Project Report

Introduction

This report presents the design, implementation, and testing of a digital communication system using VHDL (VHSIC Hardware Description Language). The system consists of a Transmitter, a Receiver, and a Top Module integrating both. The goal is to transmit data with error detection and correction capabilities using Hamming code. This report includes explanations of the Hamming code algorithm, the VHDL code for each module, their respective testbenches, and the simulation results.

Hamming Code Overview

Hamming code is a type of binary error-correcting code that can detect and correct single-bit errors. It is widely used in digital communication systems to ensure data integrity. The key features of Hamming code include:

- **Parity Bits**: Extra bits added to the original data to help detect and correct errors.

- **Error Detection and Correction**: Hamming code can detect and correct single-bit errors and detect double-bit errors.

Hamming Code Generation

For an 8-bit data word, Hamming code adds 4 parity bits to create a 12-bit encoded word. The positions of the parity bits are powers of 2 (1, 2, 4, 8). The parity bits are calculated as follows:

- **P1 (parity bit for positions 1, 3, 5, 7, 9, 11):** $P1 = D7 \oplus D6 \oplus D4 \oplus D3 \oplus D1$

- **P2 (parity bit for positions 2, 3, 6, 7, 10, 11):** $P2 = D7 \oplus D5 \oplus D4 \oplus D2 \oplus D1$

- **P4 (parity bit for positions 4, 5, 6, 7, 12):** $P4 = D6 \oplus D5 \oplus D4 \oplus D0$

- **P8 (parity bit for positions 8, 9, 10, 11, 12):** $P8 = D3 \oplus D2 \oplus D1 \oplus D0$

System Design

The digital communication system consists of three main components:

1. **Transmitter:** Encodes the data with Hamming code, introduces a noise bit, and transmits the data serially.

2. **Receiver:** Receives the serial data, detects and corrects errors using Hamming code, and extracts the original data.

3. **Top Module**: Integrates the Transmitter and Receiver to simulate the complete communication system.

Transmitter Module

The Transmitter module encodes the 8-bit data into a 12-bit Hamming code, introduces noise, and transmits the data serially.

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Transmitter is
    Port (
        clk            : in  STD_LOGIC;
        reset          : in  STD_LOGIC;
        En             : in  STD_LOGIC;
        ready          : out STD_LOGIC;
        header_in      : in  STD_LOGIC_VECTOR (7 downto 0);
        data_in        : in  STD_LOGIC_VECTOR (7 downto 0);
        footer_in      : in  STD_LOGIC_VECTOR (7 downto 0);
        Tx             : out STD_LOGIC;
        noise_bit      : in  integer range 0 to 11
    );
end Transmitter;

architecture Behavioral of Transmitter is
    -- State type definition
    type state_type is (IDLE, Noise, TRANSMIT, DONE);
    signal state        : state_type := IDLE;  -- Current state
```

```
    signal bit_count    : integer range 0 to 27 := 0;  -- Bit counter
    signal shift_reg    : STD_LOGIC_VECTOR (27 downto 0) := (others => '0');
-- Shift register to hold the data
    signal encoded_data : STD_LOGIC_VECTOR (11 downto 0) := (others => '0');
-- 12-bit encoded data

    -- Function to calculate parity bits
    function calculate_parity(d: STD_LOGIC_VECTOR(7 downto 0)) return
STD_LOGIC_VECTOR is
        variable p: STD_LOGIC_VECTOR(3 downto 0);
    begin
        p(0) := d(7) xor d(6) xor d(4) xor d(3) xor d(1);
        p(1) := d(7) xor d(5) xor d(4) xor d(2) xor d(1);
        p(2) := d(6) xor d(5) xor d(4) xor d(0);
        p(3) := d(3) xor d(2) xor d(1) xor d(0);
        return p;
    end function;

begin

    -- Main process
    process(clk, reset)
    begin
        if reset = '1' then
            -- Reset all signals
            state <= IDLE;
            bit_count <= 0;
            shift_reg <= (others => '0');
            encoded_data <= (others => '0');
            ready <= '0';
            Tx <= '0';
        elsif rising_edge(clk) then
            case state is
                when IDLE =>
                    ready <= '0';
                    Tx <= '0';
                    -- Calculate parity bits and encode data
                    encoded_data(11) <= calculate_parity(data_in)(0);
                    encoded_data(10) <= calculate_parity(data_in)(1);
                    encoded_data(9)  <= data_in(7);
                    encoded_data(8)  <= calculate_parity(data_in)(2);
                    encoded_data(7)  <= data_in(6);
                    encoded_data(6)  <= data_in(5);
                    encoded_data(5)  <= data_in(4);
                    encoded_data(4)  <= calculate_parity(data_in)(3);
                    encoded_data(3)  <= data_in(3);
                    encoded_data(2)  <= data_in(2);
                    encoded_data(1)  <= data_in(1);
                    encoded_data(0)  <= data_in(0);
                    encoded_data(noise_bit) <= not encoded_data(noise_bit);
                    if En = '1' then
                        -- Concatenate header, encoded data, and footer
                        shift_reg <= header_in & encoded_data & footer_in;
                        bit_count <= 0;
                        ready <= '0';
                        state <= Noise;
                    end if;
```

