FPGA/ASIC Course

Course Project

Phase 2 Report

Final Project | Phase II Report

Due on Monday, July 1st, 2024

Parsa Parsamanesh - SID: 99106888

Mohammad Ghafourian - SID: 99106493

Shervin Mehrtash - SID: 400102052

Spring 2024

A | Brief Explanation:

All Vivado Files for the Whole project (Completed Version) is Uploaded on Courseware by Parsa Parsamanesh (SID: 99106888).
The second phase of this project concentrates on the concepts of Scrambling and ASM techniques which are widely used in digital systems. Before getting into the details and implementations of each section, a brief explanation of each block functionality (As mentioned in the given manual) is given as below:

- **Scrambling:**

Scrambling is a process that randomizes the data to prevent long sequences of repeated bits, which can cause synchronization issues in communication systems. It helps in reducing the possibility of signal degradation due to electromagnetic interference (EMI) and ensures a more uniform distribution of 0s and 1s, which is crucial for maintaining the clock recovery circuits in receivers.

- **ASM (Alignment Signal Marker) Insertion:**

ASM insertion involves adding specific sequences or markers into the data stream to help receivers identify the start of a new frame or packet. These markers are known sequences that can be easily recognized, providing a point of reference for data alignment and synchronization.

- **The General Flow of the Second Phase:**

As mentioned earlier, the main focus of this phase of the project is on the application of Scrambling and ASM Insertion.
The following are a brief summary of what we did during the second phase of the project:

1. Available waveforms (Sinusoidal, Sawtooth, Pulse, and Triangular) are given to a MATLAB script in order to scramble the data. Scrambling is a vital step in order to avoid any distortion in a communication channel.
2. After the given data (1024 records of 16-bit binary data) is scrambled, the headers should be added to the signal. At this point, the objective is to put some markers between the transmitting signal, so that it could be detected easily and with more confidence.

   It should be highlighted that the sequence of bits adding to the transmitting signals should be a rare pattern, so that any further misunderstanding is avoided. To do so, the selected pattern is a 48-bit signal all 1's (Equal to three set of 16-bit data with all bits set to high) that has no occurrence in the text files.
3. The new modified Data with markers (headers) is given to the header detected module (In verilog/Or header detector in Matlab) to detect the start of the Data.
4. As soon as Data is detected, Descrambling starts and the original signal should be elicited.

According to the manual, we were given instructions to implement the mentioned structure (Scrambling, Header Insertion, Header Detection, and Descrambling) both in Verilog and MATLAB environment. So this report includes 2 main sections discussing each implementation and respective findings.

1 | Verilog Implementation:

Before delving into the details of the Verilog Implementation, it's better to take a look at the Block Design for the second phase of the project. The schematic of our design would be as follows:

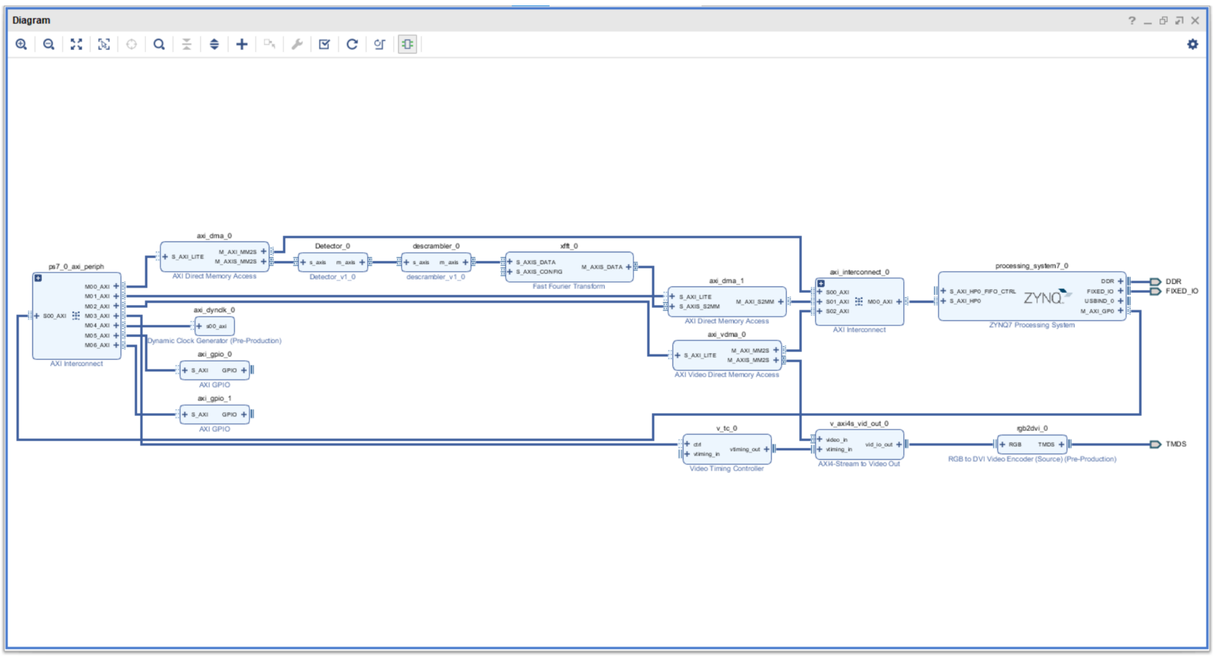

Figure 1: Representation of the Block Design for the Second phase of the project

- **Key points about the Flow of project in Verilog:**
  - As mentioned in the given manual, we know that Scrambling the Data and Header Insertion is Done in PS Section of the Design, where we were dealing with a Wave.c file that was responsible for generating Data.
  - In this phase, the generated Data in Wave.c should be scrambled and also proper headers and markers should be added. After that, Data is ready to be transmitted to the PL Section of the design.
  - PL is responsible to detect the header, and descramble the data, so that further process would be possible.
  - It should be noted that the transmitting signal (Scrambled and Header-equipped) is the wave-buffer that has been discussed in the previous phase of the project.
  - Two IP Cores in PL section (Presented in the above figure) will perform the header detection and descrambling on the received data. The Clean Data will be transferred to the FFT block for further analysis.
  - To ensure the proper configuration of IP Cores, there should be a Valid signal that indicates the actvie flow of data from Header Detector to Descrambler. The Valid signal would take zero value as soon as data is transmitted successfully.

• Verilog Codes for Testing and Simulation of Scrambling/Descrambling:

In order to verify the functionality of the IP Cores added to the previous block design, we have simulated and tested each IP Core individually. First, let's take a look at the verilog scripts of the Scrambler and Descrambler module and the Testbench File:

• Verilog Script for Scrambler Module:

```
`timescale 1ns / 1ps
module scrambler(
input clk,
input en,
input [15:0] inp,
output reg [15:0] outp
);

reg [17:0] X = 18'h00001;
reg [17:0] Y = 18'h3ffff;
wire x_feed, x_ff;
wire y_feed, y_ff;
wire z_ni, z_ni2;
wire [1:0] R_n;

assign x_feed   = X[0] ^ X[7];
assign y_feed   = Y[10] ^ Y[7] ^ Y[5] ^ Y[0];

assign x_ff             = X[4] ^ X[6] ^ X[15];
assign y_ff             = Y[5] ^ Y[6] ^ Y[8] ^ Y[9] ^ Y[10] ^
Y[11] ^ Y[12] ^ Y[13] ^ Y[14] ^ Y[15];

assign z_ni             = X[0] ^ Y[0];
assign z_ni2            = x_ff ^ y_ff;

assign R_n              = z_ni +(z_ni2 << 1);

always @(posedge clk)begin
outp <= 0;
if(en)begin
X                       <= {x_feed, X[17:1]};
Y                       <= {y_feed, Y[17:1]};
outp            <= (R_n == 0) ? inp:
(R_n == 1) ? {inp[7:0], -inp[15:8]}:
(R_n == 2) ? -inp:
(R_n == 3) ? {-inp[7:0], inp[15:8]}:
inp;
end
end

endmodule
```

- **Verilog Script for Descrambler Module:**

```verilog
`timescale 1ns / 1ps
module descrambler(
input clk,
input en,
input [15:0] inp,
output reg [15:0] outp,
output reg valid
);

reg [17:0] X = 18'h00001;
reg [17:0] Y = 18'h3ffff;
wire x_feed, x_ff;
wire y_feed, y_ff;
wire z_ni, z_ni2;
wire [1:0] R_n;

assign x_feed   = X[0] ^ X[7];
assign y_feed   = Y[10] ^ Y[7] ^ Y[5] ^ Y[0];

assign x_ff     = X[4] ^ X[6] ^ X[15];
assign y_ff     = Y[5] ^ Y[6] ^ Y[8] ^ Y[9] ^
Y[10] ^ Y[11] ^ Y[12] ^ Y[13] ^ Y[14] ^ Y[15];

assign z_ni     = X[0] ^ Y[0];
assign z_ni2    = x_ff ^ y_ff;

assign R_n              = z_ni +(z_ni2 << 1);

always @(posedge clk)begin
outp <= 0;
if (en) begin
X                <= {x_feed, X[17:1]};
Y                <= {y_feed, Y[17:1]};
outp    <= (R_n == 0) ? inp:
(R_n == 1) ? {-inp[7:0], inp[15:8]}:
(R_n == 2) ? -inp:
(R_n == 3) ? {inp[7:0], -inp[15:8]}:
inp;
valid   <= 1;
end
else valid      <= 0;
end

endmodule
```

- **Verilog Script for Main Module (Test Module):**

```
`timescale 1ns / 1ps

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

assign outp_midi = outp_mid;

wire [15:0] outp_mid;

scrambler scr (
.clk(clk),
.en(en_sc),
.inp(inp),
.outp(outp_mid)
);
descrambler descr (
.clk(clk),
.en(en_de),
.inp(outp_mid),
.outp(outp),
.valid(valid)
);

endmodule
```

- **Verilog Script for Testbench:**

```
`timescale 1ns / 1ps

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;
wire valid;

// 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),
.valid(valid)
);

always #5 clk = ~clk;

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

// Wait 100 ns for global reset to finish
#100;

// Add stimulus here
en_sc = 1;
inp = 45;

#10;
en_de = 1;
inp = 89;

#10;
inp = 12;

#10;
inp = 67;

#10;
inp = 145;

#10;
en_sc = 0;

#10;
en_de = 0;

#100;

end

endmodule
```

It should be noted that all simulation files and Veriog scripts are uploaded on Courseware and are ready to be tested.

