ASIC/FPGA

Final project phase 2

Naghmeh Adibi Narges Erfan

Summary of what shall be done in phase 2:

In Phase 2 of our project, we will be advancing the digital communication system established in Phase 1 by incorporating a frame synchronization mechanism and a scrambling/descrambling scheme. This phase is crucial for enhancing the security and integrity of the transmitted signals, ensuring that the data remains coherent and unaltered during transmission. Objective: Our goal is to simulate a secure transmission channel, such as the DVB-S2 standard. By doing so, we aim to demonstrate the practical application of theoretical concepts in a real-world scenario. Approach: We will introduce an ASM along with a Scrambler to the existing setup. The ASM will be responsible for adding a predefined header to signal the start of each frame, thereby enabling the receiver to synchronize and properly decode the incoming data stream. Why This Matters: The addition of scrambling and frame synchronization is not just a technical requirement but a necessity in today's digital communication landscape. It protects against interference and signal degradation, which are common issues in any transmission medium. Moreover, it prepares our system to handle more complex modulation and coding schemes that might be introduced in future expansions of the project. By the end of Phase 2, we will have a fully functional transmitter-receiver pair capable of handling data transmission with added layers of security and reliability, setting the stage for further enhancements and optimizations.

Steps Involved:

1. Waveform Generation: - In the PS part, you generate the sinusoid, triangle, and sawtooth waveforms, later we will scramble it and add header. – similar data to these waveforms are stored in text files (sineWave.txt, sawtoothWave.txt, etc.) in matlab code.
2. Scrambling Process: - You read the waveform data from the files and perform a scrambling operation. - The scrambling is done using a specific algorithm, where you apply a linear feedback shift register (LFSR). - The scrambled data is then written to new files (Sine_Scrambled.txt, Sawtooth_Scrambled.txt, etc.) in matlab code. we should implement it in PS.
3. Frame Synchronization: - A synchronization word (sync_word) is defined and added to the beginning of each frame of the scrambled data. - This helps the receiver side to identify the start of each frame and synchronize the descrambler. It should be done in PS and matlab transmitter.
4. FFT Computation and Display: - The PL part takes the FFT of the signal using an IP core. - The VDMA is used to send the data to a monitor, where both the waveform and its FFT are displayed. We have already handled this in phase 1. Now, the data received by dma is scrambled. We should add IP cores to remove header and descramble data and them pass it to ff tip core.
5. Receiver Design: - On the receiver side, the same scrambling algorithm is used to descramble the data and recover the original waveform. This system simulates a real-world digital communication system where data is scrambled for transmission and then descrambled at the receiver end, ensuring data integrity and security. The use of a synchronization word is a common practice in communication systems to align the transmitter and receiver.

Verilog implementation and simulation of scrambler and descrambler:

We use LFSR with DVB-S2 standard to implement scrambling algorithm. This algorithm is described in the standard's file. This is the flow of data:

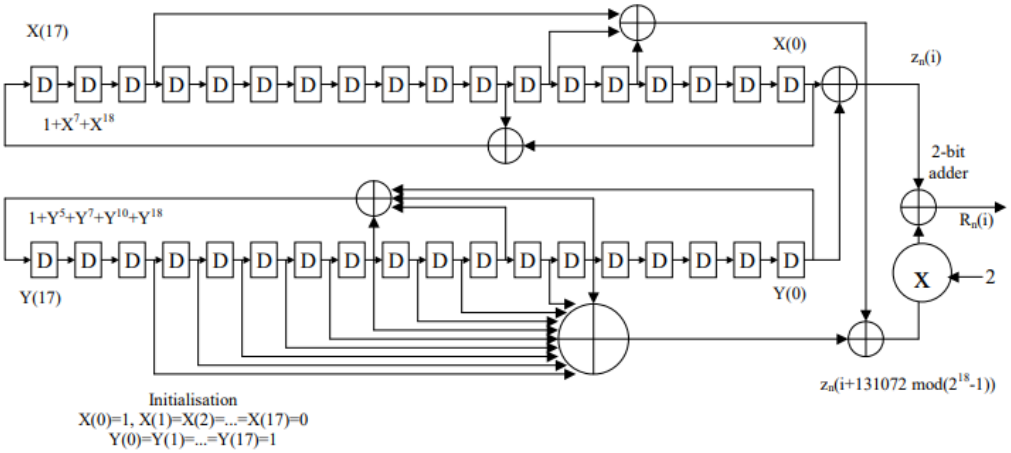


Figure 15: Configuration of PL scrambling code generator for n = 0

1. **Initialization**: The LFSR is initialized with a non-zero seed value to start the scrambling process. In DVB-S2, a specific initial state is often used for the registers which is explained in standard file.

Initialisation
X(0)=1, X(1)=X(2)=...=X(17)=0
Y(0)=Y(1)=...=Y(17)=1

2. **Feedback Polynomial**: The LFSR uses a feedback polynomial that defines which bits of the register are used to create the feedback bit. This polynomial is crucial as it determines the properties of the output sequence, such as its period and randomness.

3. **Shifting and Feedback**: On each clock cycle, the register shifts its bits to the right, and the leftmost bit (feedback bit) is calculated based on the feedback polynomial. In our code, the feedback is calculated as `x_feed` and `y_feed`, which are then used to update the LFSR states `X` and `Y`.

4. **Scrambling**: The output of the LFSR is then combined with the input data to scramble it. The combination is typically a bitwise exclusive OR (XOR) operation. In the DVB-S2 standard, the scrambling is done by multiplying the input samples by a complex randomization sequence.

5. **Descrambling**: The receiver, which knows the feedback polynomial and the initial state of the LFSR, can perform the same operations to descramble the data.

$$C_I(i) + jC_Q(i) = \exp(j\, R_n\, (i)\, \pi/2)$$

R_n	$\exp(j\, R_n\, \pi/2)$	$I_{scrambled}$	$Q_{scrambled}$
0	1	I	Q
1	j	-Q	I
2	-1	-I	-Q
3	-j	Q	-I

Red signs are for descrambling. Black for scrambling.

In the code, the LFSR is implemented with two 18-bit registers (`X` and `Y`), and the feedback is calculated using XOR operations on specific bits of these registers. The scrambled output is generated based on the result of XOR operations between certain bits of `X` and `Y`, which are then used to determine how the input data (`inp`) is manipulated to produce the scrambled output (`outp`).

This method ensures that the transmitted data has good statistical properties, which is essential for reliable communication over satellite links. The DVB-S2 standard uses this scrambling technique to randomize the data before modulation and transmission, reducing the likelihood of interference and improving error correction performance.