```
                when Noise =>
                    -- Add noise bit
                    shift_reg(8 + noise_bit) <= not shift_reg(8 + noise_bit);
                    state <= TRANSMIT;

                when TRANSMIT =>
                    -- Transmit the data bit by bit
                    ready <= '1';
                    Tx <= shift_reg(27);
                    shift_reg <= shift_reg(26 downto 0) & '0';
                    bit_count <= bit_count + 1;
                    if bit_count = 27 then
                        state <= DONE;
                    end if;

                when DONE =>
                    -- Transmission complete
                    Tx <= '0';
                    ready <= '0';
                    state <= IDLE;

                when others =>
                    state <= IDLE;
            end case;
        end if;
    end process;

end Behavioral;
```

Explanation

- **State Definition**: Defines the states IDLE, Noise, TRANSMIT, and DONE.

- **calculate_parity Function**: Calculates the parity bits for the given data.

- **Main Process**: Handles the state transitions and performs the encoding, noise addition, and transmission.

Transmitter Testbench

The testbench verifies the functionality of the Transmitter module.

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Transmitter_tb is
end Transmitter_tb;

architecture Behavioral of Transmitter_tb is

    -- Component Declaration for the Unit Under Test (UUT)
    component Transmitter
        Port (
            clk         : in  STD_LOGIC;
            reset       : in  STD_LOGIC;
            En          : in  STD_LOGIC;
            ready       : out STD_LOGIC;
            header_in   : in  STD_LOGIC_VECTOR (7 downto 0);
            data_in     : in  STD_LOGIC_VECTOR (7 downto 0);
            footer_in   : in  STD_LOGIC_VECTOR (7 downto 0);
            Tx          : out STD_LOGIC;
            noise_bit   : in  integer range 0 to 11
        );
    end component;

    -- Inputs
    signal clk       : STD_LOGIC := '0';
    signal reset     : STD_LOGIC := '0';
    signal En        : STD_LOGIC := '0';
    signal header_in : STD_LOGIC_VECTOR (7 downto 0) := "10101010";
    signal data_in   : STD_LOGIC_VECTOR (7 downto 0) := "10011110";
    signal footer_in : STD_LOGIC_VECTOR (7 downto 0) := "01010101" ;
    signal noise_bit : integer range 0 to 11;

    -- Outputs
    signal ready            : STD_LOGIC;
    signal Tx               : STD_LOGIC;

    -- Clock period definition
    constant clk_period : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: Transmitter Port map (
            clk => clk,
            reset => reset,
            En => En,
```

```
        ready => ready,
        header_in => header_in,
        data_in => data_in,
        footer_in => footer_in,
        Tx => Tx,
        noise_bit => noise_bit
    );

-- Clock process definitions
clk_process :process
begin
    clk <= '0';
    wait for clk_period/2;
    clk <= '1';
    wait for clk_period/2;
end process;

-- Stimulus process
stim_proc: process
begin
    -- Initialize Inputs
    reset <= '1';
    wait for clk_period*1.5;

    reset <= '0';
    wait for clk_period*2;

    -- First Test Case
    noise_bit <= 2;
    wait for clk_period*2;
    header_in <= "10101010";
    data_in <= "10011110";
    footer_in <= "01010101";
    wait for clk_period*4;
    En <= '1';


    -- Wait for the first transmission to complete
    wait for clk_period*30;

    En <= '0';

    -- Apply reset before the second test case
    reset <= '1';
      noise_bit <= 0;
    wait for clk_period*2;

    reset <= '0';
    wait for clk_period*2;

    -- Second Test Case
    noise_bit <= 2;
    wait for clk_period*2;
    header_in <= "11110000";
    data_in <= "01100110";
    footer_in <= "00001111";
    wait for clk_period*4;
```

