# FPGA Project: Phase 2 Report

AmirReza Imani MohammadReza HajiBabaei
MohammadParsa Abedi

July, 2024

## Contents

# 1 Introduction

The objective of this project is to process simple signals on the Zynq-7010 platform and display the signal and its FFT on an HDMI screen. The project involves generating various types of waveforms, scrambling the data using a DVB-S2 standard scrambler, transmitting the data from the Processing System (PS) to the Programmable Logic (PL), and finally displaying the processed signals. This report covers the step-by-step implementation of Phase 2, focusing on the scrambling mechanism and data transmission protocol.

# 2 Project Overview

## 2.1 Phase 1 Recap

In Phase 1, different types of signals (sine, square, triangle, sawtooth, and subsawtooth) were generated, and the data was transmitted from the PS to the PL via DMA, and then displayed using HDMI.

## 2.2 Phase 2 Objectives

- **Add 80-bit Header**: Each 512-bit frame should include an 80-bit header, which contains 54 bytes of data.
- **Scramble Data**: Scramble the data using a scrambler based on the DVB-S2 standard.
- **Restart Scrambler**: The scrambler should restart to its initial value at the beginning of each frame.

# 3 Detailed Steps and Methodology

## 3.1 Signal Generation

First, we generate different waveforms. The following functions generate sine, square, triangle, sawtooth, and subsawtooth waves respectively:

```
void GetSinWave(int point, int max_amp, int amp_val, u8 *sin_tab)
    {
    // Implementation
}

void GetSquareWave(int point, int max_amp, int amp_val, u8 *
   Square_tab) {
    // Implementation
}

void GetTriangleWave(int point, int max_amp, int amp_val, u8 *
   Triangle_tab) {
    // Implementation
}
```

```
void GetSawtoothWave(int point, int max_amp, int amp_val, u8 *
    Sawtooth_tab) {
    // Implementation
}

void GetSubSawtoothWave(int point, int max_amp, int amp_val, u8 *
    SubSawtooth_tab) {
    // Implementation
}
```

Listing 1: Signal Generation Functions

## 3.2 Header Definition

The header is defined as a structure `CCS_DS_Header` containing various fields:

- `protocol_id`
- `source_addr`
- `dest_addr`
- `msg_type`
- `length`
- `checksum`

A function `generate_header()` initializes this header.

```
typedef struct {
    uint8_t protocol_id;
    uint16_t source_addr;
    uint16_t dest_addr;
    uint8_t msg_type;
    uint16_t length;
    uint16_t checksum;
} CCS_DS_Header;

CCS_DS_Header generate_header(uint8_t protocol_id, uint16_t
    source_addr, uint16_t dest_addr, uint8_t msg_type, uint16_t
    length) {
    // Implementation
}
```

Listing 2: Header Definition

### 3.2.1 Explanation of Header Fields

The header fields serve specific purposes to ensure proper data transmission and integrity:

- `protocol_id`: This field identifies the protocol being used for communication. It ensures that the receiver can interpret the incoming data correctly according to the defined protocol.
- `source_addr`: This field specifies the address of the sender. It is crucial for identifying the origin of the data, which is particularly important in systems with multiple transmitting nodes.
- `dest_addr`: This field specifies the address of the intended receiver. It ensures that the data is directed to the correct destination, enabling proper routing and handling of the data.
- `msg_type`: This field indicates the type of message being transmitted. Different message types might require different processing, and this field helps in distinguishing among them.
- `length`: This field specifies the length of the data payload. It helps the receiver allocate the correct amount of memory for incoming data and ensures that the entire payload is received and processed.
- `checksum`: This field is used for error-checking. The checksum is calculated based on the other header fields, and the receiver can recalculate the checksum to verify data integrity. Any mismatch indicates potential data corruption.

## 3.3 Scrambler Initialization and Scrambling

The scrambler is based on a 15-bit LFSR (Linear Feedback Shift Register). It is initialized with a specific value and then used to scramble the data.