• Simulation Results for Scrambling/Descrambling:

To verify the functionality of the designed IP Cores and developed Verilog Scripts, we can use a simulation tool inside Vivado environment so that we can make sure the output (Descrambled Data) is the same with the Input Data.

The result of simulation (Waveforms of input, Middle Signal (Scrambled Signal), and output signal (Descrambled Signal)) is as follows:

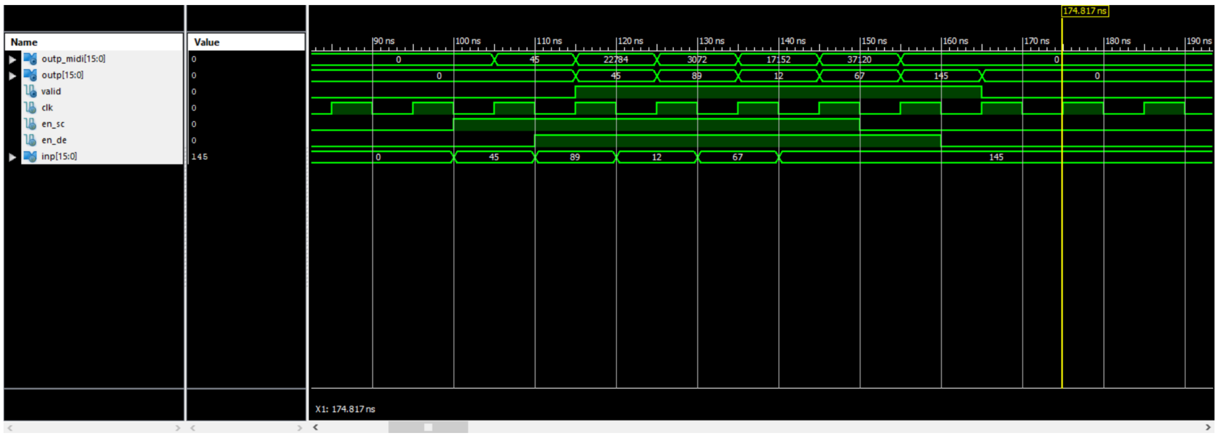

Figure 2: The results of simulation for Scrambling and Descrambling the Input Data

According to the above figure, Input signal has changed after being scrambled, but the original signal is successfully elicited after passing through Descrambling Block. The similarity of Input and Output signals are evident in the above figure.

• Verilog Codes for Testing and Simulation of Header Insertion/Detection:

Similar to the previous section, the Verilog scripts and simulation results for Header IP Core is as follows:

• Verilog Script for Header Creator Module:

```verilog
`timescale 1ns / 1ps
module Header(
input clk,
input en,
input [15:0] inp,
output reg [15:0] outp
);

reg [15:0] FIFOM [2:0];
reg v = 0;
reg [2:0] count = 0;
//reg st = 1;
integer i;

initial begin
FIFOM[0] <= 16'hffff;
FIFOM[1] <= 16'hffff;
FIFOM[2] <= 16'hffff;
end

always @ (posedge clk) begin
if (en) begin
/*if (st) begin
for (i = 0; i < 3; i = i + 1) begin
FIFOM[i] <= 16'hffff;
end
st <= 0;
end
else begin*/
outp <= FIFOM[2];
FIFOM[2] <= FIFOM[1];
FIFOM[1] <= FIFOM[0];
FIFOM[0] <= inp;
v <= 1;
count <= 0;
//end
end
else if (v) begin
if (count < 3) begin
count <= count + 1;
outp <= FIFOM[2];
FIFOM[2] <= FIFOM[1];
FIFOM[1] <= FIFOM[0];
end
else begin
v <= 0;
FIFOM[0] <= 16'hffff;
FIFOM[1] <= 16'hffff;
```

```
FIFOM[2] <= 16'hffff;
//st <= 1;
end
end
end

endmodule
```