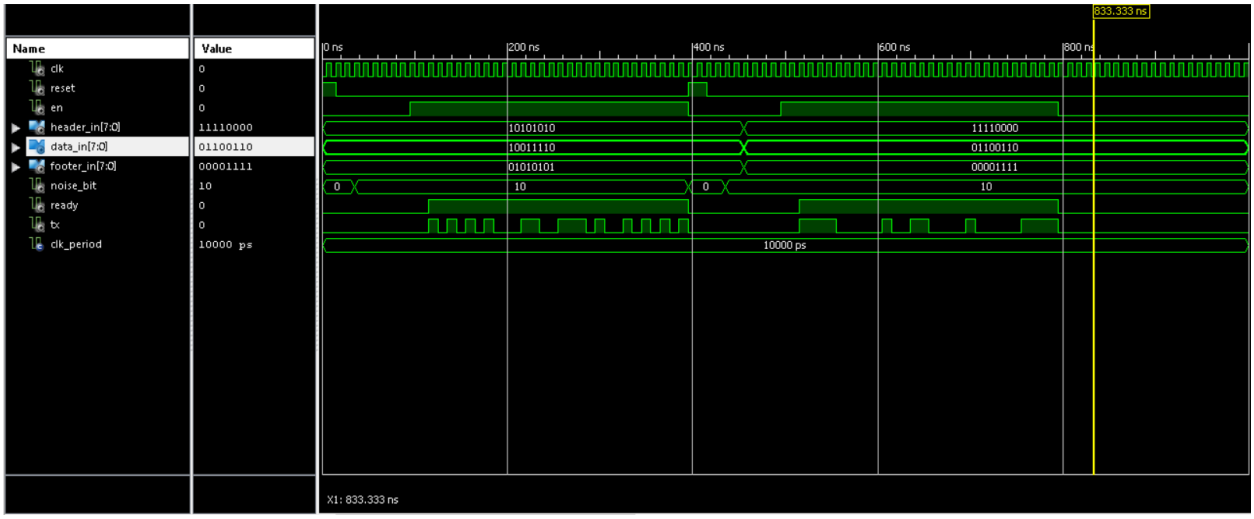
```
        En <= '1';


        -- Wait for the Second transmission to complete
        wait for clk_period*30;

        En <= '0';

        -- Stop simulation
        wait;
    end process;

end Behavioral;
```

Explanation

- Initialization: Initializes the inputs and generates a clock signal.

- Stimulus Process: Applies test cases to the Transmitter module and verifies the output.

Results



The values of Tx for each item are shown in the figure above (note the values of Tx when ready is equal to 1).

Receiver Module

The Receiver module receives the serial data, detects and corrects errors using Hamming code, and extracts the original data.

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;  -- Use only IEEE.NUMERIC_STD for type conversions

entity Receiver is
    Port (
        clk                         : in  STD_LOGIC;
        reset                       : in  STD_LOGIC;
        En                          : in  STD_LOGIC;
        Rx                          : in  STD_LOGIC;
        header_out                  : out STD_LOGIC_VECTOR (7 downto 0);
        encoded_data_out            : out STD_LOGIC_VECTOR (11 downto 0);
        footer_out                  : out STD_LOGIC_VECTOR (7 downto 0);
        ready                       : out STD_LOGIC;
        debug_shift_reg             : out STD_LOGIC_VECTOR (27 downto 0);  --
Debug signal for shift register
        corrected_data_out          : out STD_LOGIC_VECTOR (7 downto 0);  --
Signal for corrected data
        corrected_encoded_data_out : out STD_LOGIC_VECTOR (11 downto 0);  --
Signal for corrected encoded data
        corrupted_bit_out           : out STD_LOGIC_VECTOR (3 downto 0);  --
Signal for corrupted bit
        new_parity_out              : out STD_LOGIC_VECTOR (3 downto 0);  --
Signal for new parity bits
        old_parity_out              : out STD_LOGIC_VECTOR (3 downto 0)    --
Signal for old parity bits
    );
end Receiver;

architecture Behavioral of Receiver is
    -- State type definition
    type state_type is (IDLE, RECEIVE, DONE, CHECK, CHECK2, CORRECTION);
    signal state                    : state_type := IDLE;  -- Current
state
    signal bit_count                : integer range 0 to 30 := 0;  -- Bit
counter
    signal shift_reg                : STD_LOGIC_VECTOR (27 downto 0) :=
(others => '0');  -- Shift register to hold received data
    signal corrupted_bit            : STD_LOGIC_VECTOR (3 downto 0) :=
(others => '0');  -- Corrupted bit location
    signal new_parity               : STD_LOGIC_VECTOR (3 downto 0) :=
(others => '0');  -- New parity bits
```

```
    signal old_parity                  : STD_LOGIC_VECTOR (3 downto 0) :=
(others => '0');  -- Old parity bits
    signal encoded_data_internal       : STD_LOGIC_VECTOR (11 downto 0) :=
(others => '0');  -- Internal encoded data
    signal corrected_data_internal     : STD_LOGIC_VECTOR (7 downto 0) :=
(others => '0');  -- Internal corrected data
    signal corrected_encoded_data_internal : STD_LOGIC_VECTOR (11 downto 0)
:= (others => '0');  -- Internal corrected encoded data

    -- Function to calculate parity bits
    function calculate_parity(d: STD_LOGIC_VECTOR(7 downto 0)) return
STD_LOGIC_VECTOR is
        variable p: STD_LOGIC_VECTOR(3 downto 0);
    begin
        p(0) := d(7) xor d(6) xor d(4) xor d(3) xor d(1);
        p(1) := d(7) xor d(5) xor d(4) xor d(2) xor d(1);
        p(2) := d(6) xor d(5) xor d(4) xor d(0);
        p(3) := d(3) xor d(2) xor d(1) xor d(0);
        return p;
    end function;

begin
    -- Assign debug signals
    debug_shift_reg <= shift_reg;
    encoded_data_out <= encoded_data_internal;
    corrected_data_out <= corrected_data_internal;
    corrected_encoded_data_out <= corrected_encoded_data_internal;
    corrupted_bit_out <= corrupted_bit;
    new_parity_out <= new_parity;
    old_parity_out <= old_parity;