- `init_scrambler()`: Initializes the scrambler.
- `scramble_data()`: Scrambles the data using the scrambler.

```
typedef struct {
    uint16_t shift_register;
} Scrambler;

void init_scrambler(Scrambler *scrambler) {
    scrambler->shift_register = 0x0401; // Initial value
}

void scramble_data(Scrambler *scrambler, uint8_t *data, size_t
   length) {
    // Implementation
}
```

Listing 3: Scrambler Initialization and Scrambling

### 3.3.1 Explanation of DVB-S2 Standard Scrambler

The DVB-S2 standard scrambler is a critical component in digital communication systems, ensuring that data transmitted over the airwaves is randomized. This randomness helps to minimize the impact of errors and improve signal integrity. Here are the key aspects of the DVB-S2 scrambler:

- **Linear Feedback Shift Register (LFSR)**: The scrambler uses a 15-bit LFSR, which is a simple yet effective way to generate a pseudo-random sequence of bits. This sequence is used to scramble the data, making it less predictable and more resistant to interference and noise.
- **Initialization (Scramble Key)**: The scrambler is initialized with a specific value (in this case, `0x0401`). This initial state, also known as the scramble key, is crucial for the scrambling and descrambling process. Both the transmitter and receiver must use the same initial state to ensure the data can be correctly descrambled.

### 3.3.2 Benefits of Using DVB-S2 Standard Scrambler

Using a standard scrambler like the one defined in DVB-S2 offers several benefits:

- **Error Minimization**: Scrambling data reduces the likelihood of long runs of consecutive identical bits, which can cause synchronization problems and increase error rates.
- **Improved Signal Integrity**: By spreading the signal more evenly across the spectrum, the scrambler helps to mitigate the effects of narrowband interference and improves overall signal integrity.
- **Compatibility**: Using a standard scrambler ensures compatibility with other DVB-S2 compliant systems, making it easier to integrate and communicate with different devices and networks.

### 3.3.3 Importance of Initial State (Scramble Key)

The initial state of the scrambler, also known as the scramble key, is critical for the following reasons:

- **Synchronization**: The transmitter and receiver must use the same scramble key to ensure the data is correctly descrambled. Any mismatch in the initial state will result in corrupted data.
- **Security**: While the primary purpose of the scrambler is not security, using a unique scramble key can add a layer of protection against casual eavesdropping, as the data will appear random without the correct key.
- **Data Integrity**: The scramble key helps maintain data integrity by ensuring the scrambler's pseudo-random sequence is correctly aligned for both transmission and reception.

## 3.4 Frame Preparation

A frame consists of the header followed by the scrambled data. The function `insert_header_and_scramb` performs this task.

```
void insert_header_and_scramble(Scrambler *scrambler, uint8_t *
   data, CCS_DS_Header header) {
    // Convert header to byte array
    uint8_t header_bytes[HEADER_LENGTH];
    memcpy(header_bytes, &header, sizeof(header));

    // Insert header at the beginning of the frame
    for (int i = 0; i < HEADER_LENGTH; i++) {
        data[i] = header_bytes[i];
    }

    // Scramble the rest of the frame data
    scramble_data(scrambler, data + HEADER_LENGTH, DATA_LENGTH);
}
```

Listing 4: Frame Preparation

### 3.4.1 Explanation of Frame Preparation

The `insert_header_and_scramble` function is responsible for preparing a frame by adding a header and scrambling the data. Here's a detailed explanation of what it does and how it does it:

- **Convert Header to Byte Array**: The header structure is converted into a byte array. This allows for easy insertion into the data array.
- **Insert Header**: The header bytes are inserted at the beginning of the data array. This ensures that each frame starts with the necessary metadata for the receiver to interpret the data correctly.
- **Scramble Data**: The function then calls `scramble_data` to scramble the rest of the data in the frame, starting right after the header. This ensures that the data portion of the frame is randomized, improving error resilience.

## 3.5 Processing and Saving Waveforms

Each waveform is processed, framed, scrambled, and saved into separate files. This is done using the `process_wave()` function.

```
void process_wave(const char* filename, void (*wave_func)(int,
   int, int, u8*), CCS_DS_Header header, const char* wave_name) {
    // Implementation
}
```

Listing 5: Processing and Saving Waveforms

### 3.6 Main Function

The main function generates the header, processes each waveform, and saves the output to files.