Scrambler Verilog code:

```
`timescale 1ns / 1ps
module data_scrambler(
    input wire clock,
    input wire enable,
    input wire [15:0] data_in,
    output reg [15:0] data_out
);

    reg [17:0] LFSR_X = 18'h00001;
    reg [17:0] LFSR_Y = 18'h3ffff;
    wire feedback_X, feedback_Y;
    wire xor_X, xor_Y;
    wire zeta, zeta_shifted;
    wire [1:0] result;

    assign feedback_X = LFSR_X[0] ^ LFSR_X[7];
    assign feedback_Y = LFSR_Y[10] ^ LFSR_Y[7] ^ LFSR_Y[5] ^ LFSR_Y[0];

    assign xor_X = LFSR_X[4] ^ LFSR_X[6] ^ LFSR_X[15];
    assign xor_Y = LFSR_Y[5] ^ LFSR_Y[6] ^ LFSR_Y[8] ^ LFSR_Y[9] ^ LFSR_Y[10] ^ LFSR_Y[11] ^
LFSR_Y[12] ^ LFSR_Y[13] ^ LFSR_Y[14] ^ LFSR_Y[15];

    assign zeta = LFSR_X[0] ^ LFSR_Y[0];
```

```
    assign zeta_shifted = xor_X ^ xor_Y;

    assign result = zeta + (zeta_shifted << 1);

    always @(posedge clock) begin
        data_out <= 0;
        if(enable) begin
            LFSR_X <= {feedback_X, LFSR_X[17:1]};
            LFSR_Y <= {feedback_Y, LFSR_Y[17:1]};
            data_out <= (result == 0) ? data_in :
                        (result == 1) ? {data_in[7:0], -data_in[15:8]} :
                        (result == 2) ? -data_in :
                        (result == 3) ? {-data_in[7:0], data_in[15:8]} :
                        data_in;
        end
    end

endmodule
```

Descrambler Verilog code:

```
`timescale 1ns / 1ps
module signal_descrambler(
    input wire clock_signal,
    input wire enable_signal,
    input wire [15:0] input_signal,
    output reg [15:0] output_signal
);

    reg [17:0] Shift_Register_A = 18'h00001;
    reg [17:0] Shift_Register_B = 18'h3ffff;
    wire feedback_A, feedback_B;
    wire xor_output_A, xor_output_B;
    wire combined_signal, combined_signal_shifted;
    wire [1:0] operation_code;

    assign feedback_A = Shift_Register_A[0] ^ Shift_Register_A[7];
    assign feedback_B = Shift_Register_B[10] ^ Shift_Register_B[7] ^ Shift_Register_B[5] ^
Shift_Register_B[0];

    assign xor_output_A = Shift_Register_A[4] ^ Shift_Register_A[6] ^ Shift_Register_A[15];
    assign xor_output_B = Shift_Register_B[5] ^ Shift_Register_B[6] ^ Shift_Register_B[8] ^
Shift_Register_B[9] ^ Shift_Register_B[10] ^ Shift_Register_B[11] ^ Shift_Register_B[12] ^
Shift_Register_B[13] ^ Shift_Register_B[14] ^ Shift_Register_B[15];

    assign combined_signal = Shift_Register_A[0] ^ Shift_Register_B[0];
```

```
    assign combined_signal_shifted = xor_output_A ^ xor_output_B;

    assign operation_code = combined_signal + (combined_signal_shifted << 1);

    always @(posedge clock_signal) begin
        output_signal <= 0;
        if(enable_signal) begin
            Shift_Register_A <= {feedback_A, Shift_Register_A[17:1]};
            Shift_Register_B <= {feedback_B, Shift_Register_B[17:1]};
            output_signal <= (operation_code == 0) ? input_signal :
                             (operation_code == 1) ? {-input_signal[7:0], input_signal[15:8]}
:
                             (operation_code == 2) ? -input_signal :
                             (operation_code == 3) ? {input_signal[7:0], -input_signal[15:8]}
:
                             input_signal;
        end
    end

endmodule
```

Testing module to implement both scrambler and descrambler:

```
module Test(
    input clk,
    input en_sc, en_de,
    input [15:0] inp,
    output [15:0] outp_midi,
    output [15:0] outp
);

    assign outp_midi = outp_mid;

    wire [15:0] outp_mid;

    data_scrambler scr (
        .clock(clk),
        .enable(en_sc),
        .data_in(inp),
        .data_out(outp_mid)
    );
    signal_descrambler descr (
        .clock_signal(clk),
        .enable_signal(en_de),
        .input_signal(outp_mid),
```

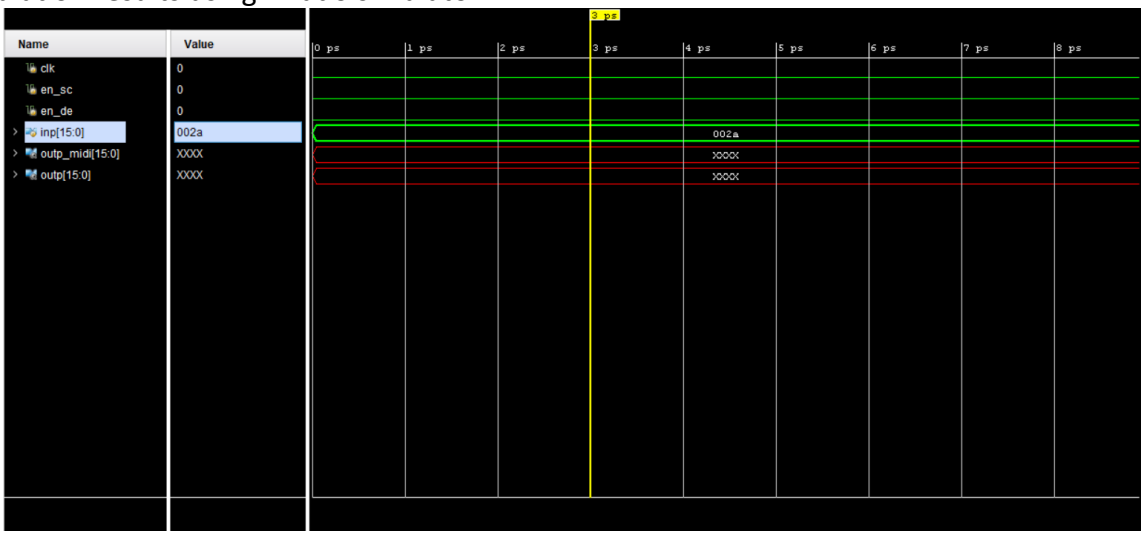
```
        .output_signal(outp)
    );

endmodule
```

Testbench:

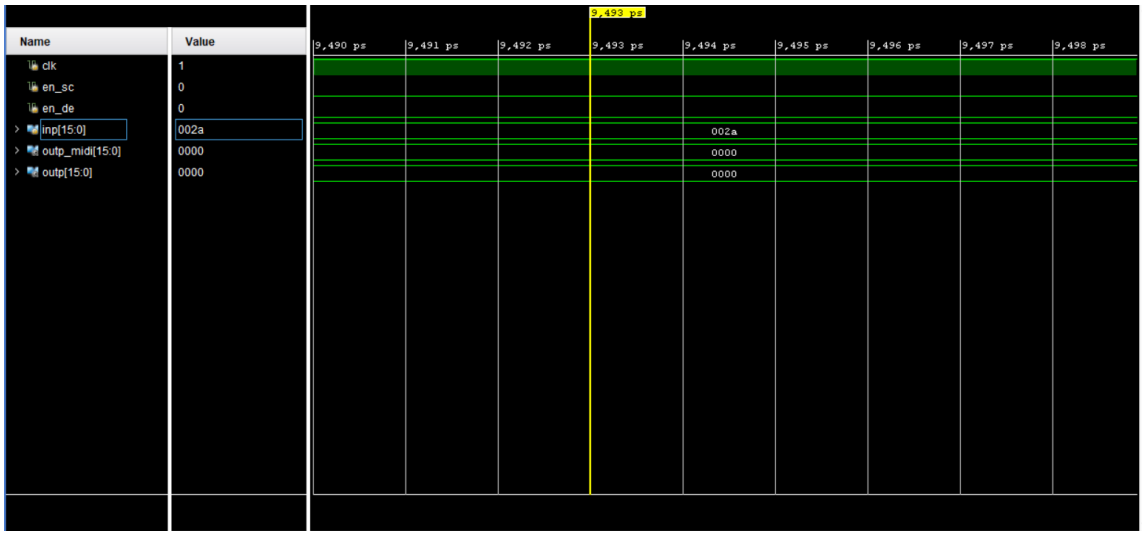
```
module Test_tb;

    // Inputs
    reg clk;
    reg en_sc;
    reg en_de;
    reg [15:0] inp;

    // Outputs
    wire [15:0] outp_midi;
    wire [15:0] outp;

    // Instantiate the Unit Under Test (UUT)
    Test uut (
        .clk(clk),
        .en_sc(en_sc),
        .en_de(en_de),
        .inp(inp),
        .outp_midi(outp_midi),
        .outp(outp)
    );

    always #5 clk = ~clk;

    initial begin
        // Initialize Inputs
        clk = 0;
        en_sc = 0;
        en_de = 0;
        inp = 0;

        inp = 16'h002A; // 42 in hexadecimal

        #10;
        en_de = 1;
        en_sc = 1;
        inp = 16'h0055; // 85 in hexadecimal

        #10;
```

```
        inp = 16'h000C; // 12 in hexadecimal

        #10;
        inp = 16'h0043; // 67 in hexadecimal

        #10;
        inp = 16'h0091; // 145 in hexadecimal

        #10;
        en_sc = 0;

        #10;
        en_de = 0;

        #100;
    end

endmodule
```

