Institute of Engineering & Management

<u>Department of Computer Science & Engineering</u>

<u>Lab Mnaual</u>

Paper name: Computer Organization &Architecture Lab

Paper Code: PCCCS492

B. Tech

2nd year 4th Semester

Faculty Coordinators: Prof. Dr. Sukanya Mukherjee, Prof. Shreejita Mukherjee, Prof. Shubhasri Roy

INSTITUTE OF ENGINEERING & MANAGEMENT

COMPUTER ORGANIZATION & ARCHITECTURE LAB

PAPER CODE: PCCCS 492

LAB ASSIGNMENTS

1.Implementation of AND, OR, NOT, XOR, NANAD, NOR gates using Xilinx ISE.

2. Implementation of Half Adder, Half Subtractor, Full Adder, Full Subtractor using Xilinx ISE.

3. Implementation of 8 bit Adder, 8bit Subtractor, 8 bit Multiplier and 4 bit parallel Adder and 4 bit parallel subtractor using Xilinx ISE.

4. Implementation of Binary to Gray and Gray to binary using Xilinx ISE.

5. Implementation of 4 bit comparator, 2:4 Decoder, 3:8 Decoder using Xilinx ISE.

6. Implementation of 2:4 Decoder (using case), 4:2 Encoder (Data flow and using case), 8:3 Encoder (using Loop).

7. Implementation of 2:1 MUX (Data flow), 4:1 MUX (Dataflow, if-else, using case), 1:4 DEMUX (using case and data flow) using Xilinx ISE.

8. Implementation of Full Adder using Half Adders (Structural Method), 4 bit Parallel Adder (using Structural Method).

9. Implementation of 2:1 MUX using basic gates(Structural Method), 4:1 MUX using 2:1 MUX (using Structural Method).

10. Implementation of SR Flip-Flop, JK Flip Flop, D Flip Flop and T (Toggle) Flip Flop using Xilinx ISE.

Assignment 3: Implementation of 8-bit Adder, 8-bit Subtractor, 8-bit Multiplier and 4 bit parallel

Objective: To implement 8-bit Adder.

Software used:

Property Name	Value
Device family	Spartan3
Device	XC3550
Package	PQ208
Speed	-5
Top-level source type	HDL
Synthesis Tool	XST(VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL

Theory:

2. 8-bit Adder

An **8-bit adder** is a combinational circuit that performs binary addition on two 8-bit inputs. It consists of **full adders** connected in cascade to propagate the carry. The sum and carry outputs are generated based on binary arithmetic rules.

Working Principle:

- A **full adder** adds three binary inputs: two operand bits and a carry-in bit.
- The carry output from one full adder is forwarded to the next higher-order bit.
- For an 8-bit adder, **eight full adders** are cascaded to compute the final sum and carry-out.

Boolean Expressions for a Full Adder:

Sum = $A \oplus B \oplus Cin$

Carry = $(A \cdot B) + (B \cdot Cin) + (A \cdot Cin)$

Design Methodology:

- The full adder logic is implemented using VHDL.
- Eight full adders are connected to create an 8-bit ripple-carry adder.
- The design is synthesized using XST and simulated in ISim.

Truth Table:

A (8-bit)	B (8-bit)	Cin	Sum (8-bit)	Cout
00000000	00000000	0	00000000	0
00000001	00000001	0	00000010	0
00001111	00000001	0	00010000	0
11111111	00000001	0	00000000	1
10101010	01010101	0	11111111	0
11001100	00110011	1	10000000	1
11111111	11111111	0	11111110	1
11111111	11111111	1	11111111	1

Code:

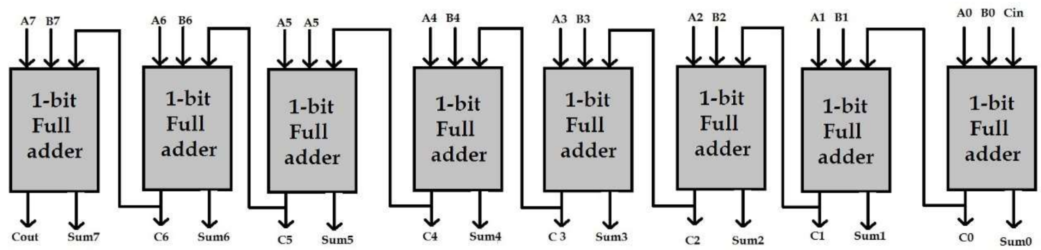
Behavioral Model Code:

```
begin
    process(A, B)
        variable temp : STD_LOGIC_VECTOR(8 downto 0);
    begin
        temp := ('0' & A) + ('0' & B);
        Sum <= temp(7 downto 0);
        Carry_Out <= temp(8);
    end process;
end Behavioral;
```

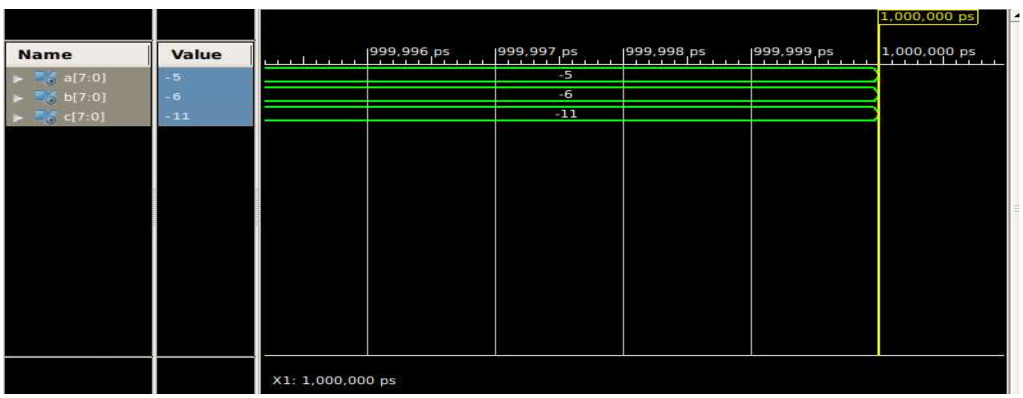
Data flow Model Code:

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity eight_bit_adder is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           C : out  STD_LOGIC_VECTOR (7 downto 0));
end eight_bit_adder;

architecture Behavioral of eight_bit_adder is

begin
C(7 downto 0) <= A(7 downto 0) + B(7 downto 0);