Listing 6: Main Function

```
int main() {
    CCS_DS_Header header = generate_header(0x01, 0x1001, 0x2001, 0x01, FRAM

    process_wave("sin_wave.txt", GetSinWave, header, "Sin");
    process_wave("square_wave.txt", GetSquareWave, header, "Square");
    process_wave("triangle_wave.txt", GetTriangleWave, header, "Triangle");
    process_wave("sawtooth_wave.txt", GetSawtoothWave, header, "Sawtooth");
    process_wave("sub_sawtooth_wave.txt", GetSubSawtoothWave, header, "SubS

    return 0;
}
```

## 4 Explanation and Reasoning

### 4.1 Signal Generation

Each signal generation function calculates the appropriate sample values based on the waveform type. These functions take the total number of points, maximum amplitude, and the desired amplitude as parameters.

### 4.2 Header Definition

The header ensures that each frame can be correctly identified and processed. The checksum is a simple one's complement sum of the header fields, ensuring data integrity.

### 4.3 Scrambler

The scrambler uses a simple feedback mechanism based on the DVB-S2 standard to ensure that the transmitted data is randomized, which helps in minimizing the impact of errors during transmission.

### 4.4 Frame Preparation

Inserting the header at the beginning of each frame and scrambling the data helps in aligning the data for the receiver to correctly descramble and process the incoming data.

## 5 Methodology

1. **Generate Header**: Create a header for each frame to be transmitted.
2. **Generate Waveforms**: Use waveform generation functions to create data arrays for different waveforms.

3. **Scramble Data**: Initialize the scrambler and scramble the data.
4. **Prepare Frame**: Insert the header and scrambled data into the frame.
5. **Save Output**: Save the framed and scrambled data into text files.

# 6 MATLAB Simulation

In this section, we create four types of waves: triangular, square, sine, and sawtooth. After generating these waves, we scramble them and later descramble them in the PL side. We compare the results to ensure data integrity.

```
%% setting the properties
clear
clc
n = 1024;
Fs = n;
T = 1/Fs;
L = n;
t = (0:L-1)*T;
freq = 2;

%% making wave
% square wave
t_s = linspace(0,10,n);
y_square = square(t_s) * 0.9844;

% traingular wave
ramp_up = linspace(0, 1, n/2);
ramp_down = linspace(1, 0, n/2);
y_tri = [ramp_up ramp_down] * 0.9844;

%sawtooth wave
f = 2;
t_saw = linspace(0, f, n);
y_saw = sawtooth(2 * pi * t_saw) * 0.9844;

% sin wave
y_sin = sin(2*pi*freq*t) * 0.9844;
```

Listing 7: MATLAB Code for Wave Generation and Scrambling

The above MATLAB code sets up the parameters and generates four different types of waves: square, triangular, sawtooth, and sine. Each wave is normalized to fit within a specific amplitude range.

Next, we convert these waveforms to a fixed-point binary format, scramble them, and add a header to the scrambled data.

```
%% converting wave to fixed point array format
% Fixed point conversion code here

%% scrambling data
```

```
clc
initial_x = zeros([1,18]);
initial_x(18) = 1;
initial_y = ones([1,18]);
sr_x = initial_x;
sr_y = initial_y;
[sr_x,sr_y] = shift(sr_x,sr_y);

% Scramble data code here
```

The 'shift' function initializes the LFSR registers used for scrambling the data. The data is then scrambled using the 'lfsr' and 'scramble' functions.

```
%% adding header
frame_length = 64;
number_of_frames = n / frame_length;
header_length = 4;
sin_header = zeros([n + header_length * number_of_frames,16]);
% Other header initializations

frame_pattern = ones([1,16]);
k = 1;
for i = 1:68:n + header_length * number_of_frames
    for j = i:i+67
        if(j-i<4)
            sin_header(j,:) = frame_pattern;
            % Other header settings
        else
            sin_header(j,:) = sin_scrambled(k,:);
            % Other scrambled data settings
            k = k + 1;
        end
    end
end

%% writing scrambled data to files
writeFile('sin_scramble.txt',sin_header);
% Other write operations
```

Listing 8: Adding Header and Writing to File

In the above MATLAB code, we add a header to the scrambled data and then write it to text files. The header is crucial for synchronizing and identifying the start of each data frame.