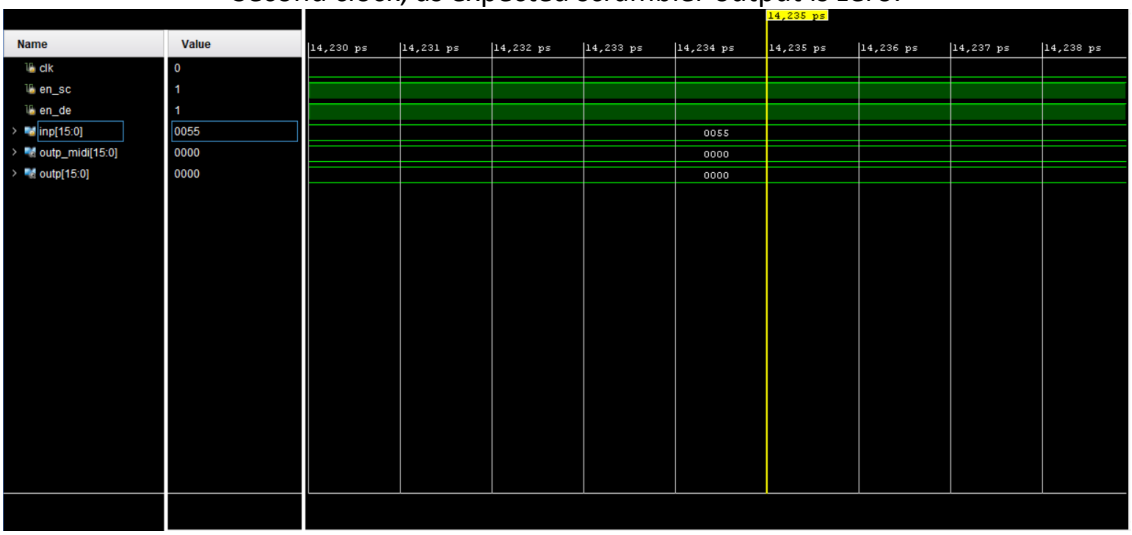
Simulation results using vivado simulator:



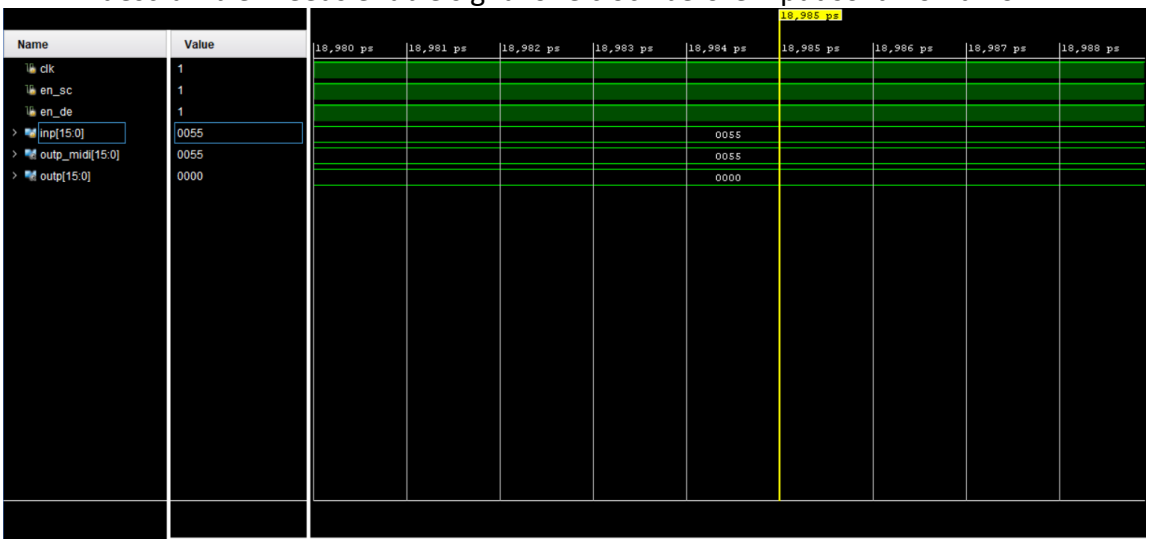
In the first clock, enable signals are bot applied so we expect no response to the input.



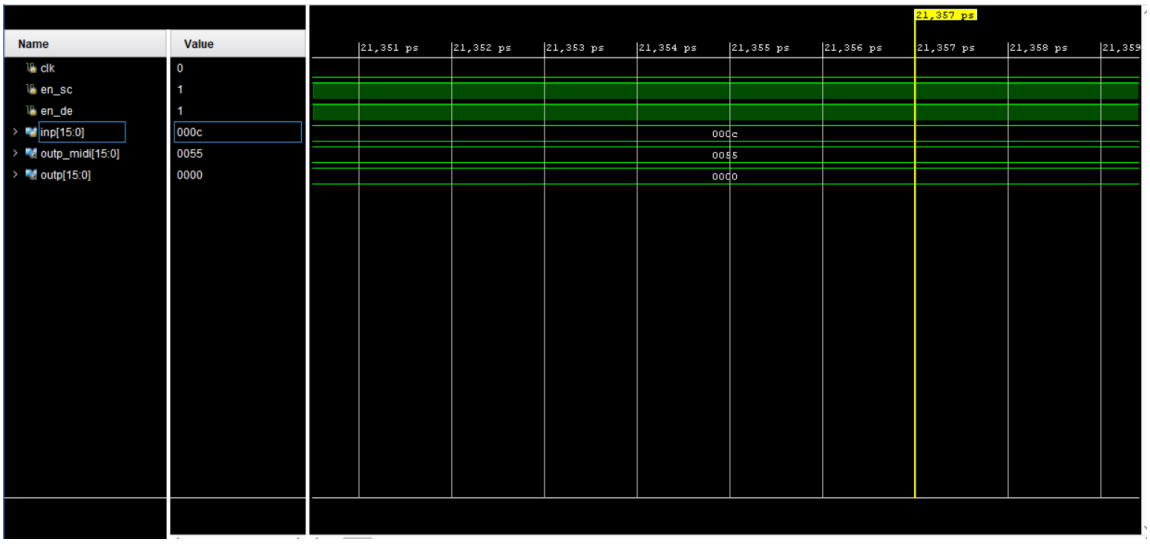
Second clock, as expected scrambler output is zero.



