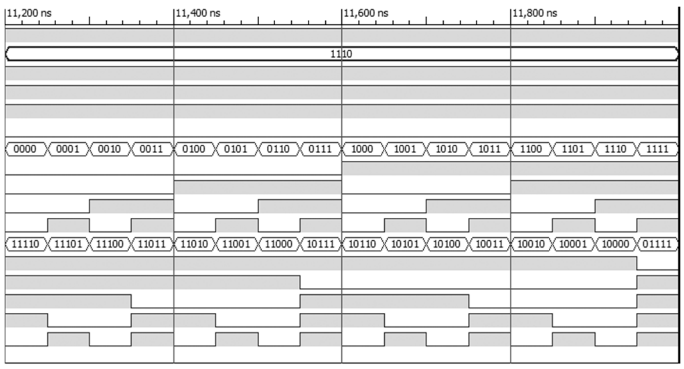

5,600 ns
5,800 ns
6,000 ns
6,200 ns
0111
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
10111 10110 10101 10100 10011 10010 10001 10000 01111 01110 01101 01100 01011 01010 01001 01000

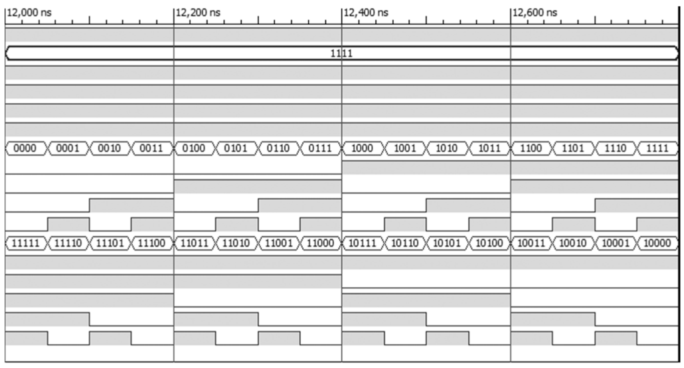

8,800 ns
9,000 ns
9,200 ns
9,400 ns
1011
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
11011 11010 11001 11000 10111 10110 10101 10100 10011 10010 10001 10000 01111 01110 01101 01100


6,400 ns
6,600 ns
6,800 ns
7,000 ns
1000
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
11000 10111 10110 10101 10100 10011 10010 10001 10000 01111 01110 01101 01100 01011 01010 01001


9,600 ns
9,800 ns
10,000 ns
10,200 ns
1100
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
11100 11011 11010 11001 11000 10111 10110 10101 10100 10011 10010 10001 10000 01111 01110 01101

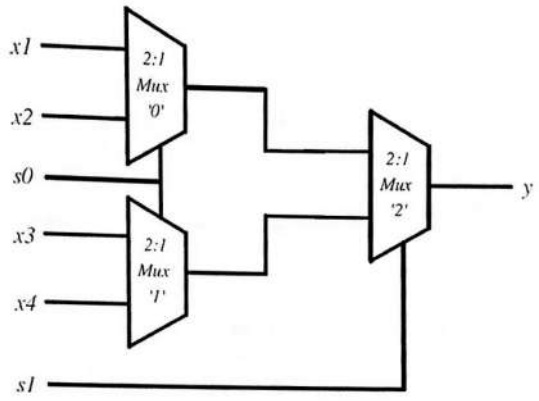






VIII. ACTIVITY VII

Write VHDL code to implement a 4:1 MUX having inputs (x1, x2, x3, and x4), select lines (s1, s0) and output (y) using three 2:1 multiplexers as the basic building blocks.



Use a hierarchical design approach. Create component definitions in separate (.vhd) files. Use either Dataflow or Behavioral or Structural design styles. Use structural design style for the 4:1 MUX architecture. Make use of 2:1 MUX component declaration. Make use of 2:1 MUX component instantiation. Write a VHDL test bench to verify the operation of the 4:1 MUX. Provide a simulation waveform depicting all possible input cases.

VHDL Code:

[mux2to1_dataflow.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux2to1 IS PORT (
    SEL, I1, I2: IN STD_LOGIC;
O: OUT STD_LOGIC
);
END mux2to1;