end Behavioral;
```

2. Data flow Model:



Output:



Objective: 8-bit Subtractor.

Theory:

An **8-bit subtractor** is a digital circuit used to perform subtraction of two 8-bit binary numbers. It takes two 8-bit inputs: **minuend (A)** and **subtrahend (B)**, and produces an **8-bit difference (D)** along with a **borrow-out (Bout)** if needed.

Types of 8-bit Subtractors

1. **Direct Subtractor**:
   - Uses **binary subtraction** (A - B) directly.
   - If `A < B`, a **borrow** is generated.
2. **Subtractor Using 2's Complement**:
   - Instead of performing direct subtraction, it **adds** the **2's complement** of `B` to `A`.
   - The 2's complement of `B` is calculated by **inverting all bits of B and adding 1**.
   - This converts subtraction into an addition operation.

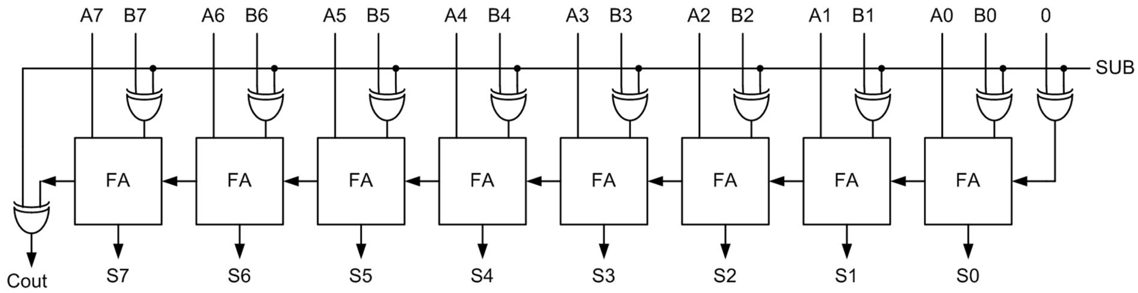
Working of 8-bit Subtractor

- If using direct subtraction: $D = A - B$
- If using 2's complement: $D = A + (\text{NOT } B + 1)$
- The **borrow-out (`Bout`)** indicates whether `A < B` (i.e., subtraction resulted in a negative value).

Truth Table:

A (8-bit)	B (8-bit)	Difference (D)	Borrow (Bout)
00000000	00000000	00000000	0
00000010	00000001	00000001	0
10000000	00000001	01111111	0
00000001	00000010	11111111	1
11111111	00000001	11111110	0
01010101	10101010	10101011	1
11111111	11111111	00000000	0
00000000	00000001	11111111	1

Data flow Model:



Code:

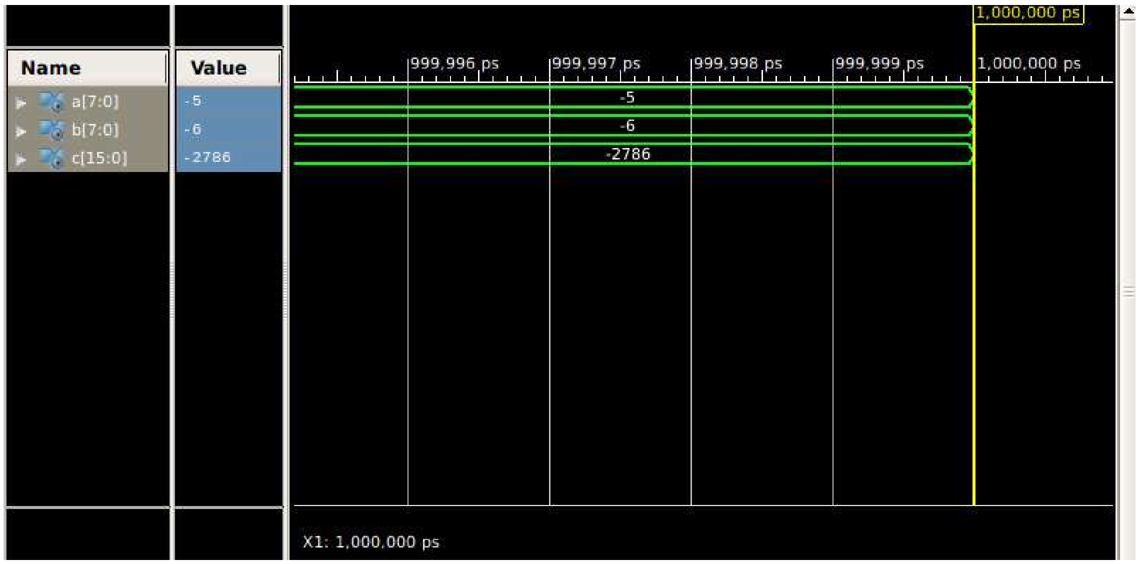
1. Behavioral Model Code:

```
        signal temp : STD_LOGIC_VECTOR(8 downto 0);
    begin
        process(A, B)
        begin
            temp <= ('0' & A) - ('0' & B);
            D    <= temp(7 downto 0);
            Bout <= temp(8);
        end process;
    end Behavioral;

```

2. Data flow Model Code:

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity eight_bit_adder is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           C : out  STD_LOGIC_VECTOR (7 downto 0));
end eight_bit_adder;

architecture Behavioral of eight_bit_adder is

begin
C(7 downto 0) <= A(7 downto 0) - B(7 downto 0);

end Behavioral;


```

Output:


1,000,000 ps
Name
Value
a[7:0]
-5
b[7:0]
-6
c[15:0]
-2786
999,996 ps
999,997 ps
999,998 ps
999,999 ps
1,000,000 ps
-5
-6
-2786
X1: 1,000,000 ps

Objective: The objective of this experiment is to design and implement an **8-bit binary multiplier** using **VHDL** on a **Spartan-3 (XC3550) FPGA**. The multiplication operation follows **binary arithmetic** principles and produces a **16-bit product** from two **8-bit inputs**.

Theory:

An **8-bit multiplier** is a combinational digital circuit that multiplies two **8-bit binary numbers**, producing a **16-bit product**. Multiplication in binary follows the same principles as decimal multiplication but uses only **0s and 1s**.

Types of 8-bit Multipliers

1. **Combinational Multiplier**
   - Uses an array of AND gates and adders to perform direct binary multiplication.
   - Fast but requires more hardware.
2. **Sequential Multiplier**
   - Uses shift-and-add method to perform multiplication over multiple clock cycles.
   - Requires fewer hardware resources but is slower.

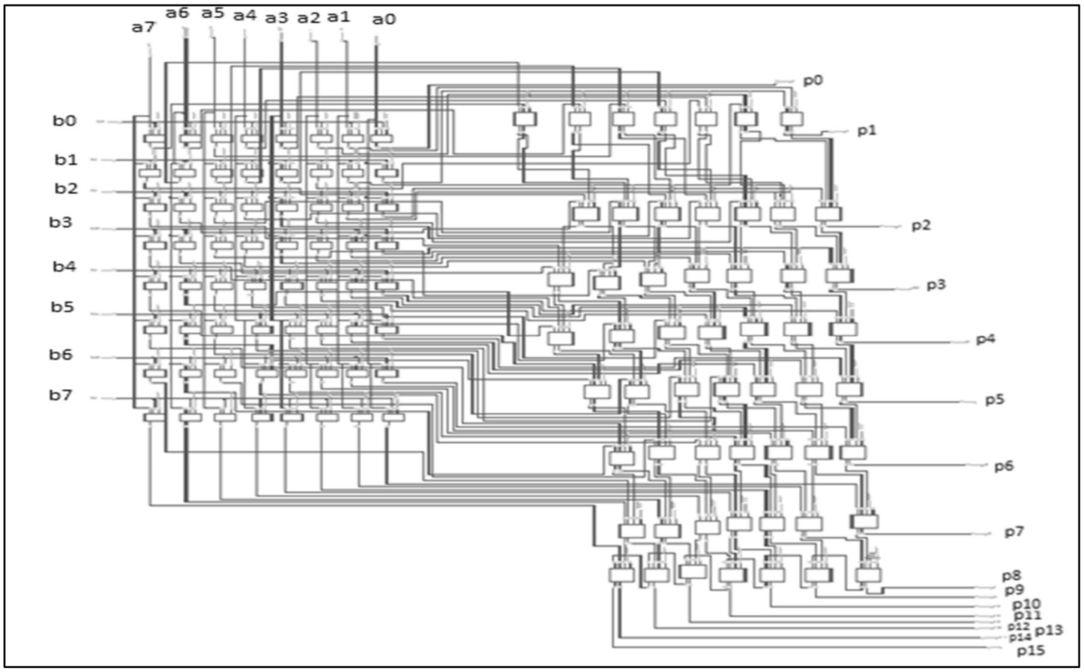
Binary Multiplication Process

- Each bit of the **multiplicand** is multiplied with each bit of the **multiplier** using AND gates.
- Partial products are generated, shifted left accordingly, and summed to get the final product.
- The result is **16 bits**, as multiplying two n-bit numbers results in a 2n-bit product.