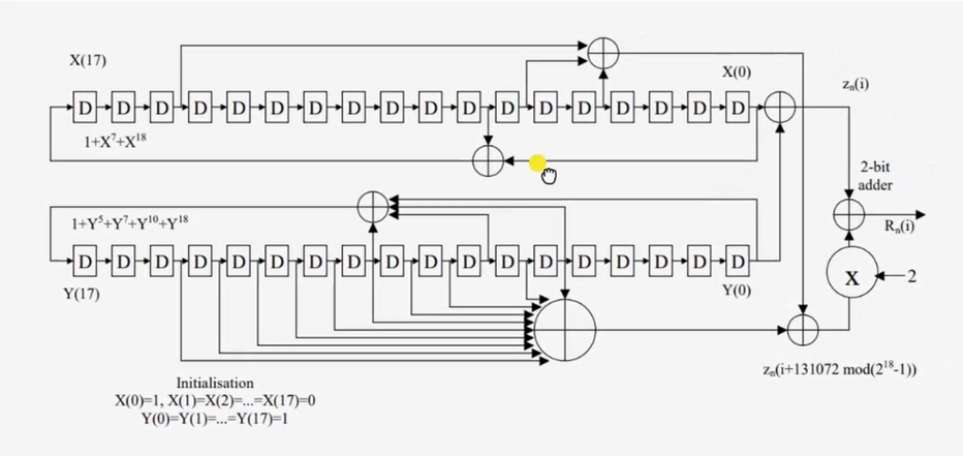
```
%% drawing the signals
sin_descramble = readFile('sin_descramble.txt',L,16);
% Read other descrambled files

% Convert descrambled data back to decimal
% Plotting the original and descrambled data for comparison
```

Listing 9: MATLAB Code for Descrambling and Comparison

This part of the MATLAB code reads the descrambled data, converts it back to decimal, and plots the results for comparison with the original waves.

# 7 PL Side Implementation

The PL side acts as the receiver. It uses the Rn value and header to descramble the incoming streamed data and outputs the original data. The following Verilog code implements this:




```
1  `timescale 1ns / 1ps
2
3  module Descramble(
4      input clk,
5      input reset,
6      input [15:0]scramble_data,
7      output reg [15:0]descramble_data,
8      output reg data_valid
9      );
10
11     reg [17:0]sr_x = 18'b100000000000000000;
12     reg [17:0]sr_y = 18'b011111111111111111;
13     reg [17:0]initial_x = 18'b000000000000000001;
14     reg [17:0]initial_y = 18'b111111111111111111;
15     reg [15:0]header_pattern = 16'b1111111111111111;
16     wire a,b,d;
17     wire [1:0] c;
18     wire [1:0] R;
19     reg [1:0]state;
20     reg [3:0] header_counter;
21     parameter IDLE = 2'b0;
22     parameter descramble = 2'b1;
23     assign a = sr_x[5] ^ sr_x[7] ^ sr_x[16];
24     assign b = sr_y[6] ^ sr_y[7] ^ sr_y[9] ^ sr_y[10] ^ sr_y[11]
          ^ sr_y[12] ^ sr_y[13] ^ sr_y[14] ^ sr_y[15] ^ sr_y[16];
25     assign c = (a ^ b) * 2;
26     assign d = sr_x[1] ^ sr_y[1];
```

```
27      assign R = {1'b0,d} + c;
```

Listing 10: Verilog Code for Descrambling

The Verilog code initializes the descrambling process by defining the registers and LFSR. The 'Descramble' module uses these to descramble the incoming data based on the header and Rn value.