ARCHITECTURE dataflow OF mux2to1 IS
BEGIN
    WITH (NOT SEL AND I1) OR (SEL AND I2)
SELECT
          O <=  '1' WHEN '1',
                '0' WHEN '0',
                '0' WHEN OTHERS;
END dataflow;
```

[mux4to1_structural.vhd]

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY mux4to1 IS PORT (
    X1, X2, X3, X4, S0, S1: IN STD_LOGIC;
    Y: OUT STD_LOGIC
);
END mux4to1;

ARCHITECTURE structural OF mux4to1 IS
    SIGNAL F1, F2: STD_LOGIC;

COMPONENT mux2to1 IS PORT (
    I1, I2, SEL: IN STD_LOGIC;
    O: OUT STD_LOGIC
);
END COMPONENT;

BEGIN
    M0: mux2to1 PORT MAP (I1 => X1, I2 =>
         X2, SEL => S0, O => F1);
    M1: mux2to1 PORT MAP (I1 => X3, I2 =>
         X4, SEL => S0, O => F2);
    M2: mux2to1 PORT MAP (I1 => F1, I2 =>
         F2, SEL => S1, O => Y);
END structural;
```

Test bench

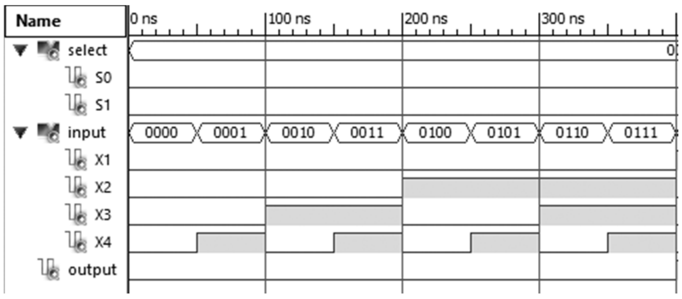
```
LIBRARY IEEE;
```

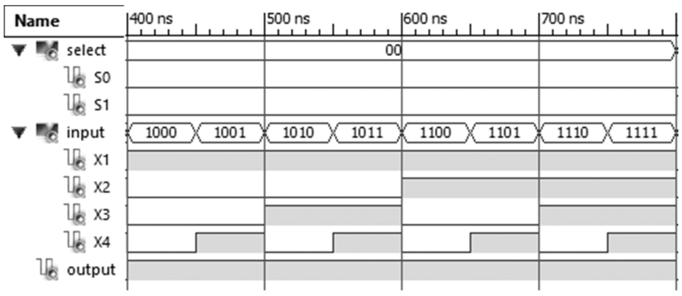
```
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY mux4to1_tb IS
END mux4to1_tb;

ARCHITECTURE behavioral OF mux4to1_tb IS
    COMPONENT mux4to1
    PORT(
          X1, X2, X3, X4: IN STD_LOGIC;
          S0, S1: IN STD_LOGIC;
          Y: OUT STD_LOGIC
    );
    END COMPONENT;

    SIGNAL                   select_vec:
STD_LOGIC_VECTOR( 1 DOWNTO 0) := "00";
    SIGNAL                    input_vec:
STD_LOGIC_VECTOR( 3 DOWNTO 0) := "0000";
    SIGNAL output: STD_LOGIC := '0';

BEGIN
    uut: mux4to1 PORT MAP(
          S0 => select_vec(0),
          S1 => select_vec(1),
          X1 => input_vec(3),
          X2 => input_vec(2),
          X3 => input_vec(1),
          X4 => input_vec(0),
          Y => output
    );

    stim_proc: PROCESS

BEGIN
    FOR selector IN 0 TO 3 LOOP
          select_vec <= std_logic_vector
(to_unsigned(selector,2));
          FOR index IN 0 TO 15 LOOP
                input_vec             <=
std_logic_vector(to_unsigned(index,4));
                WAIT FOR 50 ns;
          END LOOP;
    END LOOP;
END PROCESS;
END behavioral;
```

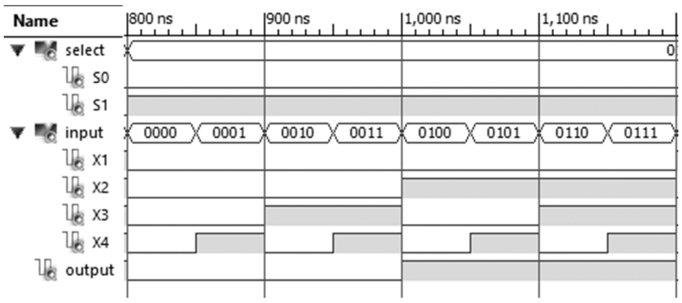
Discussion:

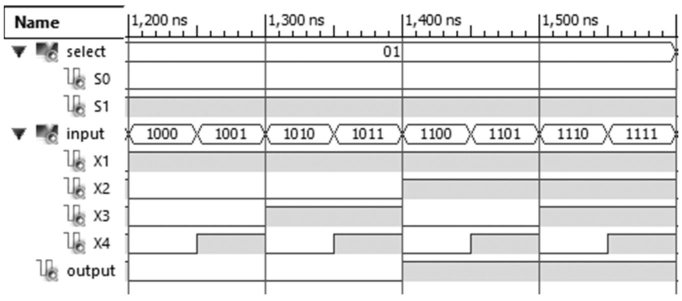
In this activity, we need to make 4:1 multiplexer using three 2:1 multiplexers as shown in the logic diagram. So, for the component declaration, simple dataflow style is used and for 4:1 multiplexer description, hierarchical structural design style is used as instructed. In 4:1 multiplexer, there are six inputs, four input lines and two select lines required to choose one of four input lines. Single output of 4:1 multiplexer is used to provide any one of four input specified by bit pattern of select lines. A VHDL test bench is made to simulate the operation of 4:1 multiplexers for all possible input cases. Waveform for all possible case is given below as output.

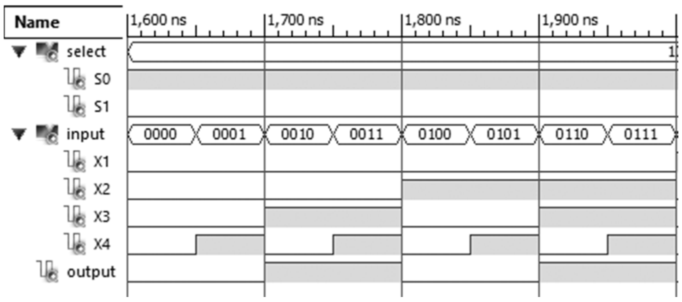
Output:

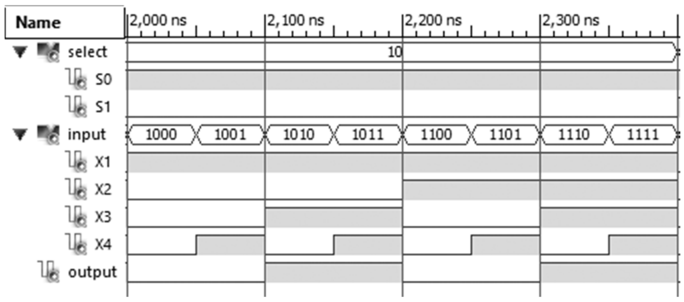


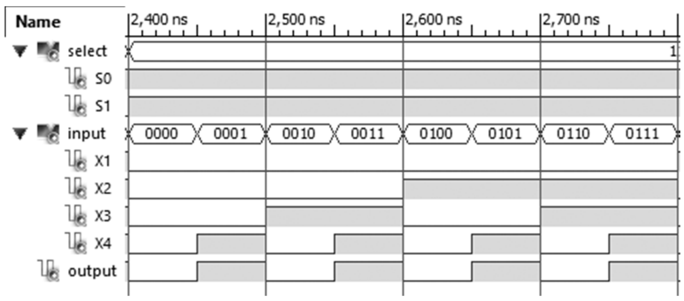


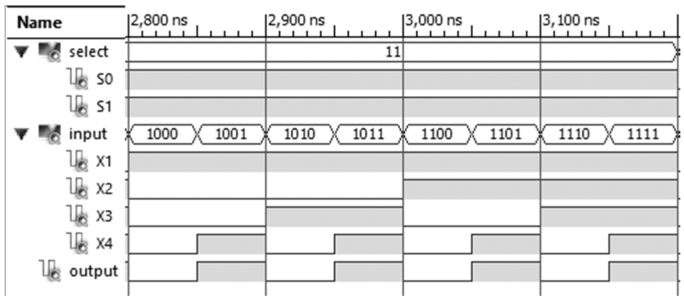












Conclusion

Various activities concerned VHDL programming were done in this lab. Different logic circuits were designed in different styles as instructed using Xilinx ISE Design Suite. Required truth table along with Karnaugh map simplification is shown in this report. VHDL codes to all activities are included in this report. Waveform depicting all cases for all activities are included as output.

Acknowledgment

This lab report is prepared as a document for activities done in lab concerned with combinational logic design using VHDL programming. This report is made accurate and professional as far as possible. I would like to express our deepest gratitude to our teacher, Mr. Dinesh Baniya Kshatri, for guiding us in the practical. I am very grateful to the Department of Electronics and Computer Engineering (DoECE) of IOE Central Campus, Pulchowk for arranging such a schedule on our academic side.

References

[1] (2016) The Wikipedia website. [Online]. Available: https://en.m.wikipedia.org/wiki/VHDL/

[2] P. J. Ashenden, The VHDL Cookbook [portable document]. Available: http://www.cs.adelaide.edu.au/

[3] D. Kshatri, Combinational Logic Design using VHDL [scanned document]. Available: Offline.

[4] S. Shrestha, VHDL [presentation]. Available: Offline