Truth Table:

A (8-bit)	B (8-bit)	Product (16-bit)
00000000	00000000	0000000000000000
00000001	00000001	0000000000000001
00000010	00000010	0000000000000100
00000011	00000010	0000000000000110
00000101	00000011	0000000000001111
11111111	00000001	0000000011111111
11111111	11111111	1111111000000001
10000000	00000010	0000000010000000

Data flow Model:



Code:

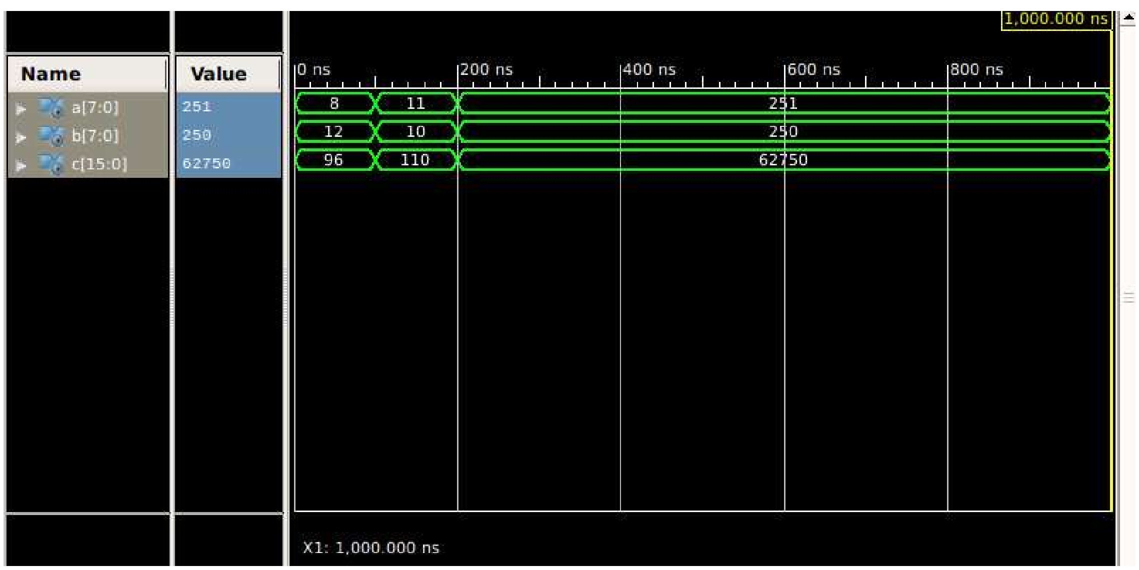
1. Behavioral Model Code:

```
begin
    process(A, B)
    begin
        C <= conv_std_logic_vector(conv_integer(A) * conv_integer(B), 16);
    end process;
end Behavioral;

```

2. Data flow Model Code:

```vhdl
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity eight_bit_adder is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           C : out  STD_LOGIC_VECTOR (15 downto 0));
end eight_bit_adder;

architecture Behavioral of eight_bit_adder is

begin
C(15 downto 0) <= A(7 downto 0) * B(7 downto 0);

end Behavioral;
```

Output:



<u>Discussion :</u>

The implementation of arithmetic circuits using **VHDL in Xilinx ISE** is a fundamental step in understanding digital design, arithmetic operations, and FPGA-based computation. This lab covers the design and implementation of essential arithmetic components, including an **8-bit adder, 8-bit subtractor, 8-bit multiplier, 4-bit parallel adder, and 4-bit parallel subtractor**. Each of these circuits plays a critical role in digital systems, particularly in arithmetic logic units (ALUs), digital processors, and embedded applications. The **8-bit adder** is designed using a **ripple carry adder (RCA) structure**, where each bit addition is performed using a **full adder** (FA), and the carry is propagated sequentially from the least significant bit (LSB) to the most significant bit (MSB). This design ensures correct binary addition while handling carry propagation efficiently. The **8-bit subtractor**, instead of performing direct subtraction, follows the **two's complement method**, where subtraction (`A - B`) is implemented as **`A + (2's complement of B)`**. The two's complement of `B` is obtained by **inverting all bits of B and adding 1**, allowing the same adder circuit to be reused for subtraction, optimizing resource usage on an FPGA.

The **8-bit multiplier** is implemented using the **shift-and-add method**, a sequential approach to binary multiplication. This method involves generating **partial products** by selectively shifting and adding based on the multiplier's bit values. Each bit in the multiplier determines whether the multiplicand is added to the partial sum, and the intermediate results are shifted accordingly. This approach effectively simulates manual multiplication in binary and is optimized for FPGA-based implementations. The **4-bit parallel adder** functions similarly to the **8-bit adder** but operates on **smaller 4-bit values**, making it useful for low-bit arithmetic computations in resource-constrained environments. The **4-bit parallel subtractor** follows the same principle as the 8-bit subtractor, using **two's complement addition** to perform subtraction while maintaining design efficiency. These smaller arithmetic circuits are essential in applications where speed, power consumption, and area optimization are crucial.

The **implementation and verification** of these arithmetic circuits are carried out using **Xilinx ISE**, which enables simulation and synthesis of the VHDL designs. Simulation ensures the correctness of the logic by allowing verification of the sum, difference, product, and carry outputs under different input conditions. Synthesis translates the VHDL description into a hardware realization that can be mapped onto an **FPGA**, ensuring efficient utilization of logic resources. Understanding and implementing these arithmetic operations provide foundational knowledge for designing **larger arithmetic units, digital processors, and signal processing units**, which are critical components in modern computing and embedded systems. The ability to implement such designs in VHDL demonstrates the power of hardware description languages (HDLs) in digital system development, reinforcing the importance of programmable logic design and FPGA-based architectures in real-world applications.

Assignment 4 : Implementation of Binary to Gray and Gray to binary using Xilinx ISE.

Software used:

Property Name	Value
Device family	Spartan3
Device	XC3550
Package	PQ208
Speed	-5
Top-level source type	HDL
Synthesis Tool	XST(VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL

Objective: To design and implement a **Binary to Gray Code Converter** using **VHDL** on a **Spartan-3 (XC3550) FPGA**. This conversion is essential in digital systems to minimize **bit transition errors**

Theory:

Binary Code

Binary numbers represent data using **base-2** (0s and 1s). However, a drawback of binary representation is that **multiple bits may change simultaneously** when transitioning between consecutive values, causing errors in **high-speed circuits**.

Gray Code

Gray code is a **reflected binary code** where **only one-bit changes** between consecutive values. This property helps in reducing errors in digital systems such as **rotary encoders, Karnaugh maps, and communication systems**.

Binary to Gray Code Conversion Rule

The **n-bit Gray code** is derived from the **n-bit binary number** using the formula:

$Gn = Bn$

$G_{n-1} = Bn \oplus B_{n-1}$

$G_{n-2} = B_{n-1} \oplus B_{n-2}$

$G0 = B1 \oplus B0$

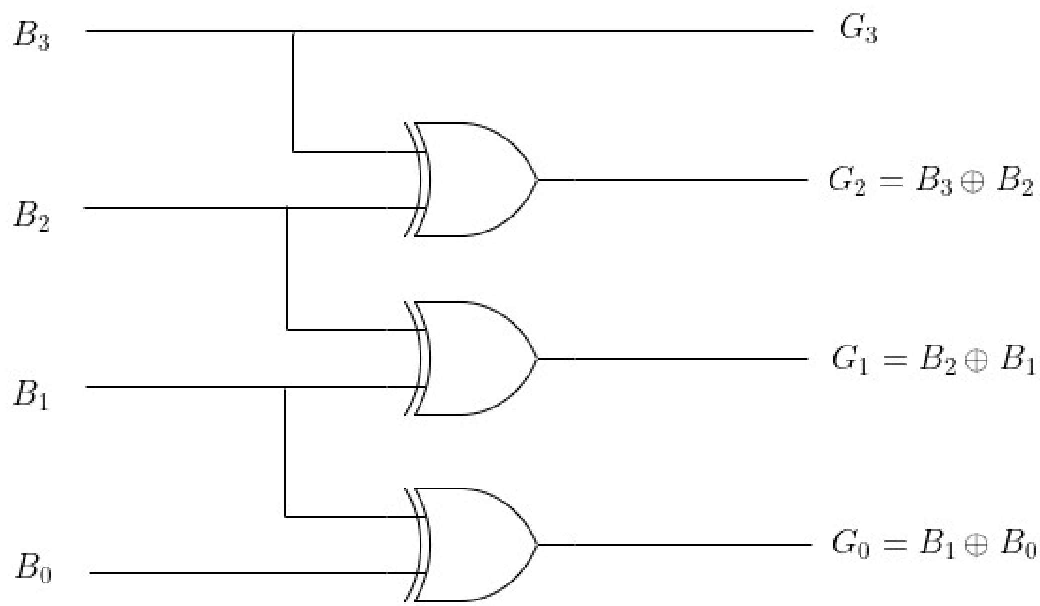
Where:

- Bn = Most Significant Bit (MSB) of the binary number.
- $\oplus$ = XOR (Exclusive OR) operation.
- Gn = Most Significant Bit (MSB) of the gray code.