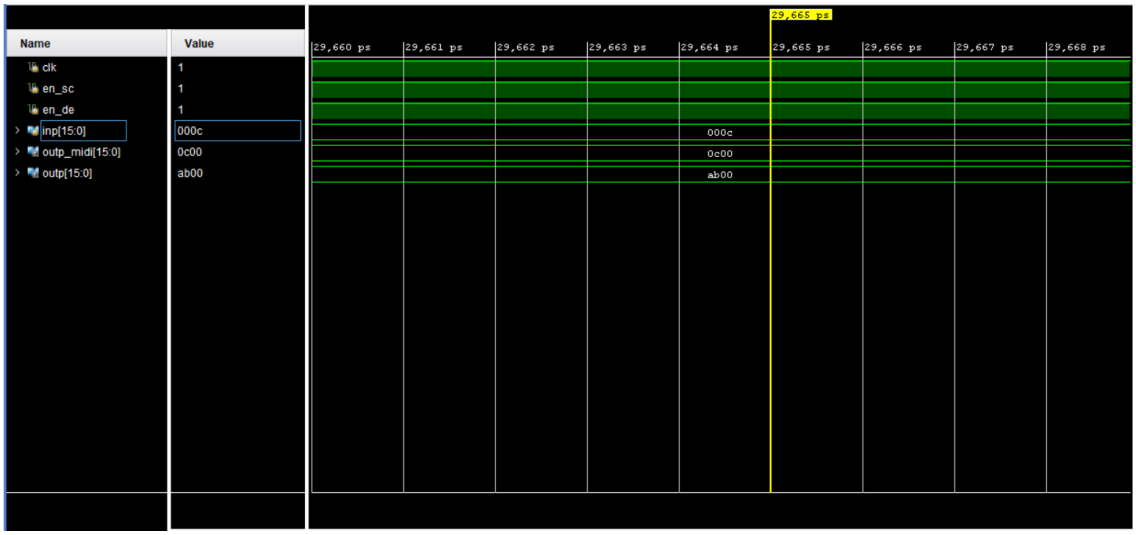
third clock, new input and enable signals are applied. Output will be scrambled wrong. descrambler needs enable signal one clock before input so it won't work.