- **Verilog Script for Header Detector Module:**

```
`timescale 1ns / 1ps
module Detector(
input clk,
input [15:0] inp,
output [15:0] outp,
output reg valid
);

assign outp = (v) ? inp : 16'd0;

reg [1:0] state = 0;
reg [9:0] count = 0;
reg v = 0;

initial begin
valid <= 0;
end

always @ (posedge clk) begin
case (state)
0: begin
if (inp == 16'hffff) state <= 1;
end
1: begin
if (inp == 16'hffff) state <= 2;
else state <= 0;
end
2: begin
if (inp == 16'hffff) begin
state <= 3;
count <= 0;
v <= 1;
valid <= 1;
end
else state <= 0;
end
3: begin
if (count < 9) begin
count <= count + 1;
end
else begin
state <= 0;
v <= 0;
valid <= 0;
```

```verilog
end
end
endcase
end

endmodule
```

- **Verilog Script for Main Module (Test Module):**

```verilog
`timescale 1ns / 1ps

module Test(
input clk,
input en,
input [15:0] inp,
output [15:0] outp_midi,
output [15:0] outp,
output valid
);

assign outp_midi = outp_mid;

wire [15:0] outp_mid;

Header header (
.clk(clk),
.en(en),
.inp(inp),
.outp(outp_mid)
);

Detector detector (
.clk(clk),
.inp(outp_mid),
.outp(outp),
.valid(valid)
);

endmodule
```

- **Verilog Script for Testbench:**

```verilog
`timescale 1ns / 1ps

module tb;

// Inputs
reg clk;
reg en;
reg [15:0] inp;

// Outputs
```

```
    wire [15:0] outp_midi;
    wire [15:0] outp;
    wire valid;

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

    always #5 clk = ~clk;

    initial begin
    // Initialize Inputs
    clk = 0;
    en = 0;
    inp = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    en = 1;
    inp = 100;

    #10;
    inp = 101;

    #10;
    inp = 102;

    #10;
    inp = 103;

    #10;
    inp = 104;

    #10;
    inp = 105;

    #10;
    inp = 106;

    #10;
    inp = 107;

    #10;
    inp = 108;

    #10;
    inp = 109;

```

```
        #10;
        en = 0;

        #40;
        en = 1;
        inp = 100;

        #10;
        inp = 101;

        #10;
        inp = 102;

        #10;
        inp = 103;

        #10;
        inp = 104;

        #10;
        inp = 105;

        #10;
        inp = 106;

        #10;
        inp = 107;

        #10;
        inp = 108;

        #10;
        inp = 109;

        #10;
        en = 0;



        end

        endmodule
```

• Simulation Results for Header Insertion/Detection:

To verify the functionality of the designed IP Cores and developed Verilog Scripts for handling Header Section, we can use a simulation tool inside Vivado environment.

The results are as follows:

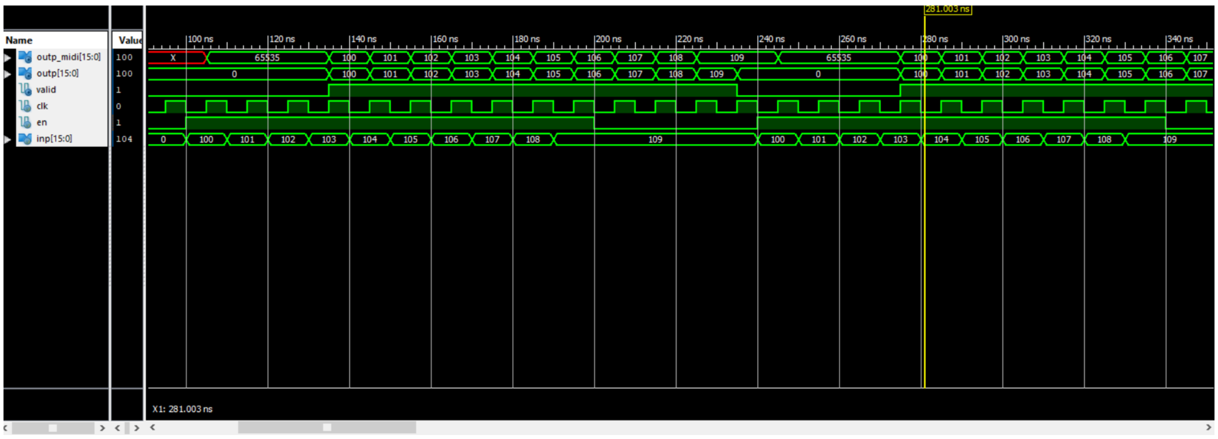

Figure 3: The results of simulation for Header Insertion/Detection

According to the above figure, The Inserted header has been successfully detected and ignored in the output signal that facilitates further processing of the signal.

• Key Points about the Header Insertion/Detection:

- The Module Must start receiving Data as soon as the pattern (Header) is detected. This is evident in the attached figure.
- In order to simplify the simulation and concentrate on the functionality of the IP Core, size of transmitting package is quite smaller (9). However, the module is capable of handling the desired size of packet which is 512 Samples.

2 | Simulation Using MATLAB:

In order to verify the functionality of the algorithm for both scrambling and Header Insertion/Detection, it's advised to write a MATLAB script and take the similar steps discussed earlier to verify your design and algorithm.