Truth Table:

Binary (B)	Gray Code (G)
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Data flow Model:



Code:

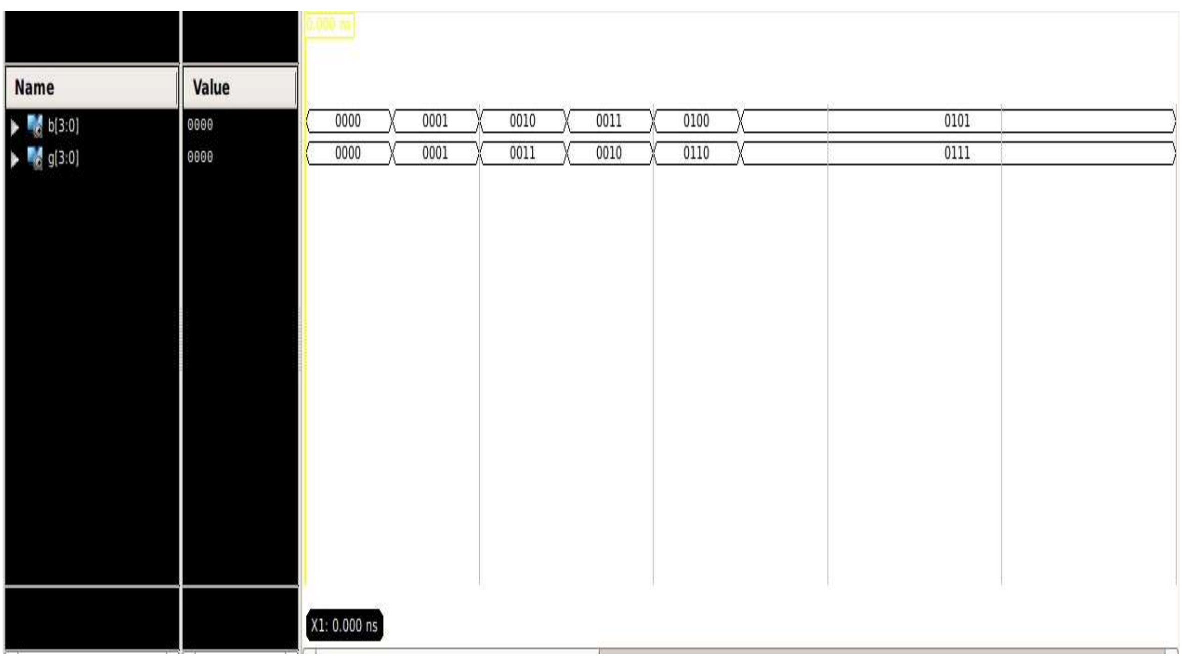
1. Behavioral Model Code:

```

entity binary_to_gray is
    Port ( B : in  STD_LOGIC_VECTOR (3 downto 0);
           G : out  STD_LOGIC_VECTOR (3 downto 0));
end binary_to_gray;

architecture Behavioral of binary_to_gray is

begin
process(B)
begin
G(3) <= B(3);
for i in 2 downto 0 loop
if(B(i+1) = B(i))
then
G(i) <= '0';
else
G(i) <= '1';
end if;
end loop;
end process;
end Behavioral;

```

2. Data flow Model Code:

```
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity binary_to_gray is
    Port ( B : in  STD_LOGIC_VECTOR (3 downto 0);
           G : out  STD_LOGIC_VECTOR (3 downto 0));
end binary_to_gray;

architecture Behavioral of binary_to_gray is

begin
G(3) <= B(3);
G(2) <= B(3) XOR B(2);
G(1) <= B(2) XOR B(1);
G(0) <= B(1) XOR B(0);
end Behavioral;

```

Output:



Objective: design and implement a **Gray to Binary Code Converter** using **VHDL** and verify its functionality using **Xilinx ISE**. The purpose of this conversion is to facilitate efficient digital communication, reduce errors in data transmission, and enable smooth processing in digital circuits such as rotary encoders, communication systems, and signal processing applications.

Theory:

Gray Code and Its Importance

Gray code is a **non-weighted, cyclic binary code** in which **only one-bit changes** between two consecutive numbers. This property makes gray code highly useful in applications where minimizing switching errors is crucial, such as **rotary encoders, analog-to-digital conversion, and asynchronous data transfer**. Since traditional binary numbers can result in multiple bits changing simultaneously during counting, gray code helps avoid glitches and transition errors.

Binary Code

Binary code is the standard numbering system used in digital electronics, where numbers are represented using only two digits: **0 and 1**. Unlike Gray code, **binary counting can result in multiple bit changes between consecutive values**, which can introduce errors in sensitive systems. Therefore, converting gray code back to Binary is essential for digital systems to process data accurately.

Gray to Binary Conversion Logic

The conversion from **Gray to Binary** follows a simple rule:

1. The **most significant bit (MSB) of the Binary number** is the same as the **MSB of the Gray code**.
2. Each subsequent **Binary bit** is obtained by performing an **XOR operation** between the previous **Binary bit** and the corresponding **gray bit**.

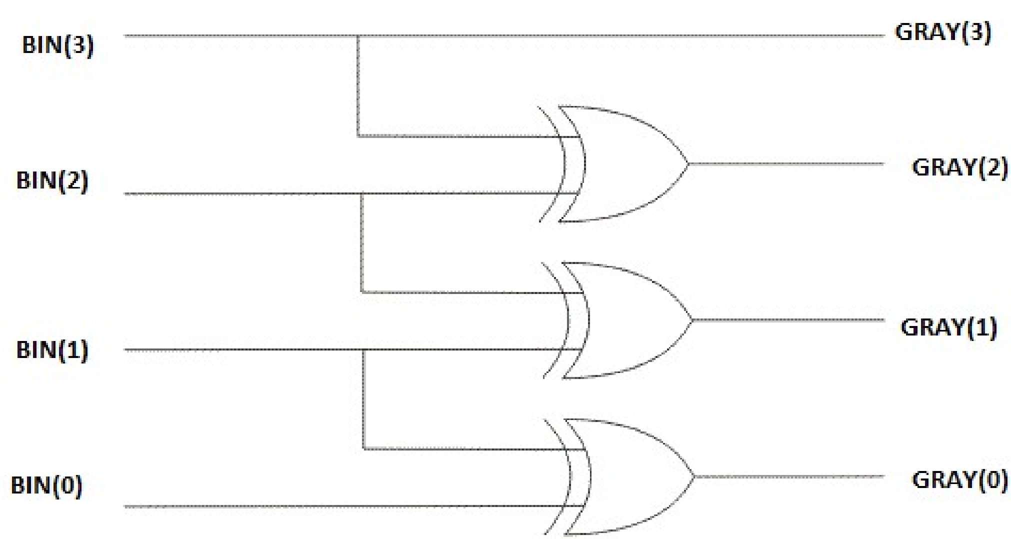
The conversion formula is:

- **Binary [MSB] = Gray [MSB]**
- **Binary[i] = Binary[i+1] $\oplus$ Gray[i]** (for all remaining bits)

Truth Table:

Gray Code (G3 G2 G1 G0)	**Binary Code (B3 B2 B1 B0)**
0000	0000
0001	0001
0011	0010
0010	0011
0110	0100
0111	0101
0101	0110
0100	0111
1100	1000
1101	1001
1111	1010
1110	1011
1010	1100
1011	1101
1001	1110
1000	1111

Data Flow Model:



Code:

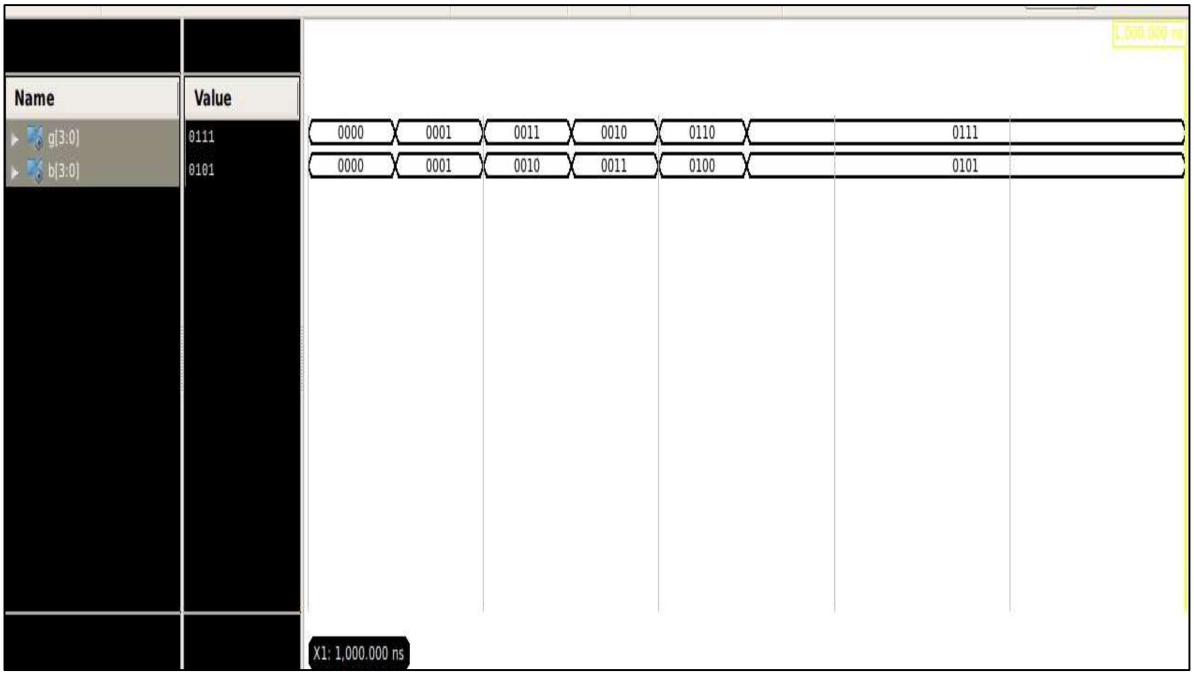
Data Flow Model code:

```
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity binary_to_gray is
    Port ( G : in  STD_LOGIC_VECTOR (3 downto 0);
           B : inout  STD_LOGIC_VECTOR (3 downto 0));
end binary_to_gray;

architecture Behavioral of binary_to_gray is

begin
B(3) <= G(3);
B(2) <= B(3) XOR G(2);
B(1) <= B(2) XOR G(1);
B(0) <= B(1) XOR G(0);
end Behavioral;
```

Behavioral Model Code:

```
begin
    process(G)
    begin
        B(3) <= G(3);  -- MSB remains the same
        for i in 2 downto 0 loop
            if (B(i+1) = G(i)) then
                B(i) <= '0';
            else
                B(i) <= '1';
            end if;
        end loop;
    end process;
end Behavioral;
```

Output:



Discussion:

The implementation of **Binary to Gray and Gray to Binary code converters** is crucial in digital logic design, particularly in applications such as error correction, digital communication, and hardware optimization. Gray code is widely used in minimizing switching errors in sequential circuits and reducing power consumption in hardware design. The Binary to Gray code conversion follows a straightforward rule where the **Most Significant Bit (MSB) remains the same**, and each subsequent **Gray code bit** is obtained by performing an **XOR operation** between the current binary bit and the previous binary bit. This method helps reduce bit transition errors, making it suitable for applications like **rotary encoders** and **digital communication systems**. In the **VHDL implementation**, a **process block** is used to iterate through the input binary vector, computing the corresponding Gray code bits using conditional statements. The conversion logic is synthesized and verified using **Xilinx ISE**, ensuring correct functionality and optimization for FPGA implementation. On the other hand, **Gray to Binary conversion** follows a slightly different approach where the **MSB remains unchanged**, and each **subsequent binary bit** is computed using an **XOR operation between the previous binary bit and the corresponding Gray code bit**. This ensures an accurate reversal of the Binary to Gray conversion. The VHDL implementation of this process also follows a structured approach, utilizing a loop to iterate through the input Gray code vector and compute the respective binary bits. By implementing and testing these designs in **Xilinx ISE**, we verify the correctness of the conversion logic and ensure that they function as expected when synthesized for FPGA hardware. These conversions play a significant role in digital systems, where efficient encoding and decoding of data are required to minimize errors and enhance system reliability.

<u>Assignment 5:</u> Implementation of 4-bit comparator, 2:4 Decoder, 3:8 Decoder using Xilinx ISE.

<u>Software used:</u>

Property Name	Value
Device family	Spartan3
Device	XC3550
Package	PQ208
Speed	-5
Top-level source type	HDL
Synthesis Tool	XST(VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL

<u>Objective:</u> To design and implement a **4-bit comparator** using **VHDL in Xilinx ISE**, which compares two 4-bit binary numbers and determines whether one is greater than, equal to, or less than the other. The goal is to understand binary comparison logic and its application in digital systems.

<u>Theory:</u>

A **4-bit comparator** is a combinational circuit that compares two **4-bit binary numbers** and determines their magnitude relationship. The circuit has two inputs, **A (A3, A2, A1, A0)** and **B (B3, B2, B1, B0)**, representing two 4-bit numbers. The comparator produces three outputs:

- **A > B** (High when A is greater than B)
- **A = B** (High when A is equal to B)
- **A < B** (High when A is less than B)

The comparison begins from the **Most Significant Bit (MSB) down to the Least Significant Bit (LSB)**. The logic behind the comparison is as follows:

1. If **A3 > B3**, then **A > B**, and no further comparison is needed.
2. If **A3 = B3**, then compare **A2 and B2**. The process continues down to A0 and B0.
3. If all corresponding bits are equal, then **A = B**.
4. If **A3 < B3**, then **A < B**, and the remaining bits do not need to be checked.

The comparator logic can be implemented using **basic logic gates (AND, OR, XOR, NOT)** or using a conditional structure in **VHDL**. The **XOR gate** is used to check equality, while **AND-OR logic** is used to determine the greater and lesser conditions.

Logical Expressions for a 4-bit Comparator

- **Equality Check:**

  $$A=B \Rightarrow (A3 \oplus B3)' \cdot (A2 \oplus B2)' \cdot (A1 \oplus B1)' \cdot (A0 \oplus B0)'$$

- **A > B Condition:**

  $$A3\,B3' + (A3 \oplus B3)'\,(A2B2') + (A3 \oplus B3)'\,(A2 \oplus B2)'\,(A1B1') + (A3 \oplus B3)'\,(A2 \oplus B2)'\,(A1 \oplus B1)'\,(A0B0')$$

- **A < B Condition:**

  $$A3'B3+(A3 \oplus B3)'(A2'B2)+(A3 \oplus B3)'(A2 \oplus B2)'(A1'B1)+(A3 \oplus B3)'(A2 \oplus B2)'(A1 \oplus B1)'(A0'B0)$$

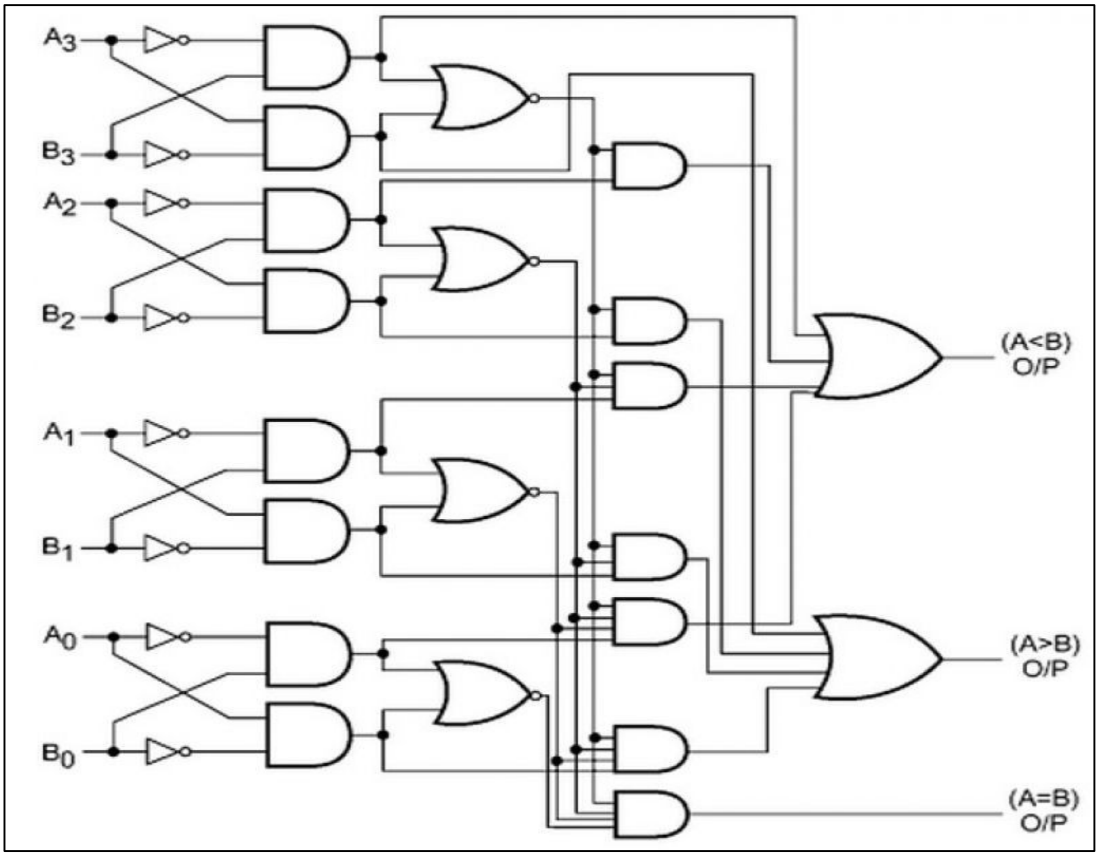
The **VHDL implementation** of a **4-bit comparator** involves defining an **entity** for inputs and outputs, and using an **if-else statement inside a process block** to implement the comparison logic. The circuit is **tested through simulation** in Xilinx ISE to verify correct operation before synthesizing for FPGA implementation.