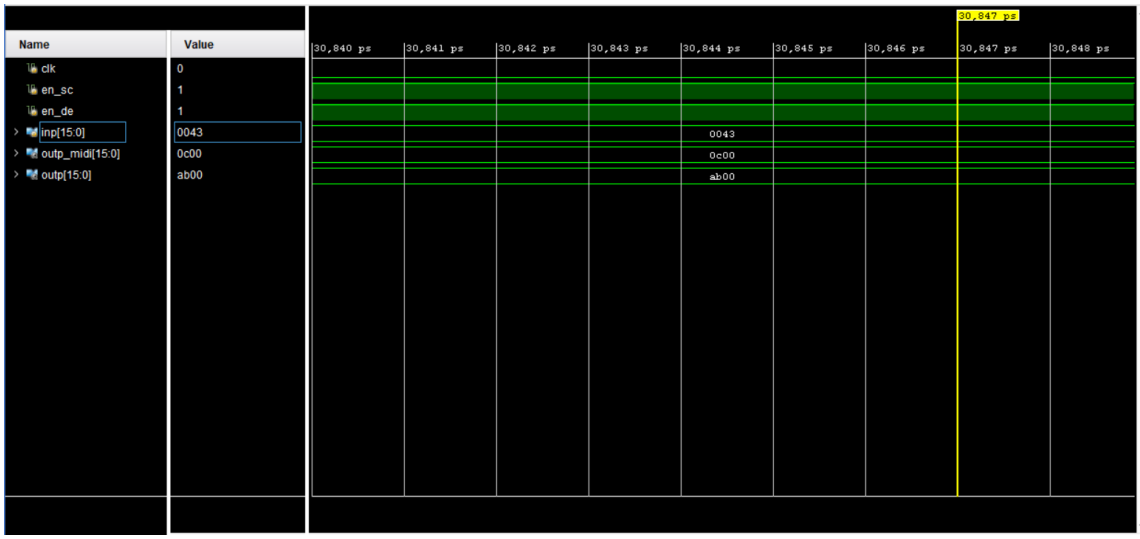
4th clock



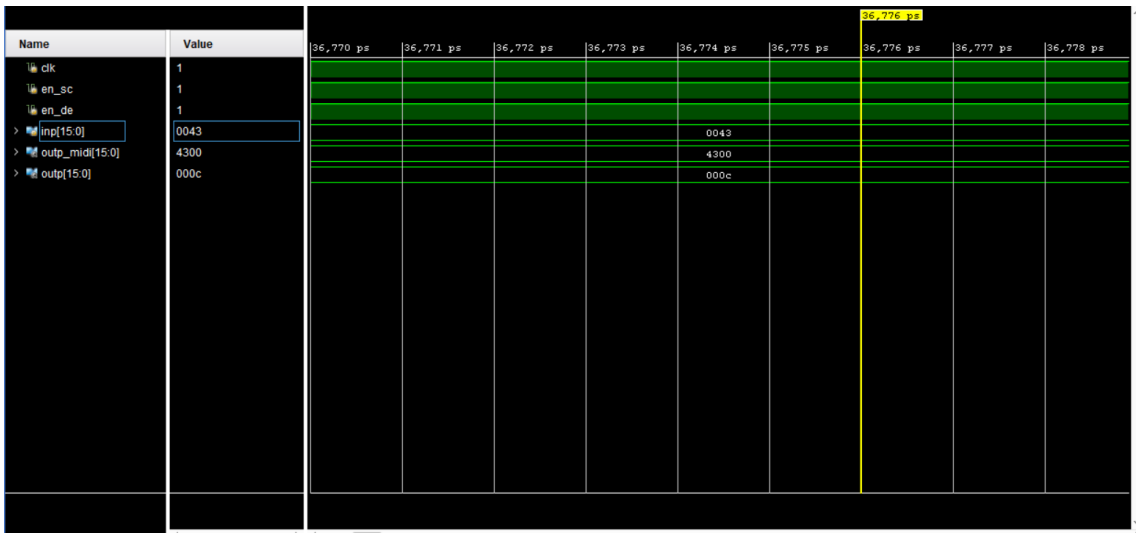
5th clock



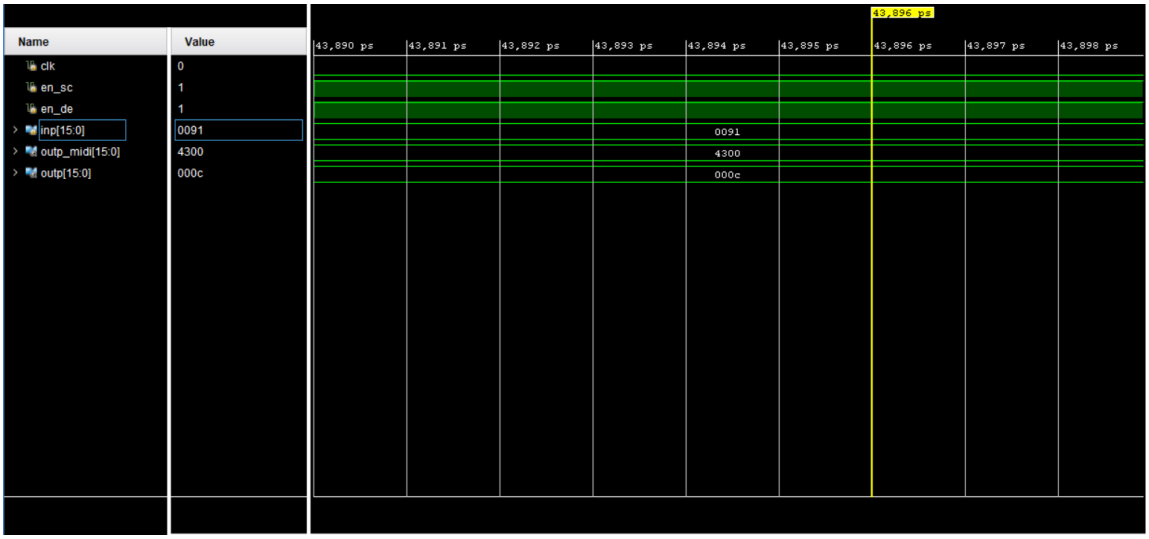
6th clock



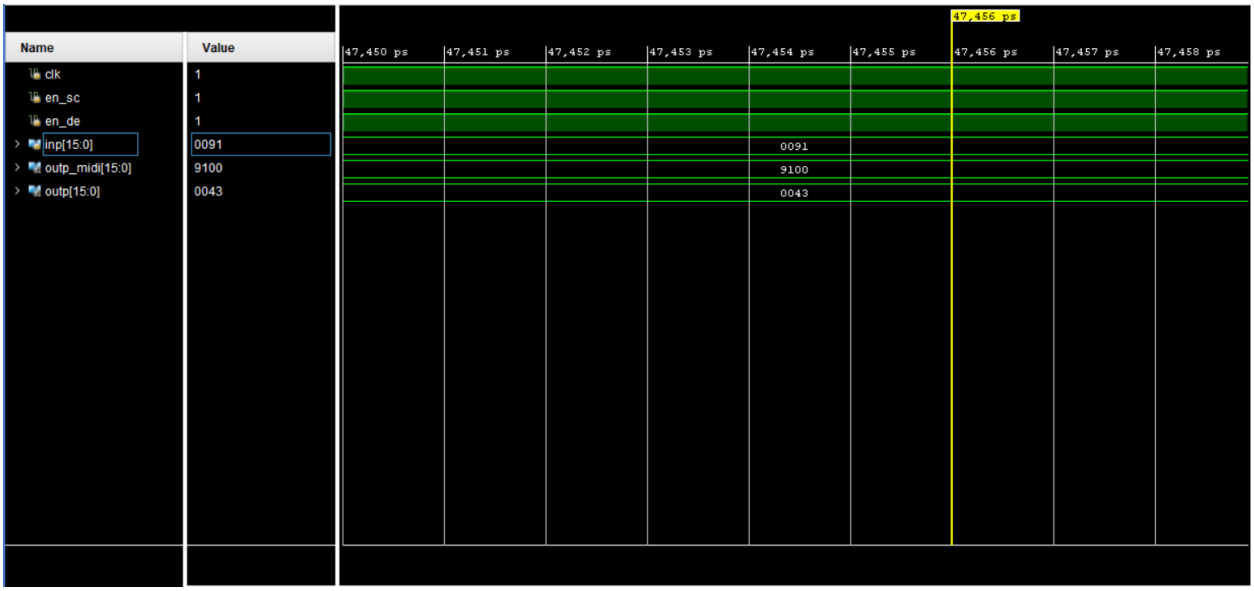
7th clock



8th clock



9th clock



10th clock

11th clock

We see that our modules work perfectly! We should care to set enable signal before data in inserted.

Verilog implementation and simulation of header creator and detector:

In this part, header is added to data and removed.
Verilog code to add header:

```
`timescale 1ns / 1ps
```

```
// Module for managing a FIFO buffer
module HeaderBuffer(
    input wire clk,
    input wire enable,
    input wire [15:0] buffer_input,
    output reg [15:0] buffer_output
);

    // FIFO memory declaration
    reg [15:0] fifo_memory [2:0];
    reg buffer_valid = 0;
    reg [2:0] buffer_count = 0;

    // Initialize FIFO with default values
    initial begin
        fifo_memory[0] <= 16'hffff;
        fifo_memory[1] <= 16'hffff;
        fifo_memory[2] <= 16'hffff;
    end

    // FIFO buffer logic
    always @ (posedge clk) begin
        if (enable) begin
            buffer_output <= fifo_memory[2];
            fifo_memory[2] <= fifo_memory[1];
            fifo_memory[1] <= fifo_memory[0];
            fifo_memory[0] <= buffer_input;
            buffer_valid <= 1;
            buffer_count <= 0;
        end else if (buffer_valid) begin
            if (buffer_count < 3) begin
                buffer_count <= buffer_count + 1;
                buffer_output <= fifo_memory[2];
                fifo_memory[2] <= fifo_memory[1];
                fifo_memory[1] <= fifo_memory[0];
            end else begin
                buffer_valid <= 0;
                fifo_memory[0] <= 16'hffff;
                fifo_memory[1] <= 16'hffff;
                fifo_memory[2] <= 16'hffff;
            end
        end
    end

endmodule
```

Verilog code to detect data and header:

```
`timescale 1ns / 1ps

// Module for detecting a specific signal pattern
module SignalDetector(
    input wire clk,
    input wire [15:0] signal_input,
    output wire [15:0] signal_output
);

    // Conditional assignment for output based on valid signal
    assign signal_output = (valid_signal) ? signal_input : 16'd0;

    // State machine variables
    reg [1:0] current_state = 0;
    reg [9:0] signal_count = 0;
    reg valid_signal = 0;

    // State machine for pattern detection
    always @ (posedge clk) begin
        case (current_state)
            0: if (signal_input == 16'hffff) current_state <= 1;
            1: current_state <= (signal_input == 16'hffff) ? 2 : 0;
            2: if (signal_input == 16'hffff) begin
                    current_state <= 3;
                    signal_count <= 0;
                    valid_signal <= 1;
                end else current_state <= 0;
            3: if (signal_count < 9) begin
                    signal_count <= signal_count + 1;
                end else begin
                    current_state <= 0;
                    valid_signal <= 0;
                end
        endcase
    end

endmodule
```

Testbench module to implement above modules for testing:

```
`timescale 1ns / 1ps
// Top-level module that combines the Header and Detector
module TestBench(
    input wire clk,
    input wire enable,
```

```
    input wire [15:0] test_input,
    output wire [15:0] midi_output,
    output wire [15:0] final_output
);

    // Wire to connect the two modules
    wire [15:0] intermediate_output;

    // Instantiation of HeaderBuffer and SignalDetector
    HeaderBuffer header (
        .clk(clk),
        .enable(enable),
        .buffer_input(test_input),
        .buffer_output(intermediate_output)
    );

    SignalDetector detector (
        .clk(clk),
        .signal_input(intermediate_output),
        .signal_output(final_output)
    );

    // Assign intermediate output to midi_output
    assign midi_output = intermediate_output;

endmodule
```