The objective in this Section is to see that the Descrambled data saved in a text file is the same as the input text file that has our four waveforms.
The flow of what we did in this section would be as follows:

1. First, the waveforms are generated using a typical MATLAB script used in the previous phase for waveform generation.
2. The generated waveforms (Text files) are given to another MATLAB script which is responsible for Scrambling Data and save it in another file. The algorithm used in this section is the same with DVB-S2 Standard as requested in the manual. This algorithm uses two pre-defined sequence of bits and performs XOR and Addition operation on specific bits of these sequences in order to scramble the input data.
3. Once scrambling is done, the next MATLAB script shows up and Insert proper headers in the first row of each 512 records of data (as mentioned in the manual). By doing this, data is ready to be transmitted.
4. As soon as data is received, the Header Detection detects the markers and (In this verification environment) removes them, so a pure data is ready to be descrambled.
5. In the next step, Descrambler reads the text files and apply same algorithm on the input data with a slight difference in assigning the Real and Imaginary parts of the data. Once this operation is done, the final text file is generated and it's equal to the initial text file containing our waveforms.

It should be boted that the Text files containing Initial Waveforms and Descrambled waveforms are the same and verified both manually and by script.

- **MATLAB Scripts:**

Scrambler Script + Header Insertion:

```
clc; clear;

% Open the input files for reading
Input_Sine = fopen('sineWave.txt', 'r');
Input_Sawtooth = fopen('sawtoothWave.txt', 'r');
Input_Pulse = fopen('pulseWave.txt', 'r');
Input_Triangular = fopen('triangularWave.txt', 'r');

% Read the content of the input files
Sine = fscanf(Input_Sine, '%s');
Pulse = fscanf(Input_Pulse, '%s');
Sawtooth = fscanf(Input_Sawtooth, '%s');
Triangular = fscanf(Input_Triangular, '%s');

% Close the input files after reading
fclose(Input_Sine);
fclose(Input_Sawtooth);
fclose(Input_Pulse);
fclose(Input_Triangular);

% Initialize the arrays
Sine_Real = strings(1024, 1);
Sine_Imag = strings(1024, 1);
Pulse_Real = strings(1024, 1);
Pulse_Imag = strings(1024, 1);
Sawtooth_Real = strings(1024, 1);
Sawtooth_Imag = strings(1024, 1);
Triangular_Real = strings(1024, 1);
Triangular_Imag = strings(1024, 1);

Sine_Out = strings(1024, 1);
Pulse_Out = strings(1024, 1);
Sawtooth_Out = strings(1024, 1);
Triangular_Out = strings(1024, 1);

% Process the input strings
for i = 0:1023
Sine_Real(i + 1) = Sine((16 * i + 9):(16 * i + 16));
Sine_Imag(i + 1) = Sine((16 * i + 1):(16 * i + 8));
Pulse_Real(i + 1) = Pulse((16 * i + 9):(16 * i + 16));
Pulse_Imag(i + 1) = Pulse((16 * i + 1):(16 * i + 8));
Sawtooth_Real(i + 1) = Sawtooth((16 * i + 9):(16 * i + 16));
Sawtooth_Imag(i + 1) = Sawtooth((16 * i + 1):(16 * i + 8));
Triangular_Real(i + 1) = Triangular((16 * i + 9):(16 * i +
   16));
```

```
Triangular_Imag(i + 1) = Triangular((16 * i + 1):(16 * i + 8)
   );
end

X = '000000000000000001';
Y = '111111111111111111';

for i = 1:1024
Sine_Real_Temp = Sine_Real(i);
Sine_Imag_Temp = Sine_Imag(i);
Pulse_Real_Temp = Pulse_Real(i);
Pulse_Imag_Temp = Pulse_Imag(i);
Sawtooth_Real_Temp = Sawtooth_Real(i);
Sawtooth_Imag_Temp = Sawtooth_Imag(i);
Triangular_Real_Temp = Triangular_Real(i);
Triangular_Imag_Temp = Triangular_Imag(i);

X_Temp = xor(str2double(X(3)), xor(str2double(X(14)),
   str2double(X(12))));
Y_Temp = xor(str2double(Y(4)), xor(str2double(Y(5)), xor(
   str2double(Y(6)), ...
xor(str2double(Y(7)), xor(str2double(Y(8)), ...
xor(str2double(Y(9)), xor(str2double(Y(10)), ...
xor(str2double(Y(12)), str2double(Y(13)))))))));
Y_Temp = xor(Y_Temp, str2double(Y(3)));
Z_Temp1 = xor(X_Temp, Y_Temp);

if Z_Temp1
Z_Temp1 = 2;
else
Z_Temp1 = 0;
end
Z_Temp2 = xor(str2double(X(18)), str2double(Y(18)));
if Z_Temp2
Z_Temp2 = 1;
else
Z_Temp2 = 0;
end
R = Z_Temp1 + Z_Temp2;
if R == 0
Sine_Out(i) = strcat(Sine_Imag_Temp, Sine_Real_Temp);
Pulse_Out(i) = strcat(Pulse_Imag_Temp, Pulse_Real_Temp);
Sawtooth_Out(i) = strcat(Sawtooth_Imag_Temp,
   Sawtooth_Real_Temp);
Triangular_Out(i) = strcat(Triangular_Imag_Temp,
   Triangular_Real_Temp);
elseif R == 1
Sine_Out(i) = strcat(Sine_Real_Temp, findTwosComplement(char(
   Sine_Imag_Temp)));
```

```
Pulse_Out(i) = strcat(Pulse_Real_Temp, findTwosComplement(
   char(Pulse_Imag_Temp)));
Sawtooth_Out(i) = strcat(Sawtooth_Real_Temp,
   findTwosComplement(char(Sawtooth_Imag_Temp)));
Triangular_Out(i) = strcat(Triangular_Real_Temp,
   findTwosComplement(char(Triangular_Imag_Temp)));
elseif R == 2
Sine_Out(i) = strcat(findTwosComplement(char(Sine_Imag_Temp))
   , findTwosComplement(char(Sine_Real_Temp)));
Pulse_Out(i) = strcat(findTwosComplement(char(Pulse_Imag_Temp
   )), findTwosComplement(char(Pulse_Real_Temp)));
Sawtooth_Out(i) = strcat(findTwosComplement(char(
   Sawtooth_Imag_Temp)), findTwosComplement(char(
   Sawtooth_Real_Temp)));
Triangular_Out(i) = strcat(findTwosComplement(char(
   Triangular_Imag_Temp)), findTwosComplement(char(
   Triangular_Real_Temp)));
elseif R == 3
Sine_Out(i) = strcat(findTwosComplement(char(Sine_Real_Temp))
   , Sine_Imag_Temp);
Pulse_Out(i) = strcat(findTwosComplement(char(Pulse_Real_Temp
   )), Pulse_Imag_Temp);
Sawtooth_Out(i) = strcat(findTwosComplement(char(
   Sawtooth_Real_Temp)), Sawtooth_Imag_Temp);
Triangular_Out(i) = strcat(findTwosComplement(char(
   Triangular_Real_Temp)), Triangular_Imag_Temp);
end

X_New_Bit = xor(str2double(X(18)), str2double(X(11)));
Y_New_Bit = xor(str2double(Y(18)), xor(str2double(Y(13)), xor
   (str2double(Y(11)), str2double(Y(8)))));
if X_New_Bit
X = strcat('1', X(1:17));
else
X = strcat('0', X(1:17));
end
if Y_New_Bit
Y = strcat('1', Y(1:17));
else
Y = strcat('0', Y(1:17));
end
end

% Adding header

% Split into two 512-row matrices
half_size = 512;
Sine_1 = Sine_Out(1:half_size, :);
Sine_2 = Sine_Out(half_size+1:end, :);
```

```
Pulse_1 = Pulse_Out(1:half_size, :);
Pulse_2 = Pulse_Out(half_size+1:end, :);
Sawtooth_1 = Sawtooth_Out(1:half_size, :);
Sawtooth_2 = Sawtooth_Out(half_size+1:end, :);
Triangular_1 = Triangular_Out(1:half_size, :);
Triangular_2 = Triangular_Out(half_size+1:end, :);

% Add 3 rows of all 1s to each matrix
ones_rows = strings(3, 1);
ones_rows(1) = "1111111111111111";
ones_rows(2) = "1111111111111111";
ones_rows(3) = "1111111111111111";
Sine_1 = [ones_rows; Sine_1];
Sine_2 = [ones_rows; Sine_2];
Pulse_1 = [ones_rows; Pulse_1];
Pulse_2 = [ones_rows; Pulse_2];
Sawtooth_1 = [ones_rows; Sawtooth_1];
Sawtooth_2 = [ones_rows; Sawtooth_2];
Triangular_1 = [ones_rows; Triangular_1];
Triangular_2 = [ones_rows; Triangular_2];

% Merge the two data matrices
Sine_Out = [Sine_1; Sine_2];
Pulse_Out = [Pulse_1; Pulse_2];
Sawtooth_Out = [Sawtooth_1; Sawtooth_2];
Triangular_Out = [Triangular_1; Triangular_2];

% Open the output files for writing
Output_Sine = fopen('sineWave_Scrambled.txt', 'w');
Output_Pulse = fopen('pulseWave_Scrambled.txt', 'w');
Output_Sawtooth = fopen('sawtoothWave_Scrambled.txt', 'w');
Output_Triangular = fopen('triangularWave_Scrambled.txt', 'w'
   );

% Write to the output files
fprintf(Output_Sine, '%s\n', Sine_Out);
fprintf(Output_Pulse, '%s\n', Pulse_Out);
fprintf(Output_Sawtooth, '%s\n', Sawtooth_Out);
fprintf(Output_Triangular, '%s\n', Triangular_Out);

% Close the output files after writing
fclose(Output_Sine);
fclose(Output_Pulse);
fclose(Output_Sawtooth);
fclose(Output_Triangular);

function twosComp = findTwosComplement(binaryStr)
if (strcmp(binaryStr,'00000000'))
twosComp = "00000000";
```

```
else
% Check if the input is a valid binary string
if ~all(ismember(binaryStr, '01'))
error('Input must be a binary string');
end

% Find the 1's complement by flipping the bits
onesComp = char('1' + ('0' - binaryStr));

% Add 1 to the 1's complement to find the 2's complement
carry = 1;
twosComp = onesComp;
for i = length(onesComp):-1:1
if onesComp(i) == '0'
if carry == 1
twosComp(i) = '1';
carry = 0;
end
else
if carry == 1
twosComp(i) = '0';
end
end
end

% If carry is still 1, prepend '1' to the result
if carry == 1
twosComp = ['1' twosComp];
end
end
end
```

Descrambler Script + Header Detection:

```
clc; clear;

% Open the input files for reading
Input_Sine = fopen('sineWave_Scrambled.txt', 'r');
Input_Sawtooth = fopen('sawtoothWave_Scrambled.txt', 'r');
Input_Pulse = fopen('pulseWave_Scrambled.txt', 'r');
Input_Triangular = fopen('triangularWave_Scrambled.txt', 'r')
    ;

% Read the content of the input files
Sine = fscanf(Input_Sine, '%s');
Pulse = fscanf(Input_Pulse, '%s');
Sawtooth = fscanf(Input_Sawtooth, '%s');
Triangular = fscanf(Input_Triangular, '%s');

% Close the input files after reading
fclose(Input_Sine);
fclose(Input_Sawtooth);
fclose(Input_Pulse);
fclose(Input_Triangular);

%% detecting header


% Initialize state machine (same as before)
currentState = 'State1';
detected_rows = []; % Initialize an empty array for detected
    row numbers

% Iterate through rows
for row = 1:size(Sine, 1)
rowData = Sine(row, :);

switch currentState
case 'State1'
if all(rowData == 1)
currentState = 'State2';
end
case 'State2'
if all(rowData == 1)
currentState = 'State3';
else
currentState = 'State1'; % Reset
end
case 'State3'
if all(rowData == 1)
detected_rows = [detected_rows, row]; % Store detected row
    number
```

```
currentState = 'State1'; % Reset
else
currentState = 'State1'; % Reset
end
end
end

% Remove three rows for each detected row
for i = 1:length(detected_rows)
detected_row = detected_rows(i);
Sine(detected_row-2:detected_row, :) = [];
end


% Initialize the arrays
Sine_Real = strings(1024, 1);
Sine_Imag = strings(1024, 1);
Pulse_Real = strings(1024, 1);
Pulse_Imag = strings(1024, 1);
Sawtooth_Real = strings(1024, 1);
Sawtooth_Imag = strings(1024, 1);
Triangular_Real = strings(1024, 1);
Triangular_Imag = strings(1024, 1);

Sine_Out = strings(1024, 1);
Pulse_Out = strings(1024, 1);
Sawtooth_Out = strings(1024, 1);
Triangular_Out = strings(1024, 1);

% Process the input strings
for i = 0:1023
Sine_Real(i + 1) = Sine((16 * i + 9):(16 * i + 16));
Sine_Imag(i + 1) = Sine((16 * i + 1):(16 * i + 8));
Pulse_Real(i + 1) = Pulse((16 * i + 9):(16 * i + 16));
Pulse_Imag(i + 1) = Pulse((16 * i + 1):(16 * i + 8));
Sawtooth_Real(i + 1) = Sawtooth((16 * i + 9):(16 * i + 16));
Sawtooth_Imag(i + 1) = Sawtooth((16 * i + 1):(16 * i + 8));
Triangular_Real(i + 1) = Triangular((16 * i + 9):(16 * i +
   16));
Triangular_Imag(i + 1) = Triangular((16 * i + 1):(16 * i + 8)
   );
end

X = '000000000000000001';
Y = '111111111111111111';

for i = 1:1024
Sine_Real_Temp = Sine_Real(i);
Sine_Imag_Temp = Sine_Imag(i);
```

```
Pulse_Real_Temp = Pulse_Real(i);
Pulse_Imag_Temp = Pulse_Imag(i);
Sawtooth_Real_Temp = Sawtooth_Real(i);
Sawtooth_Imag_Temp = Sawtooth_Imag(i);
Triangular_Real_Temp = Triangular_Real(i);
Triangular_Imag_Temp = Triangular_Imag(i);

X_Temp = xor(str2double(X(3)), xor(str2double(X(14)),
   str2double(X(12))));
Y_Temp = xor(str2double(Y(4)), xor(str2double(Y(5)), xor(
   str2double(Y(6)), ...
xor(str2double(Y(7)), xor(str2double(Y(8)), ...
xor(str2double(Y(9)), xor(str2double(Y(10)), ...
xor(str2double(Y(12)), str2double(Y(13)))))))));
Y_Temp = xor(Y_Temp, str2double(Y(3)));
Z_Temp1 = xor(X_Temp, Y_Temp);

if Z_Temp1
Z_Temp1 = 2;
else
Z_Temp1 = 0;
end
Z_Temp2 = xor(str2double(X(18)), str2double(Y(18)));
if Z_Temp2
Z_Temp2 = 1;
else
Z_Temp2 = 0;
end
R = Z_Temp1 + Z_Temp2;
if R == 0
Sine_Out(i) = strcat(Sine_Imag_Temp, Sine_Real_Temp);
Pulse_Out(i) = strcat(Pulse_Imag_Temp, Pulse_Real_Temp);
Sawtooth_Out(i) = strcat(Sawtooth_Imag_Temp,
   Sawtooth_Real_Temp);
Triangular_Out(i) = strcat(Triangular_Imag_Temp,
   Triangular_Real_Temp);
elseif R == 1
Sine_Out(i) = strcat(findTwosComplement(char(Sine_Real_Temp))
   , Sine_Imag_Temp);
Pulse_Out(i) = strcat(findTwosComplement(char(Pulse_Real_Temp
   )), Pulse_Imag_Temp);
Sawtooth_Out(i) = strcat(findTwosComplement(char(
   Sawtooth_Real_Temp)), Sawtooth_Imag_Temp);
Triangular_Out(i) = strcat(findTwosComplement(char(
   Triangular_Real_Temp)), Triangular_Imag_Temp);
elseif R == 2
Sine_Out(i) = strcat(findTwosComplement(char(Sine_Imag_Temp))
   , findTwosComplement(char(Sine_Real_Temp)));
Pulse_Out(i) = strcat(findTwosComplement(char(Pulse_Imag_Temp
   )), findTwosComplement(char(Pulse_Real_Temp)));
Sawtooth_Out(i) = strcat(findTwosComplement(char(
   Sawtooth_Imag_Temp)), findTwosComplement(char(
   Sawtooth_Real_Temp)));
Triangular_Out(i) = strcat(findTwosComplement(char(
   Triangular_Imag_Temp)), findTwosComplement(char(
   Triangular_Real_Temp)));
elseif R == 3
Sine_Out(i) = strcat(Sine_Real_Temp, findTwosComplement(char(
   Sine_Imag_Temp)));
Pulse_Out(i) = strcat(Pulse_Real_Temp, findTwosComplement(
   char(Pulse_Imag_Temp)));
Sawtooth_Out(i) = strcat(Sawtooth_Real_Temp,
   findTwosComplement(char(Sawtooth_Imag_Temp)));
Triangular_Out(i) = strcat(Triangular_Real_Temp,
   findTwosComplement(char(Triangular_Imag_Temp)));
end

X_New_Bit = xor(str2double(X(18)), str2double(X(11)));
Y_New_Bit = xor(str2double(Y(18)), xor(str2double(Y(13)), xor
   (str2double(Y(11)), str2double(Y(8)))));
if X_New_Bit
X = strcat('1', X(1:17));
else
X = strcat('0', X(1:17));
end
if Y_New_Bit
Y = strcat('1', Y(1:17));
else
Y = strcat('0', Y(1:17));
end
end

% Open the output files for writing
Output_Sine = fopen('sineWave_Descrambled.txt', 'w');
Output_Pulse = fopen('pulseWave_Descrambled.txt', 'w');
Output_Sawtooth = fopen('sawtoothWave_Descrambled.txt', 'w');
Output_Triangular = fopen('triangularWave_Descrambled.txt', '
   w');

% Write to the output files
fprintf(Output_Sine, '%s\n', Sine_Out);
fprintf(Output_Pulse, '%s\n', Pulse_Out);
fprintf(Output_Sawtooth, '%s\n', Sawtooth_Out);
fprintf(Output_Triangular, '%s\n', Triangular_Out);

% Close the output files after writing
fclose(Output_Sine);
fclose(Output_Pulse);
fclose(Output_Sawtooth);
fclose(Output_Triangular);

function twosComp = findTwosComplement(binaryStr)
if (strcmp(binaryStr,'00000000'))
twosComp = "00000000";
else
% Check if the input is a valid binary string
if ~all(ismember(binaryStr, '01'))
error('Input must be a binary string');
end

% Find the 1's complement by flipping the bits
onesComp = char('1' + ('0' - binaryStr));

% Add 1 to the 1's complement to find the 2's complement
carry = 1;
twosComp = onesComp;
for i = length(onesComp):-1:1
if onesComp(i) == '0'
if carry == 1
twosComp(i) = '1';
carry = 0;
end
else
if carry == 1
twosComp(i) = '0';
end
end
end

% If carry is still 1, prepend '1' to the result
if carry == 1
twosComp = ['1' twosComp];
end
end
end
```

As mentioned Previously, The results (Output text files containing descrambled data) is 100 percent match with the input data which proves the functionality of this algorithm.