The **4-bit comparator** is widely used in **arithmetic circuits, sorting algorithms, data processing units, microprocessors, and control systems** where binary numbers need to be compared for decision-making.

Truth Table:

A3	A2	A1	A0	B3	B2	B1	B0	A > B	A = B	A < B
0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	1	0	0	1	0
0	1	0	0	0	1	0	1	0	0	1
0	1	1	1	0	1	1	1	0	1	0
1	0	0	0	0	1	1	1	1	0	0
1	0	0	1	1	0	0	0	0	0	1
1	0	1	0	1	0	1	0	0	1	0
1	1	0	0	1	1	0	1	0	0	1
1	1	1	1	1	1	1	1	0	1	0

Data Flow Model:



Code:

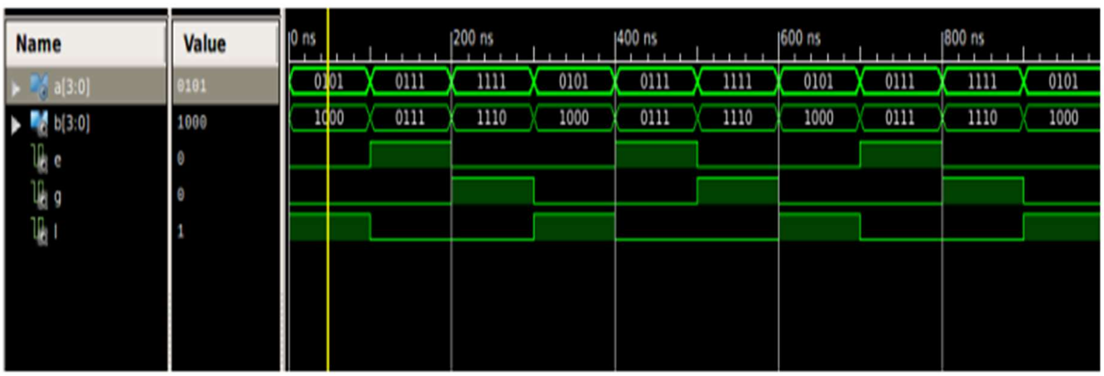
Data Flow Model Code:

```
end four_bit_comparator;

architecture Dataflow of four_bit_comparator is
begin
    E <= (not (A(3) xor B(3))) and
         (not (A(2) xor B(2))) and
         (not (A(1) xor B(1))) and
         (not (A(0) xor B(0)));

    G <= (A(3) and not B(3)) or
         (not (A(3) xor B(3)) and A(2) and not B(2)) or
         (not (A(3) xor B(3)) and not (A(2) xor B(2)) and A(1) and not B(1)) or
         (not (A(3) xor B(3)) and not (A(2) xor B(2)) and not (A(1) xor B(1)) and A(0) and not B(0));

    L <= (not A(3) and B(3)) or
         (not (A(3) xor B(3)) and not A(2) and B(2)) or
         (not (A(3) xor B(3)) and not (A(2) xor B(2)) and not A(1) and B(1)) or
         (not (A(3) xor B(3)) and not (A(2) xor B(2)) and not (A(1) xor B(1)) and not A(0) and B(0));
end Dataflow;
```

Behavioral Model code:

```
end four_bit_comparator;

architecture Behavioral of four_bit_comparator is

begin
process(A,B)
begin
   if(A>B) then
   E <= '0';
   L <= '0';
   G <= '1';

   elsif(A=B) then
   E <= '1';
   L <= '0';
   G <= '0';
   else
   E <= '0';
   L <= '1';
   G <= '0';
   end if;
end process;
end Behavioral;
```

Output:



Objective: The objective of this experiment is to design and implement a **2-to-4 Decoder** using **VHDL** in **Xilinx ISE**. The aim is to understand how a decoder works by converting **2-bit binary inputs** into **four distinct output lines**, where each output corresponds to one of the possible input combinations. The implementation will be tested through simulation and synthesis on an FPGA.

Theory:

A **decoder** is a combinational circuit that converts **binary input data** into a **one-hot output format**, meaning that for every unique input combination, exactly **one output** is activated while the rest remain low. A **2-to-4 decoder** takes a **2-bit binary input** and produces **four distinct outputs**, making it useful in memory addressing, instruction decoding, and other digital logic applications.

The **block diagram** of a 2:4 decoder consists of:

- **2 input lines (A1, A0)** representing a 2-bit binary number.
- **4 output lines (Y3, Y2, Y1, Y0)**, each representing a unique decoded value.

Logic Expressions for a 2:4 Decoder

Each output line is activated based on the input combination:

- **Y0 = A1' A0'** → Active when A = `00`
- **Y1 = A1' A0** → Active when A = `01`
- **Y2 = A1 A0'** → Active when A = `10`
- **Y3 = A1 A0** → Active when A = `11`

Here, **A' (NOT A)** represents the complement of A.

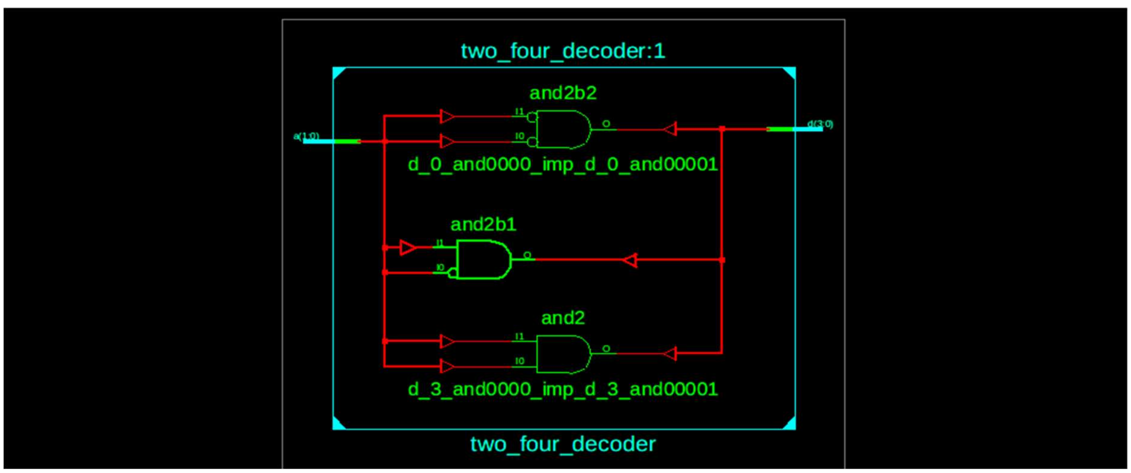
Working of the Decoder:

- When **A1 = 0, A0 = 0**, only **Y0** is high (`1`), and others are `0`.
- When **A1 = 0, A0 = 1**, only **Y1** is high (`1`), and others are `0`.
- When **A1 = 1, A0 = 0**, only **Y2** is high (`1`), and others are `0`.
- When **A1 = 1, A0 = 1**, only **Y3** is high (`1`), and others are `0`.