    -- Main process
    process(clk, reset)
    begin
        if reset = '1' then
            -- Reset all signals
            state <= IDLE;
            bit_count <= 0;
            shift_reg <= (others => '0');
            header_out <= (others => '0');
            encoded_data_internal <= (others => '0');
            corrected_data_internal <= (others => '0');
            corrected_encoded_data_internal <= (others => '0');
            footer_out <= (others => '0');
            ready <= '0';
            corrupted_bit <= "0000";
            new_parity <= (others => '0');
            old_parity <= (others => '0');
        elsif rising_edge(clk) then
            if En = '1' then
                case state is
                    when IDLE =>
                        ready <= '0';
                        if Rx = '1' or Rx = '0' then  -- Start receiving on
any change on Rx
                            state <= RECEIVE;
                            bit_count <= 0;
```

Explanation

- **State Definition:** Defines the states IDLE, RECEIVE, DONE, CHECK, CHECK2, and CORRECTION.

- **calculate_parity Function:** Calculates the parity bits for the given data.

- **Main Process:** Handles the state transitions and performs the reception, error detection, and correction.

Receiver Testbench

The testbench verifies the functionality of the Receiver module.

VHDL Code

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Receiver_tb is
end Receiver_tb;

architecture Behavioral of Receiver_tb is

    -- Component Declaration for the Unit Under Test (UUT)
    component Receiver
        Port (
            clk                         : in  STD_LOGIC;
            reset                       : in  STD_LOGIC;
            En                          : in  STD_LOGIC;
            Rx                          : in  STD_LOGIC;
            header_out                  : out STD_LOGIC_VECTOR (7 downto 0);
            encoded_data_out            : out STD_LOGIC_VECTOR (11 downto 0);
            footer_out                  : out STD_LOGIC_VECTOR (7 downto 0);
            ready                       : out STD_LOGIC;
            debug_shift_reg             : out STD_LOGIC_VECTOR (27 downto 0);
-- Debug signal for shift register
            corrected_data_out          : out STD_LOGIC_VECTOR (7 downto 0);
-- Signal for corrected data
            corrected_encoded_data_out : out STD_LOGIC_VECTOR (11 downto 0);
-- Signal for corrected encoded data
            corrupted_bit_out           : out STD_LOGIC_VECTOR (3 downto 0);
-- Signal for corrupted bit
            new_parity_out              : out STD_LOGIC_VECTOR (3 downto 0);
-- Signal for new parity bits
            old_parity_out              : out STD_LOGIC_VECTOR (3 downto 0)
-- Signal for old parity bits
        );
    end component;

    -- Inputs
    signal clk                         : STD_LOGIC := '0';
    signal reset                       : STD_LOGIC := '0';
    signal En                          : STD_LOGIC := '0';
    signal Rx                          : STD_LOGIC := '0';

    -- Outputs
    signal header_out                  : STD_LOGIC_VECTOR (7 downto 0);
```

```
    signal encoded_data_out           : STD_LOGIC_VECTOR (11 downto 0);
    signal footer_out                 : STD_LOGIC_VECTOR (7 downto 0);
    signal ready                      : STD_LOGIC;
    signal debug_shift_reg            : STD_LOGIC_VECTOR (27 downto 0);
    signal corrected_data_out         : STD_LOGIC_VECTOR (7 downto 0);
    signal corrected_encoded_data_out : STD_LOGIC_VECTOR (11 downto 0);
    signal corrupted_bit_out          : STD_LOGIC_VECTOR (3 downto 0);
    signal new_parity_out             : STD_LOGIC_VECTOR (3 downto 0);
    signal old_parity_out             : STD_LOGIC_VECTOR (3 downto 0);

    -- Clock period definition
    constant clk_period : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: Receiver Port map (
            clk => clk,
            reset => reset,
            En => En,
            Rx => Rx,
            header_out => header_out,
            encoded_data_out => encoded_data_out,
            footer_out => footer_out,
            ready => ready,
            debug_shift_reg => debug_shift_reg,
            corrected_data_out => corrected_data_out,
            corrected_encoded_data_out => corrected_encoded_data_out,
            corrupted_bit_out => corrupted_bit_out,
            new_parity_out => new_parity_out,
            old_parity_out => old_parity_out
        );

    -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Initialize Inputs
        reset <= '1';
        En <= '0';
        wait for clk_period*3.5;

        reset <= '0';
        wait for clk_period*4;

        En <= '1';
        wait for clk_period*4;

        -- Send 28 bits (header, encoded_data, footer)
```

```
Rx <= '1'; wait for clk_period;  -- Start bit (or any bit)
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;

-- Encoded data (example)
Rx <= '0'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;

-- Footer
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '1'; wait for clk_period;

-- Wait for output to be ready
wait for clk_period*4;

-- Apply reset before the second test case
reset <= '1';
En <= '0';
wait for clk_period*4;

reset <= '0';
wait for clk_period*4;

En <= '1';
wait for clk_period*2;

-- Send another 28 bits
Rx <= '1'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '1'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '0'; wait for clk_period;
Rx <= '0'; wait for clk_period;
```