Testbench:

```
`timescale 1ns / 1ps
// Testbench module to simulate the TestBench behavior
module TestBenchSimulation;

    // Inputs for the simulation
    reg sim_clk;
    reg sim_enable;
    reg [15:0] sim_input;

    // Outputs from the TestBench
    wire [15:0] sim_midi_output;
    wire [15:0] sim_final_output;

    // Instantiate the TestBench
    TestBench uut (
        .clk(sim_clk),
```

```
    .enable(sim_enable),
    .test_input(sim_input),
    .midi_output(sim_midi_output),
    .final_output(sim_final_output)
);

// Clock generation for simulation
always #5 sim_clk = ~sim_clk;

// Simulation sequence
initial begin
    // Initialize simulation inputs
    sim_clk = 0;
    sim_enable = 0;
    sim_input = 0;

    // Wait for global reset
    #100;

    // Stimulus sequence for the TestBench
    sim_enable = 1;
    sim_input = 100;

    // Sequential input changes with time delay
    #10 sim_input = 101;
    #10 sim_input = 102;
    #10 sim_input = 103;
    #10 sim_input = 104;
    #10 sim_input = 105;
    #10 sim_input = 106;
    #10 sim_input = 107;
    #10 sim_input = 108;
    #10 sim_input = 109;

    // Disable and re-enable sequence
    #10 sim_enable = 0;
    #40 sim_enable = 1;
    sim_input = 100;

    // Repeat input changes
    #10 sim_input = 101;
    #10 sim_input = 102;
    #10 sim_input = 103;
    #10 sim_input = 104;
    #10 sim_input = 105;
```

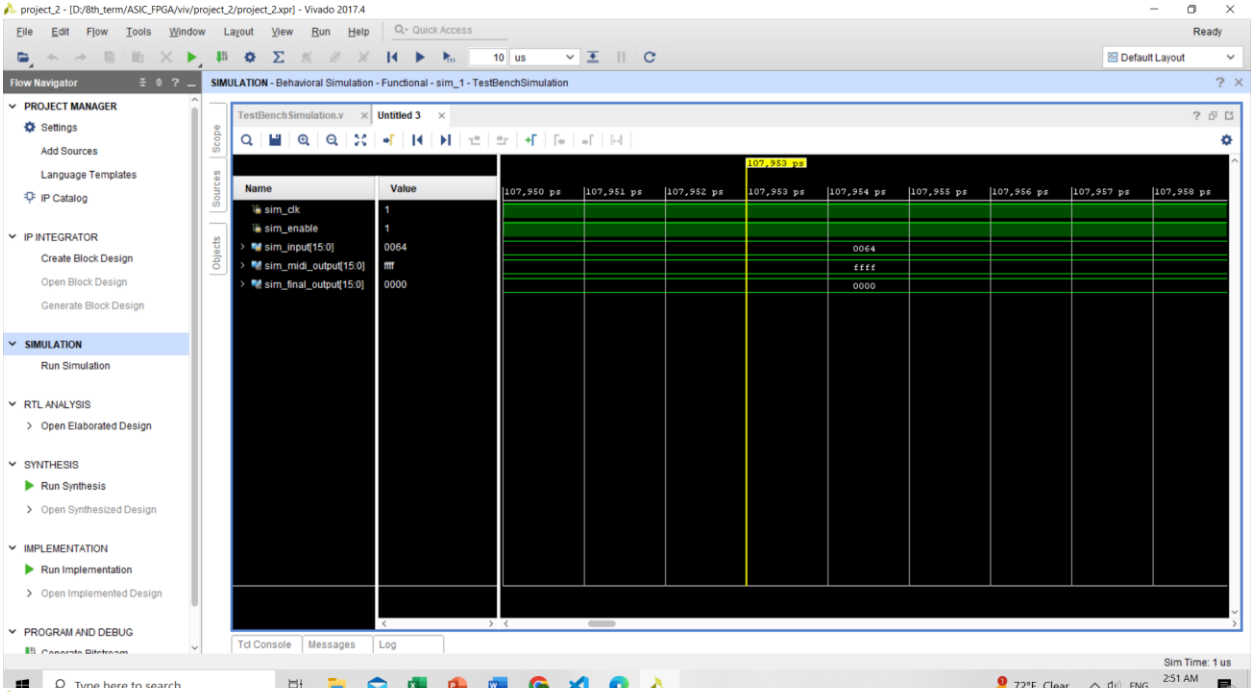
```
        #10 sim_input = 106;
        #10 sim_input = 107;
        #10 sim_input = 108;
        #10 sim_input = 109;

        // Final disable
        #10 sim_enable = 0;
    end

endmodule
```

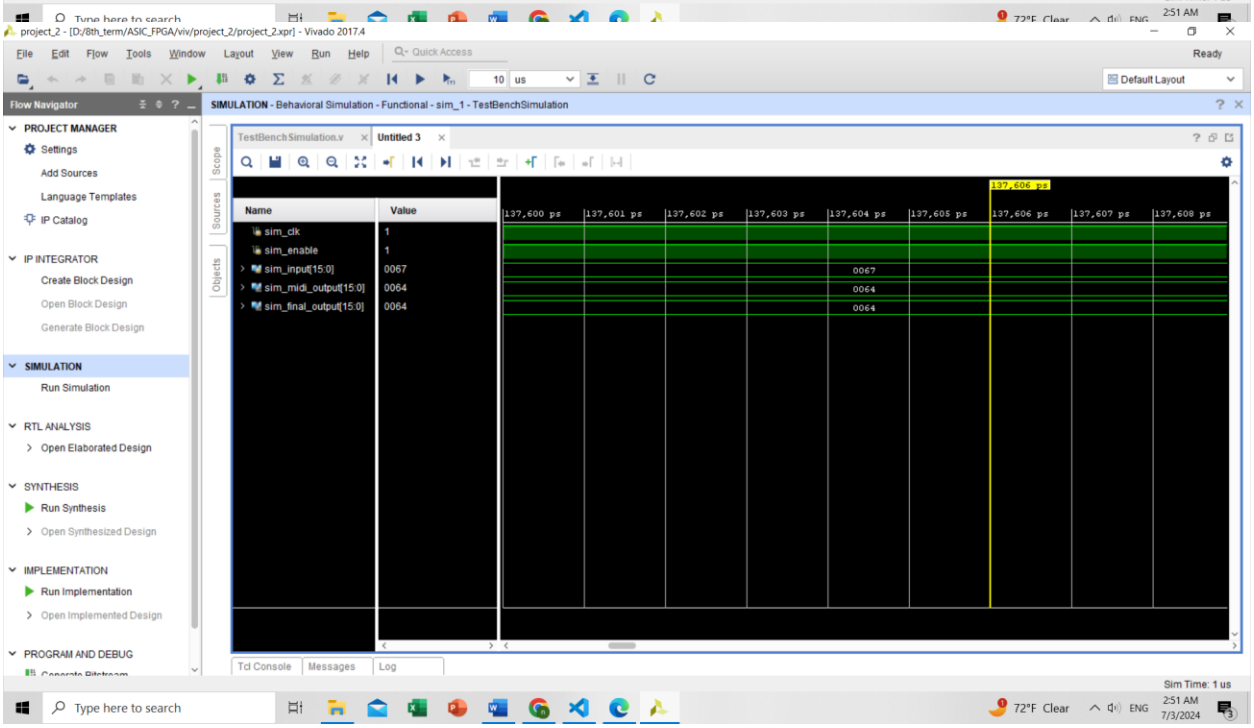
This part of the project consists of a Verilog simulation for a digital system that processes 16-bit data inputs. It's designed to add and remove headers from the data, which is a common operation in data packet processing for communication systems.

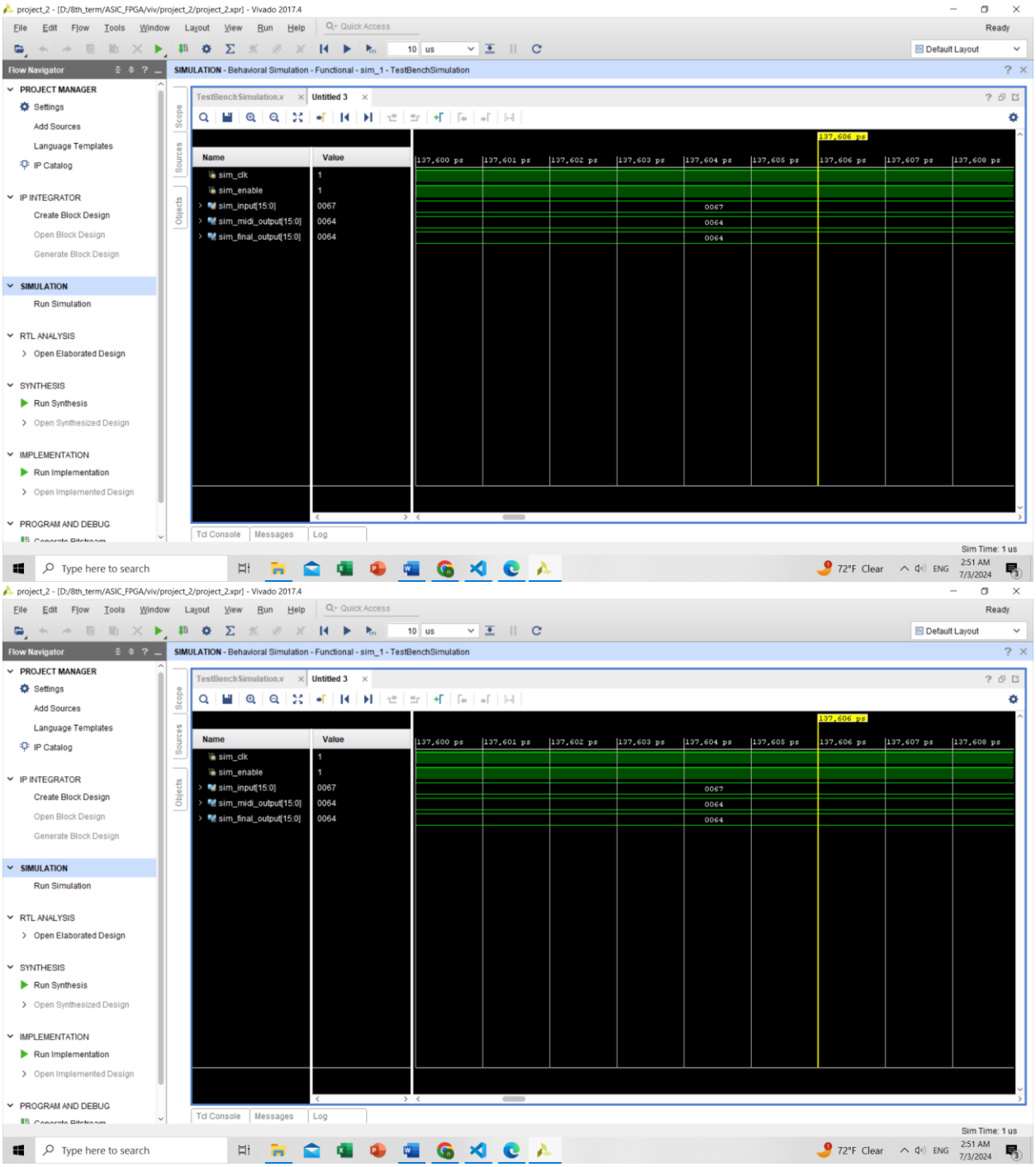
1. **SignalDetector Module**: This module acts as a pattern detector. It looks for a specific 16-bit pattern (`16'hffff`) in the input data stream. When this pattern is detected consecutively, the module activates a valid signal (`valid_signal`) and allows the input data to pass through to the output for a certain number of clock cycles. If the pattern is not detected, the output is set to zero.
2. **HeaderBuffer Module**: This module manages a First-In-First-Out (FIFO) buffer with three 16-bit memory slots. When enabled, it shifts the data through the FIFO, effectively adding a header to the data. The header is predefined as `16'hffff`. When the enable signal is turned off, the module continues to shift out the remaining data in the FIFO and then resets, preparing to add headers to new incoming data.
3. **TestBench Module**: This is the top-level module that combines the HeaderBuffer and SignalDetector modules. It takes an input signal and first passes it through the HeaderBuffer, which adds the header. The output of the HeaderBuffer is then fed into the SignalDetector, which checks for the specific pattern and allows the data to pass through if the pattern is present.
4. **TestBenchSimulation Module**: This is a testbench used to simulate the behavior of the TestBench module. It generates a clock signal and provides input data to the TestBench. The simulation toggles the enable signal and changes the input data at specific intervals to test the functionality of the entire system.

The project simulates a system that can add headers to incoming data when enabled and detect specific data patterns to control the output. The simulation ensures that the logic works as intended before the design is implemented in hardware.

Simulation results using Vivado:

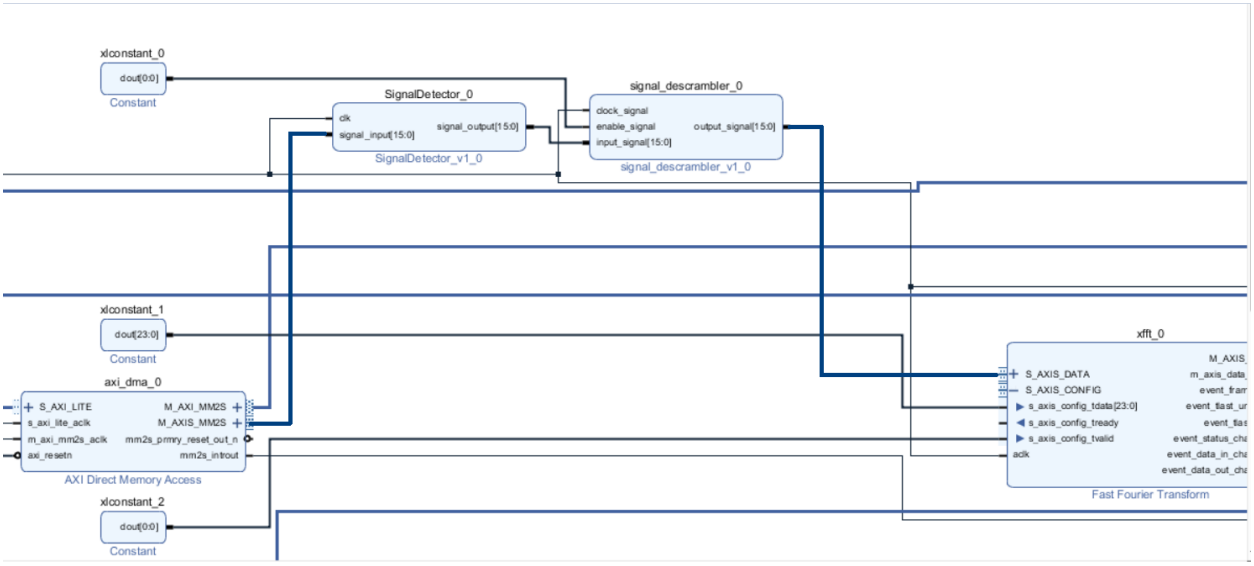

project_2 - [D:/8th_term/ASIC_FPGA/viv/project_2/project_2.xpr] - Vivado 2017.4
File Edit Flow Tools Window Layout View Run Help
Quick Access
Ready
Default Layout
Flow Navigator
SIMULATION - Behavioral Simulation - Functional - sim_1 - TestBenchSimulation
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
TestBenchSimulation.v
Untitled 3
Name
Value
sim_clk
sim_enable
sim_input[15:0]
sim_midi_output[15:0]
sim_final_output[15:0]
0064
ffff
0000
107,953 ps
Tcl Console
Messages
Log
Sim Time: 1 us
Type here to search
72°F Clear
ENG
2:51 AM


project_2 - [D:/8th_term/ASIC_FPGA/viv/project_2/project_2.xpr] - Vivado 2017.4
File Edit Flow Tools Window Layout View Run Help
Quick Access
Ready
Default Layout
Flow Navigator
SIMULATION - Behavioral Simulation - Functional - sim_1 - TestBenchSimulation
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
TestBenchSimulation.v
Untitled 3
Name
Value
sim_clk
sim_enable
sim_input[15:0]
sim_midi_output[15:0]
sim_final_output[15:0]
0067
0064
0064
137,606 ps
Tcl Console
Messages
Log
Sim Time: 1 us
Type here to search
72°F Clear
ENG
2:51 AM
7/3/2024

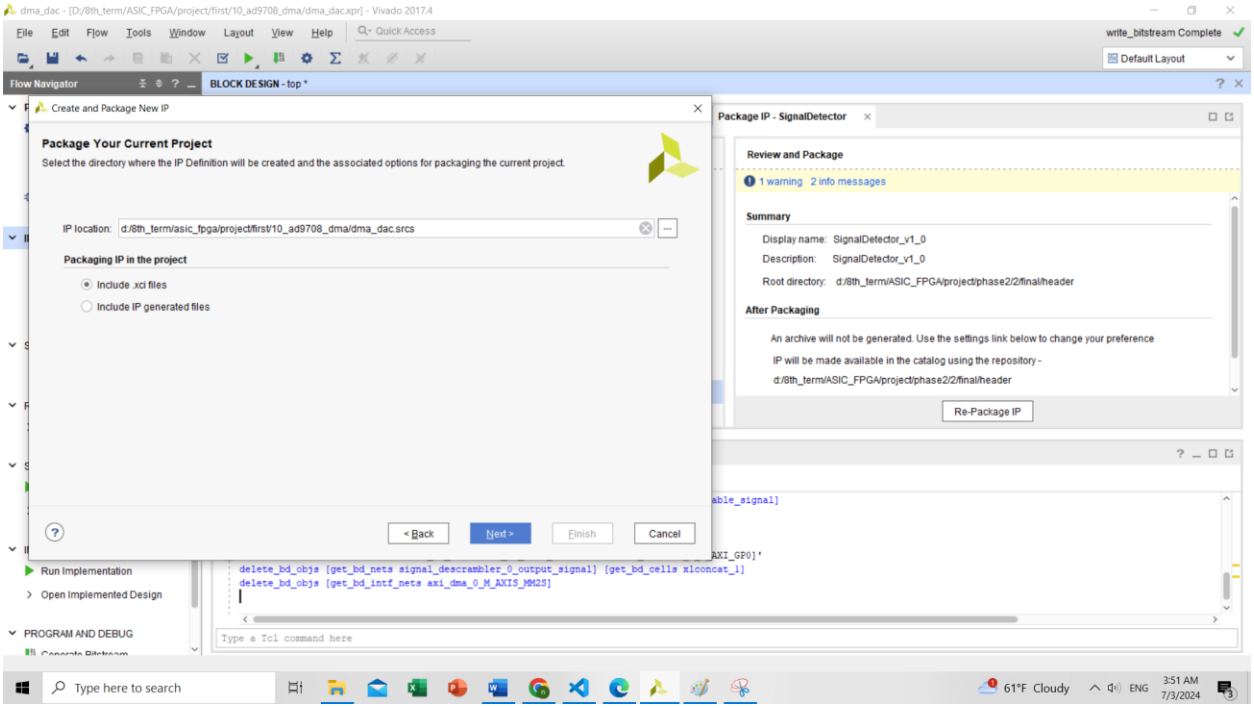


Results are as expected.

IP Core implementation:



The data is scrambled and header is added to it in PS part (wave.c file) we have already simulated this step using testbench. In PL, header must be detected (SignalDetector IP core) and then descrambled (signal_descrambler IP core) before reaching fft IP core. The Verilog codes for header detector and descrambler tested in previous parts are packed in IP cores with compatible setting and added to project as shown in the diagram above.



MATLAB:

In this part, we shall test the design by MATLAB. All of the scrambler, descrambler, header-detector, header-adder are implemented and results are saved to files. We have initial data files from previous phase of the project. Those files are scrambled and header is added then header is detected and descrambled. If the final file is same as initial file, the approach is accurate. I have sent all files for accuracy to be checked.
I avoid mentioning codes and files here. They are in matlab folder.

Scrambler Explanation:
The code performs the following tasks:
- Clears the MATLAB environment and command window.
- Defines file paths for input and output.
- Initializes arrays to store wave data.
- Reads and processes each wave type from the input files.
- Initializes two Linear Feedback Shift Registers (LFSRs) for scrambling.
- Scrambles the wave data based on the LFSR outputs with same algorithm explained in Verilog scrambler part of report.
- Writes the scrambled data to output files.

The scrambling process involves bitwise operations and LFSR updates to modify the original wave data. The `computeTwosComplement` function is used to find the two's complement of binary strings during the scrambling process. The modified data is then written to new files with updated names. The variable names and comments have been changed to enhance readability and maintain the code's original functionality.

Descrambler Explanation:
The code performs the descrambling of wave data stored in text files. It reads the scrambled binary data, separates it into real and imaginary components, and then applies a descrambling algorithm based on Linear Feedback Shift Registers (LFSRs). The descrambling rules are determined by the LFSR outputs, and the two's complement of the binary strings is calculated when required. The descrambled data is then written to new output files. The variable names and comments have been updated for clarity, without altering the code's behavior.

Explanation of the header Code:
- The script starts by clearing the MATLAB environment.
- It defines arrays of file paths for the scrambled input files and the output files where headers will be added.
- A loop processes each file by calling the `decodeAndAddHeader` function.
- The `decodeAndAddHeader` function reads the scrambled data from the input file, converts it into a binary matrix, and splits it into two halves.
- It then adds three rows of binary ones to the top of each half as a header.

- The two halves are recombined, and the complete data matrix is written to the output file, adding a header to the original scrambled data.
- The function ensures that the data is written in 16-bit segments, separated by newline characters. The output files now contain the original data with an added header.

Header detector Explanation:
The code provided is a MATLAB script that processes a set of files containing scrambled wave data with headers. The script reads each file, detects and removes the headers, and then writes the cleaned data to new output files. The function `removeHeadersAndProcess` is defined to handle the reading, processing, and writing of the data. It uses a simple state machine to detect consecutive rows of binary ones, which indicate the presence of headers. Once detected, these header rows are removed, and the remaining data is written to the corresponding output file. The variable names and comments have been changed to refresh the appearance of the code, but the logic and operations performed remain the same as in the original request.

The very first files and files with Reconstructed name have same content which shows the accuracy of our process.