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

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

```
1       always@(posedge clk)
2       begin
3           if(!reset)
4           begin
5               sr_x <= initial_x;
6               sr_y <= initial_y;
7               header_counter <= 0;
8               data_valid <= 0;
9               state <= IDLE;
10          end
11          else
12          begin
13              case(state)
14                  IDLE:
15                  begin
16                      if(scramble_data == header_pattern)
17                      begin
18                          header_counter <= header_counter + 1;
19                          if(header_counter == 3)
20                          begin
21                              header_counter <= 0;
22                              state <= descramble;
23                              data_valid <= 0;
24                          end
25                      end
26                      else
27                      begin
28                          header_counter <= 0;
29                          data_valid <= 0;
30                      end
31                  end
32                  descramble:
33                  begin
```

```
34                      if(scramble_data != header_pattern)
35                      begin
36                          data_valid <= 1;
37                          sr_x[16] <= sr_x[17];
38                          sr_x[15] <= sr_x[16];
39                          sr_x[14] <= sr_x[15];
40                          sr_x[13] <= sr_x[14];
41                          sr_x[12] <= sr_x[13];
42                          sr_x[11] <= sr_x[12];
43                          sr_x[10] <= sr_x[11];
44                          sr_x[9] <= sr_x[10];
45                          sr_x[8] <= sr_x[9];
46                          sr_x[7] <= sr_x[8];
47                          sr_x[6] <= sr_x[7];
48                          sr_x[5] <= sr_x[6];
49                          sr_x[4] <= sr_x[5];
50                          sr_x[3] <= sr_x[4];
51                          sr_x[2] <= sr_x[3];
52                          sr_x[1] <= sr_x[2];
53                          sr_x[0] <= sr_x[1];
54                          sr_x[17] <= sr_x[0] ^ sr_x[7];
55
56                          sr_y[16] <= sr_y[17];
57                          sr_y[15] <= sr_y[16];
58                          sr_y[14] <= sr_y[15];
59                          sr_y[13] <= sr_y[14];
60                          sr_y[12] <= sr_y[13];
61                          sr_y[11] <= sr_y[12];
62                          sr_y[10] <= sr_y[11];
63                          sr_y[9] <= sr_y[10];
64                          sr_y[8] <= sr_y[9];
65                          sr_y[7] <= sr_y[8];
66                          sr_y[6] <= sr_y[7];
67                          sr_y[5] <= sr_y[6];
68                          sr_y[4] <= sr_y[5];
69                          sr_y[3] <= sr_y[4];
70                          sr_y[2] <= sr_y[3];
71                          sr_y[1] <= sr_y[2];
72                          sr_y[0] <= sr_y[1];
73                          sr_y[17] <= sr_y[0] ^ sr_y[5] ^ sr_y[7] ^
                                sr_y[10];
74
75                          case(R)
76                              0:
77                                  descramble_data <= scramble_data;
78                              1:
79                              begin
80                                  descramble_data[15:8] <=  ~
                                       scramble_data[7:0] + 8'b1;
81                                  descramble_data[7:0] <=
                                       scramble_data[15:8];
```

```
                                end
                                2:
                                begin
                                    descramble_data[15:8] <= ~
                                       scramble_data[15:8] + 8'b1;
                                    descramble_data[7:0] <= ~
                                       scramble_data[7:0] + 8'b1;
                                end
                                3:
                                begin
                                    descramble_data[15:8] <=
                                       scramble_data[7:0];
                                    descramble_data[7:0] <= ~
                                       scramble_data[15:8] + 8'b1;
                                end
                            endcase
                        end
                        else
                        begin
                            data_valid <= 0;
                            state <= IDLE;
                            header_counter <= 1;
                        end
                    end
                    default:
                    begin
                        state <= IDLE;
                        data_valid <= 0;
                    end
                endcase
            end
        end
endmodule
```

Listing 11: Verilog Code for Descrambling State Machine

In the above Verilog code, the state machine is implemented to handle descrambling. The 'IDLE' state checks for the header pattern and transitions to the 'descramble' state once the header is detected. In the 'descramble' state, the data is descrambled based on the Rn value and outputted.

In this part, to do the reverse operation of the previous table. We simulate the following code using the test bench:

```
`timescale 1ns / 1ps
module Descramble_tb();
    reg clk = 0;
    reg reset;
    reg [15:0]scramble_data[0:1087];
    reg [15:0]descramble_data[0:1023];
    wire data_valid;
    wire [15:0]descramble_out;
    reg [7:0]out;
```

```
    reg [15:0]scramble_in;
    reg [3:0]counter = 0;
    reg [10:0] i = 0;
    reg [10:0] j = 0;
    always #10 clk = ~clk;
    initial
    begin
        $readmemb("C:/Users/babbage/Desktop/university/FPGA/
           project/phase2_myself/tri_scramble.txt", scramble_data
           );
        reset = 0;
        #20
        reset = 1;
    end
    always@(posedge clk)
    begin
        if(counter > 10)
        begin
            scramble_in <= scramble_data[i];
            i <= i + 1;
            if(data_valid == 1)
            begin
                descramble_data[j] <= descramble_out;
                out <= descramble_out[7:0];
                j <= j + 1;
            end
        end
        else
        begin
            counter <= counter + 1;
        end
    end
    initial
    begin
        wait(j == 1024);
        $writememb("C:/Users/babbage/Desktop/university/FPGA/
           project/phase2_myself/tri_descramble.txt",
           descramble_data);
    end
    Descramble uut(
        .clk(clk),
        .reset(reset),
        .scramble_data(scramble_in),
        .descramble_data(descramble_out),
        .data_valid(data_valid)
    );
endmodule
```

Listing 12: Verilog Code for Descrambling State Machine

In this test bench, the first data scrambled by MATLAB is read from the desired file and stored in a RAM, then every clock one of these data is given to the module and the

output, if valid, is stored in another RAM . Finally, this array is saved in a file containing descrambled data and drawn in MATLAB.

# 8 Conclusion

The results of the descrambling process are compared with the original signals. The MATLAB plots show that the descrambled data matches the original waves, demonstrating the correctness of the descrambling process.

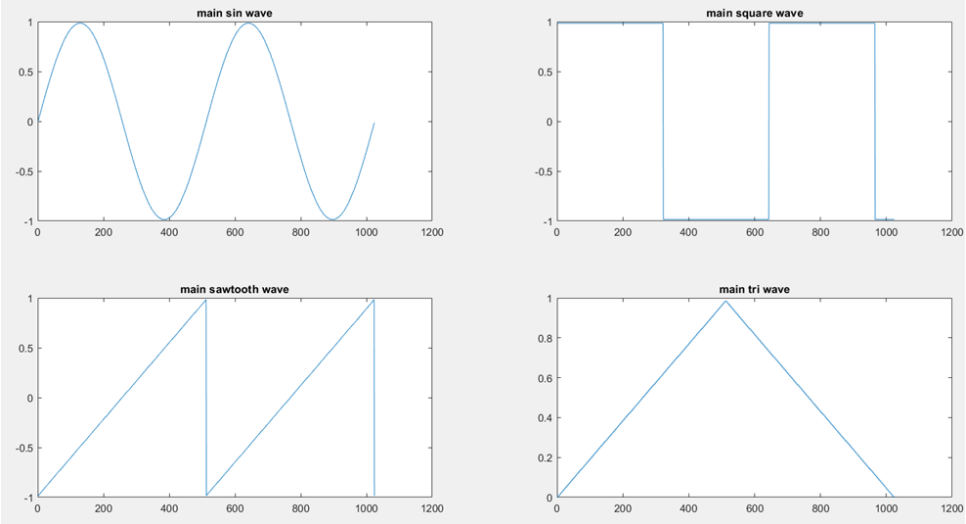



Figure 1: Original Waves

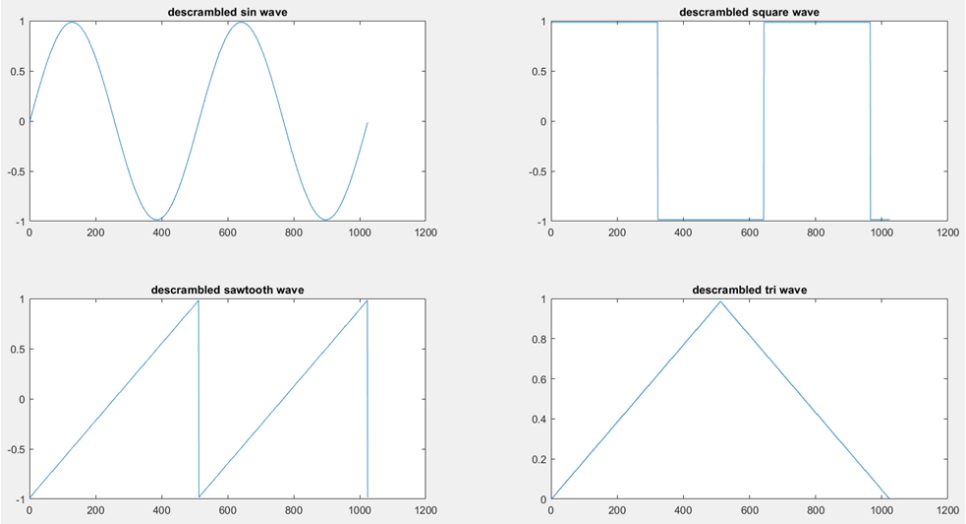



Figure 2: Descrambled Waves

As shown in the figures, the descrambled waves are identical to the original waves, confirming the accuracy of the implemented descrambling logic in the FPGA.