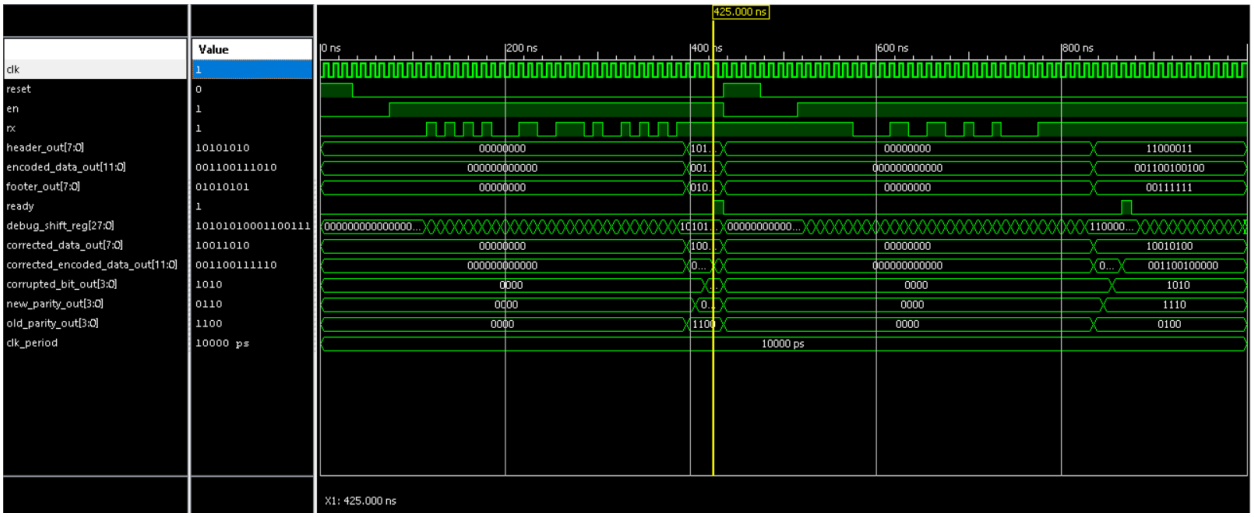
```
        -- Encoded data (example)
        Rx <= '1'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '1'; wait for clk_period;

        -- Footer
        Rx <= '0'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '0'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '1'; wait for clk_period;
        Rx <= '1'; wait for clk_period;

        -- Wait for the second output to be ready
        wait for clk_period*4;

        -- Stop simulation
        wait;
    end process;

end Behavioral;
```

Explanation

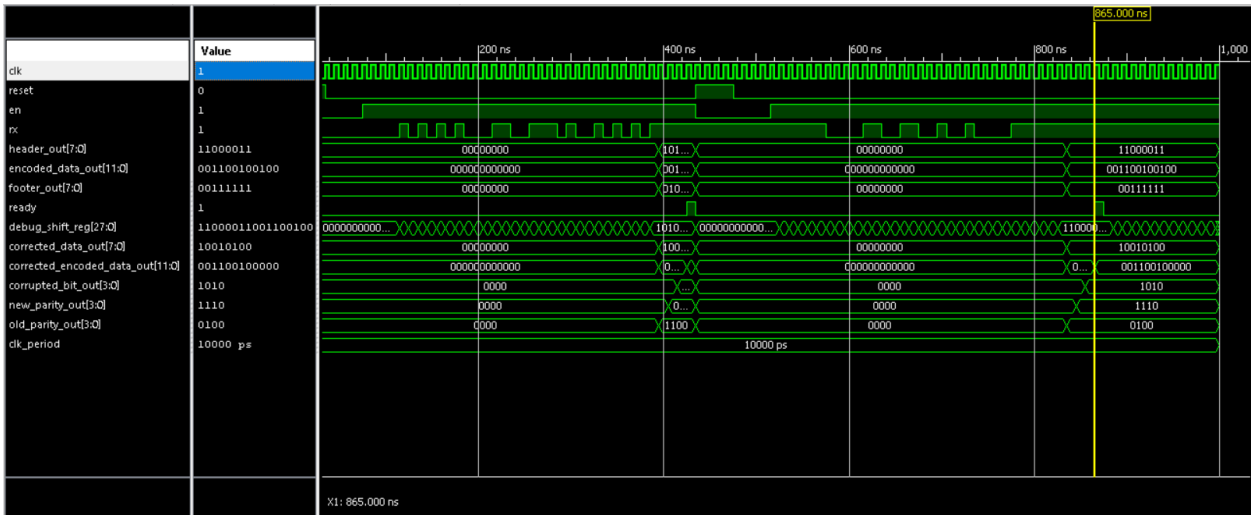
- **Initialization:** Initializes the inputs and generates a clock signal.

- **Stimulus Process**: Applies test cases to the Receiver module and verifies the output.

Results



The value of the output signals of the first test bench when ready is equal to 1.



The value of the output signals of the second test bench when ready is equal to 1.

Top Module

The Top Module integrates the Transmitter and Receiver to simulate the complete communication system.

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity TopModule is
    Port (
        clk                         : in  STD_LOGIC;
        reset                       : in  STD_LOGIC;
        En                          : in  STD_LOGIC;
        header_in                   : in  STD_LOGIC_VECTOR (7 downto 0);
        data_in                     : in  STD_LOGIC_VECTOR (7 downto 0);
        footer_in                   : in  STD_LOGIC_VECTOR (7 downto 0);
        noise_bit                   : in  integer range 0 to 11;
        ready                       : out STD_LOGIC;
        header_out                  : out STD_LOGIC_VECTOR (7 downto 0);
        encoded_data_out            : out STD_LOGIC_VECTOR (11 downto 0);
        footer_out                  : out STD_LOGIC_VECTOR (7 downto 0);
        corrected_data_out          : out STD_LOGIC_VECTOR (7 downto 0);
        corrected_encoded_data_out  : out STD_LOGIC_VECTOR (11 downto 0);
        corrupted_bit_out           : out STD_LOGIC_VECTOR (3 downto 0);
        new_parity_out              : out STD_LOGIC_VECTOR (3 downto 0);
        old_parity_out              : out STD_LOGIC_VECTOR (3 downto 0)
    );
end TopModule;

architecture Behavioral of TopModule is

    signal Tx                       : STD_LOGIC;
    signal Rx                       : STD_LOGIC;
    signal ready_tx                 : STD_LOGIC;

    component Transmitter
        Port (
            clk         : in  STD_LOGIC;
            reset       : in  STD_LOGIC;
            En          : in  STD_LOGIC;
            ready       : out STD_LOGIC;
            header_in   : in  STD_LOGIC_VECTOR (7 downto 0);
            data_in     : in  STD_LOGIC_VECTOR (7 downto 0);
            footer_in   : in  STD_LOGIC_VECTOR (7 downto 0);
            Tx          : out STD_LOGIC;
```

```
            noise_bit   : in  integer range 0 to 11
        );
    end component;

    component Receiver
        Port (
            clk                          : in  STD_LOGIC;
            reset                        : in  STD_LOGIC;
            En                           : in  STD_LOGIC;
            Rx                           : in  STD_LOGIC;
            header_out                   : out STD_LOGIC_VECTOR (7 downto 0);
            encoded_data_out             : out STD_LOGIC_VECTOR (11 downto 0);
            footer_out                   : out STD_LOGIC_VECTOR (7 downto 0);
            ready                        : out STD_LOGIC;
            debug_shift_reg              : out STD_LOGIC_VECTOR (27 downto 0);
            corrected_data_out           : out STD_LOGIC_VECTOR (7 downto 0);
            corrected_encoded_data_out : out STD_LOGIC_VECTOR (11 downto 0);
            corrupted_bit_out            : out STD_LOGIC_VECTOR (3 downto 0);
            new_parity_out               : out STD_LOGIC_VECTOR (3 downto 0);
            old_parity_out               : out STD_LOGIC_VECTOR (3 downto 0)
        );
    end component;

begin

    -- Instantiate the Transmitter
    U1: Transmitter
        Port map (
            clk        => clk,
            reset      => reset,
            En         => En,
            ready      => ready_tx,
            header_in  => header_in,
            data_in    => data_in,
            footer_in  => footer_in,
            Tx         => Tx,
            noise_bit  => noise_bit
        );

    -- Connect the transmitted signal to the receiver input
    process(clk, reset)
    begin
        if reset = '1' then
            Rx <= '1';
        elsif rising_edge(clk) then
            if ready_tx = '1' then
                Rx <= Tx;
            end if;
        end if;
    end process;

    -- Instantiate the Receiver
    U2: Receiver
        Port map (
            clk                          => clk,
            reset                        => reset,
            En                           => En,
```

```
            Rx                          => Rx,
            header_out                  => header_out,
            encoded_data_out            => encoded_data_out,
            footer_out                  => footer_out,
            ready                       => ready,
            debug_shift_reg             => open,  -- Not connected as per your
design
            corrected_data_out          => corrected_data_out,
            corrected_encoded_data_out => corrected_encoded_data_out,
            corrupted_bit_out           => corrupted_bit_out,
            new_parity_out              => new_parity_out,
            old_parity_out              => old_parity_out
        );

end Behavioral;
```

Explanation

- **Transmitter Instantiation**: Instantiates the Transmitter module and connects the inputs and outputs.

- **Receiver Instantiation**: Instantiates the Receiver module and connects the inputs and outputs.

- **Signal Connection**: Connects the transmitted signal from the Transmitter to the Receiver.

Top Module Testbench

The testbench verifies the functionality of the Top Module.

VHDL Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TopModule_tb is
end TopModule_tb;

architecture Behavioral of TopModule_tb is

    -- Component Declaration for the Unit Under Test (UUT)
    component TopModule
        Port (
            clk                         : in  STD_LOGIC;
            reset                       : in  STD_LOGIC;
            En                          : in  STD_LOGIC;
            header_in                   : in  STD_LOGIC_VECTOR (7 downto 0);
            data_in                     : in  STD_LOGIC_VECTOR (7 downto 0);
            footer_in                   : in  STD_LOGIC_VECTOR (7 downto 0);
            noise_bit                   : in  integer range 0 to 11;
            ready                       : out STD_LOGIC;
            header_out                  : out STD_LOGIC_VECTOR (7 downto 0);
            encoded_data_out            : out STD_LOGIC_VECTOR (11 downto 0);
            footer_out                  : out STD_LOGIC_VECTOR (7 downto 0);
            corrected_data_out          : out STD_LOGIC_VECTOR (7 downto 0);
            corrected_encoded_data_out  : out STD_LOGIC_VECTOR (11 downto 0);
            corrupted_bit_out           : out STD_LOGIC_VECTOR (3 downto 0);
            new_parity_out              : out STD_LOGIC_VECTOR (3 downto 0);
            old_parity_out              : out STD_LOGIC_VECTOR (3 downto 0)
        );
    end component;

    -- Inputs
    signal clk       : STD_LOGIC := '0';
    signal reset     : STD_LOGIC := '0';
    signal En        : STD_LOGIC := '0';
    signal header_in : STD_LOGIC_VECTOR (7 downto 0) := "10101010";
    signal data_in   : STD_LOGIC_VECTOR (7 downto 0) := "10011110";
    signal footer_in : STD_LOGIC_VECTOR (7 downto 0) := "01010101" ;
    signal noise_bit : integer range 0 to 11;

    -- Outputs
    signal ready                    : STD_LOGIC;
    signal header_out               : STD_LOGIC_VECTOR (7 downto 0);
```

```
    signal encoded_data_out           : STD_LOGIC_VECTOR (11 downto 0);
    signal footer_out                 : STD_LOGIC_VECTOR (7 downto 0);
    signal corrected_data_out         : STD_LOGIC_VECTOR (7 downto 0);
    signal corrected_encoded_data_out : STD_LOGIC_VECTOR (11 downto 0);
    signal corrupted_bit_out          : STD_LOGIC_VECTOR (3 downto 0);
    signal new_parity_out             : STD_LOGIC_VECTOR (3 downto 0);
    signal old_parity_out             : STD_LOGIC_VECTOR (3 downto 0);

    -- Clock period definition
    constant clk_period : time := 10 ns;

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: TopModule Port map (
            clk                          => clk,
            reset                        => reset,
            En                           => En,
            header_in                    => header_in,
            data_in                      => data_in,
            footer_in                    => footer_in,
            noise_bit                    => noise_bit,
            ready                        => ready,
            header_out                   => header_out,
            encoded_data_out             => encoded_data_out,
            footer_out                   => footer_out,
            corrected_data_out           => corrected_data_out,
            corrected_encoded_data_out   => corrected_encoded_data_out,
            corrupted_bit_out            => corrupted_bit_out,
            new_parity_out               => new_parity_out,
            old_parity_out               => old_parity_out
        );

    -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Initialize Inputs
        reset <= '1';
        wait for clk_period*1.5;

        reset <= '0';
        wait for clk_period*2;

        -- First Test Case
        noise_bit <= 2;
        wait for clk_period*2;
        header_in <= "10101010";
        data_in <= "10011110";
```

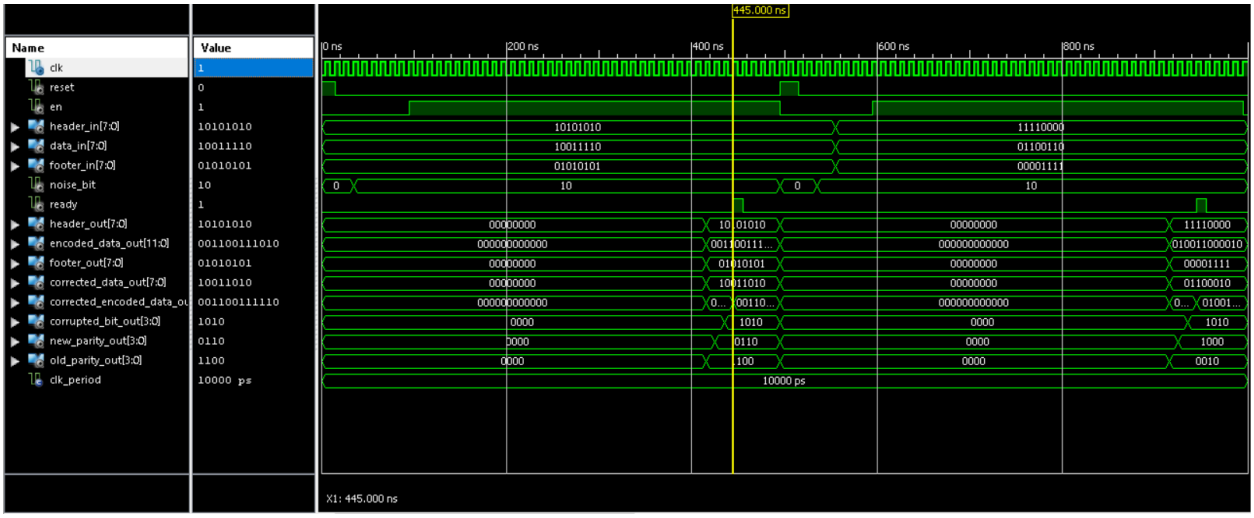
```
        footer_in <= "01010101";
        wait for clk_period*4;
        En <= '1';


        -- Wait for the first transmission to complete
        wait for clk_period*40;

        En <= '0';

        -- Apply reset before the second test case
        reset <= '1';
          noise_bit <= 0;
        wait for clk_period*2;

        reset <= '0';
        wait for clk_period*2;

        -- Second Test Case
        noise_bit <= 2;
        wait for clk_period*2;
        header_in <= "11110000";
        data_in <= "01100110";
        footer_in <= "00001111";
        wait for clk_period*4;
        En <= '1';


        -- Wait for the Second transmission to complete
        wait for clk_period*40;

        En <= '0';

        -- Stop simulation
        wait;
    end process;

end Behavioral;
```

Explanation

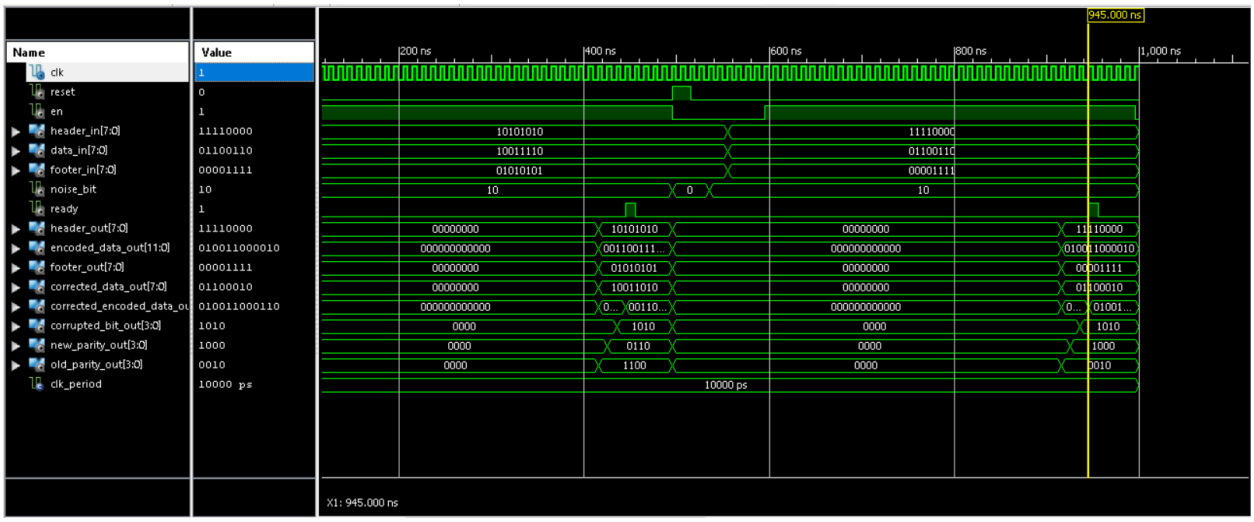
- **Initialization**: Initializes the inputs and generates a clock signal.

- **Stimulus Process**: Applies test cases to the Top Module and verifies the output.

Results



The value of the output signals of the first test bench when ready is equal to 1. As can be seen, the values of these signals are exactly equal to the results of the receiver section; Because the input signals are chosen exactly the same.



The value of the output signals of the second test bench when ready is equal to 1. As can be seen, the values of these signals are exactly equal to the results of the receiver section; Because the input signals are chosen exactly the same.

Conclusion

This report presents the design and implementation of a digital communication system using VHDL, incorporating Hamming code for error detection and correction. The system consists of a Transmitter, a Receiver, and a Top Module that integrates both. The functionality of each module was verified using testbenches, ensuring the system's capability to detect and correct single-bit errors effectively. This implementation demonstrates the robustness and reliability of Hamming code in digital communication systems.