Truth Table:

A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Data flow Model:



Code:

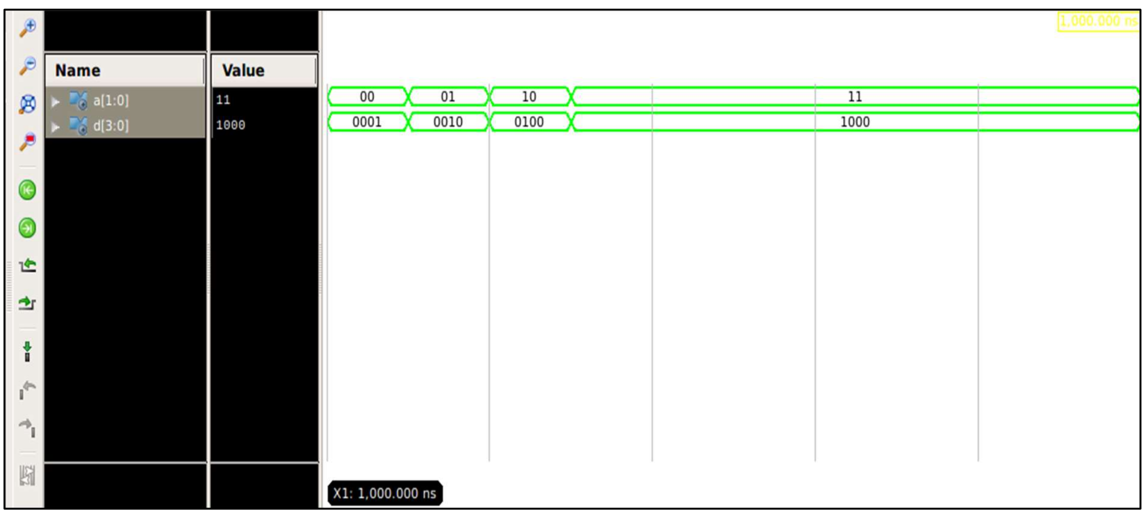
1. Behavioral Model Code:

```
--use UNISIM.VComponents.all;

entity two_four_decoder is
    Port ( a : in  STD_LOGIC_VECTOR (1 downto 0);
           d : out  STD_LOGIC_VECTOR (3 downto 0));
end two_four_decoder;

architecture Behavioral of two_four_decoder is

begin
process(a)
variable S : integer;
begin
S := conv_integer(a);
for i in 0 to 3 loop
if(i = s) then
d(i) <= '1';
else
d(i) <= '0';
end if;
end loop;
end process;
end Behavioral;
```

2. Data flow Model Code:

```
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity two_four_decoder is
    Port ( a : in  STD_LOGIC_VECTOR (1 downto 0);
           d : out  STD_LOGIC_VECTOR (3 downto 0));
end two_four_decoder;

architecture Behavioral of two_four_decoder is

begin
d(0) <= (not a(1)) and (not a(0));
d(1) <= (not a(1)) and (a(0));
d(2) <= (a(1)) and (not a(0));
d(3) <= (a(1)) and (a(0));

end Behavioral;
```

Output:



Objective:

To design and implement a **3:8 Decoder** using Xilinx ISE, where a **three-bit binary input** is used to activate one of the **eight outputs**, demonstrating the working of a combinational decoder circuit.

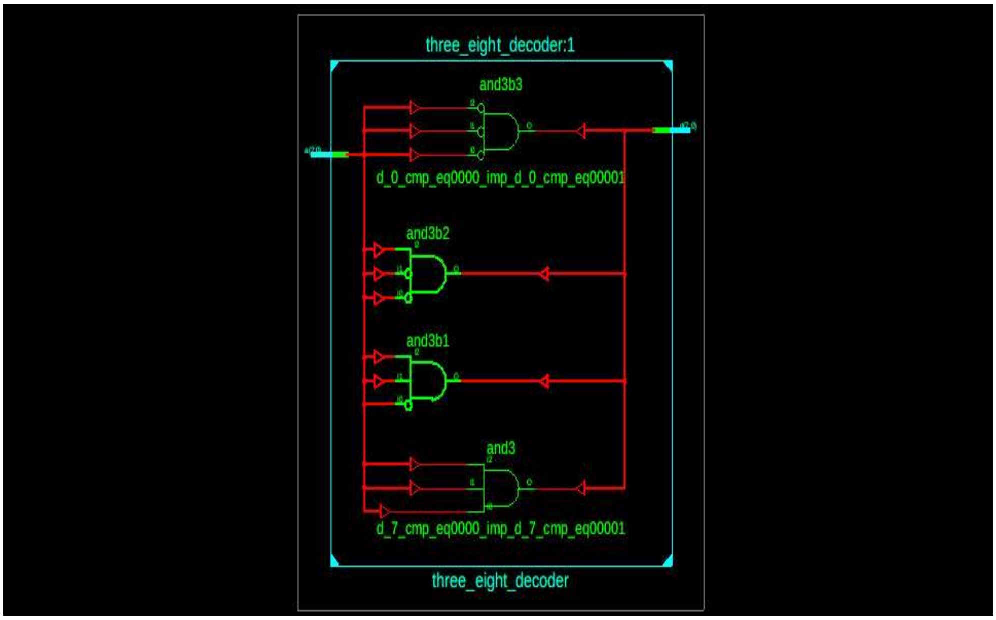
Theory:

A **3-to-8 decoder** is a combinational logic circuit that takes a 3-bit binary input and produces eight distinct output lines, with only one active at a time. It is used in digital systems to convert binary-coded data into a one-hot output format, where each unique input combination corresponds to exactly one high output while all others remain low. The decoder plays a crucial role in memory addressing, data selection, instruction decoding, and various control applications in microprocessors and digital circuits. It operates by logically ANDing the input bits and their complements to produce the required output pattern, ensuring that only one output line is enabled for each unique input combination. The circuit consists of three input lines, which can represent eight different states, and each state corresponds to activating one of the eight outputs. The 3:8 decoder can be implemented using basic logic gates, such as AND and NOT gates, where each output is derived from a specific combination of the input bits. Additionally, the decoder can be modified to work in an active-low configuration using NAND gates, where the selected output goes low instead of high. The design and implementation of a 3:8 decoder in **Xilinx ISE** involve defining the inputs and outputs in VHDL or Verilog, specifying the logic equations for each output, and synthesizing the design for FPGA or hardware verification. Decoders are extensively used in memory address decoding, where they enable specific memory locations in response to binary address inputs, allowing microprocessors to access different memory blocks efficiently. They are also used in instruction decoding, where specific operations are selected based on opcode values. In digital display controllers, such as 7-segment displays, decoders help determine which segment should be illuminated based on the given input. Furthermore, they play a role in multiplexing and demultiplexing circuits, allowing multiple signals to be directed to different locations based on control inputs. Overall, the **3-to-8 decoder** is a fundamental component in digital circuit design, enabling efficient binary-to-output conversion for a wide range of practical applications in computing and communication systems.

Truth Table:

A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Data flow Model:



Code:

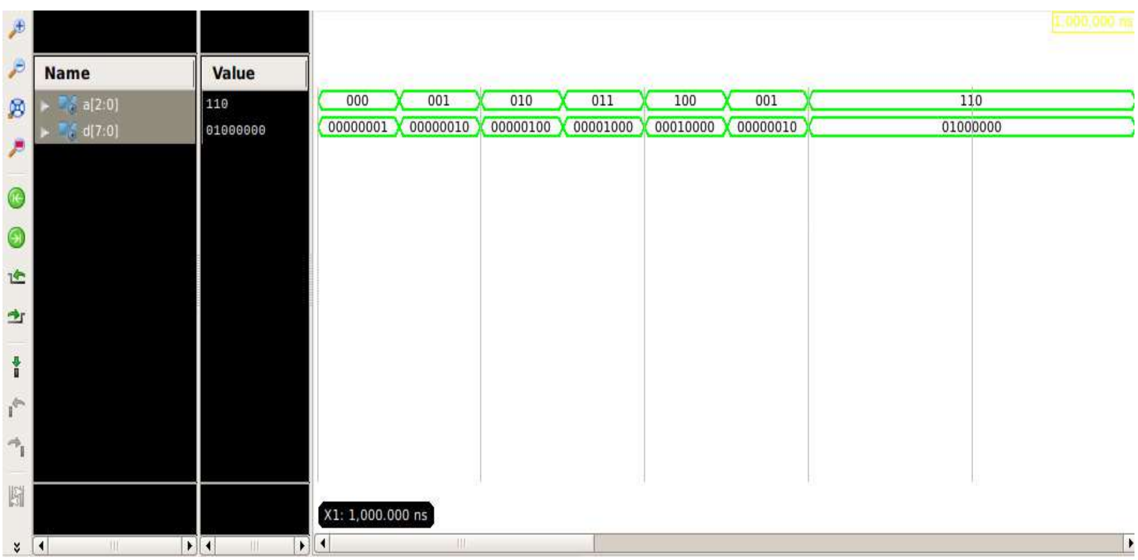
1. Behavioral Model Code:

```
entity three_eight_decoder is
    Port ( a : in  STD_LOGIC_VECTOR (2 downto 0);
           d : out  STD_LOGIC_VECTOR (7 downto 0));
end three_eight_decoder;

architecture Behavioral of three_eight_decoder is

begin
process(a)
variable S : integer;
begin
S := conv_integer(a);
for i in 0 to 7 loop
if(i = s) then
d(i) <= '1';
else
d(i) <= '0';
end if;
end loop;
end process;
end Behavioral;
```

2. Data flow Model Code:

```
architecture Dataflow of Decoder3to8 is
begin
    Y(0) <= not A(2) and not A(1) and not A(0);
    Y(1) <= not A(2) and not A(1) and A(0);
    Y(2) <= not A(2) and A(1) and not A(0);
    Y(3) <= not A(2) and A(1) and A(0);
    Y(4) <= A(2) and not A(1) and not A(0);
    Y(5) <= A(2) and not A(1) and A(0);
    Y(6) <= A(2) and A(1) and not A(0);
    Y(7) <= A(2) and A(1) and A(0);
end Dataflow;
```

Output:



Discussion:

The implementation of a **4-bit comparator, 2:4 decoder, and 3:8 decoder using Xilinx ISE** provides a fundamental understanding of combinational circuits and their role in digital systems. These circuits are essential building blocks in processors, memory systems, and various digital logic applications.

A **4-bit comparator** is a combinational circuit used to compare two 4-bit binary numbers. It determines whether one number is **greater than, equal to, or less than** the other. The comparison starts from the **most significant bit (MSB)** and proceeds to the **least significant bit (LSB)** to make logical decisions. The implementation in VHDL utilizes basic **AND, OR, XOR** gates or conditional statements to generate the output based on the comparison results. Comparators are widely used in **arithmetic operations, sorting algorithms, and control systems** where decision-making is required.

The **2:4 decoder** is a combinational logic circuit that takes a **2-bit binary input** and activates only one of the **four output lines** at a time. This circuit is crucial in applications such as **memory addressing, data selection, and instruction decoding in microprocessors**. Using the **"case" statement in VHDL**, the design can be efficiently implemented with structured logic. Each input combination corresponds to an active output line, ensuring that only one output is high at any given time.

The **3:8 decoder** extends the concept of a decoder by taking a **3-bit input** and producing **eight distinct outputs**, with only one being active at a time. This circuit is widely used in **larger address decoding schemes, multiplexing, and microcontroller-based applications**. The implementation in VHDL uses Boolean expressions or a **case structure**, making it easy to synthesize and test using Xilinx ISE. By enabling different output lines based on the input values, the decoder facilitates effective **signal routing and data selection**.

Through Xilinx ISE, these circuits were **designed, synthesized, simulated, and tested** for correctness. The simulation waveforms confirm the expected behavior, verifying that the logic is correctly implemented. This experiment enhances the understanding of **combinational logic, hierarchical design, and FPGA-based implementation**, providing a strong foundation for more complex digital circuit designs.

Assignment 6: Implementation of 2:4 Decoder (using case), 4:2 Encoder (Data flow and using case), 8:3 Encoder (using Loop).

Software used:

Property Name	Value
Device family	Spartan3
Device	XC3550
Package	PQ208
Speed	-5
Top-level source type	HDL
Synthesis Tool	XST(VHDL/Verilog)
Simulator	ISim (VHDL/Verilog)
Preferred Language	VHDL

Objective: The objective of this experiment is to design and implement a **2-to-4 Decoder using the "case" statement** in VHDL. The decoder will take a **2-bit binary input** and produce a **4-bit output**, where only one output line is active (logic '1') at a time based on the input combination. The design will be simulated and synthesized using **Xilinx ISE** to verify its correctness and practical functionality.

Theory:

A **2-to-4 decoder** is a combinational circuit that takes **two input bits** and decodes them into **one of four possible output lines**. Only one output is **high (1)** at any given time, while all others remain **low (0)**. This type of decoder is widely used in digital systems for applications such as memory address decoding, data selection, and control signal generation.

In this implementation, the **"case" statement** in VHDL is used to describe the behavior of the decoder. The **case statement** provides an efficient and structured way to assign output values based on different input conditions. It simplifies the logic by eliminating the need for multiple Boolean expressions.

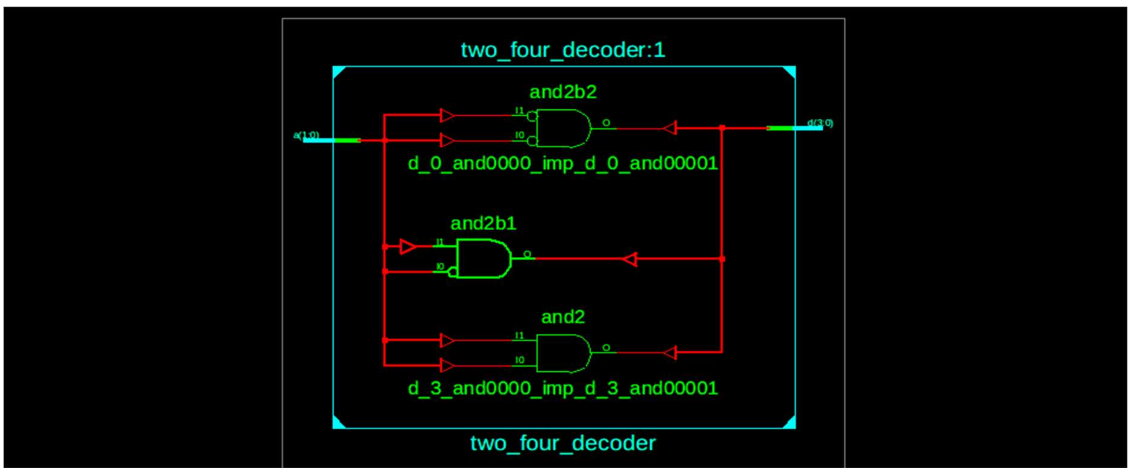
The working of a **2-to-4 decoder** is explained as follows:

- The **2-bit input (A1, A0)** determines which one of the **four outputs (Y3, Y2, Y1, Y0)** is activated.
- The VHDL implementation using a **case statement** assigns each output based on the input value, making the design more readable and structured. This method avoids redundant conditional checks, making it **efficient for FPGA synthesis**.

Truth Table:

A1	A0	Y3	Y2	Y1	Y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

Data flow Model:



Code:

1. Behavioral Model Code:

```
--use UNISIM.VComponents.all;

entity two_four_decoder is
    Port ( a : in  STD_LOGIC_VECTOR (1 downto 0);
           d : out  STD_LOGIC_VECTOR (3 downto 0));
end two_four_decoder;

architecture Behavioral of two_four_decoder is

begin
process(a)
variable S : integer;
begin
S := conv_integer(a);
for i in 0 to 3 loop
if(i = s) then
d(i) <= '1';
else
d(i) <= '0';
end if;
end loop;
end process;
end Behavioral;
```

2. Data flow Model Code:

```
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity two_four_decoder is
    Port ( a : in  STD_LOGIC_VECTOR (1 downto 0);
           d : out  STD_LOGIC_VECTOR (3 downto 0));
end two_four_decoder;

architecture Behavioral of two_four_decoder is

begin
d(0) <= (not a(1)) and (not a(0));
d(1) <= (not a(1)) and (a(0));
d(2) <= (a(1)) and (not a(0));
d(3) <= (a(1)) and (a(0));

end Behavioral;
```

Output: