ASIC & FPGA Chip Design

Spring 2024

Dr. Mahdi Shaabani

Project Assignment: Stage 2 :

Transmission

Arman yazdani 400102255

Zahra Mojtahedin 99102167

1 Intro

At this point, a scrambler module has been added to the Processing System (PS) along with a corresponding descrambler module in the Programmable Logic (PL) of a system-on-chip (SoC) device. This scrambling process follows the DVB-S (Digital Video Broadcasting - Satellite) physical layer scrambling standard, which is a widely used technique in satellite communications.

We first discuss the creation of these scrambler and descrambler modules using the C programming language and the Verilog hardware description language. The scrambler module is implemented in C, while the descrambler module is implemented in Verilog. These modules work together to scramble and descramble the data, ensuring secure and reliable data transmission.

Next, we test these modules using a testbench, which is a simulation environment that allows us to verify the functionality of the modules. During the testing phase, we try to minimize the error in the communicated data by treating any bugs or issues that are discovered in our implementation. This involves debugging the code, refining the algorithms, and ensuring that the scrambler and descrambler modules work seamlessly together.

Finally, we explain how the descrambler Verilog module can be added to the block design of the SoC device and how it can intercept the numbers in between the Direct Memory Access (DMA) and the Fast Fourier Transform (FFT) blocks. This integration of the descrambler module into the overall system design allows for the seamless processing of the scrambled data, ensuring that the data is properly descrambled before being passed to the subsequent signal processing blocks.

2 Matlab:simulation

Here we try generating 1024 8-bit samples:

2.1 Sinusoid

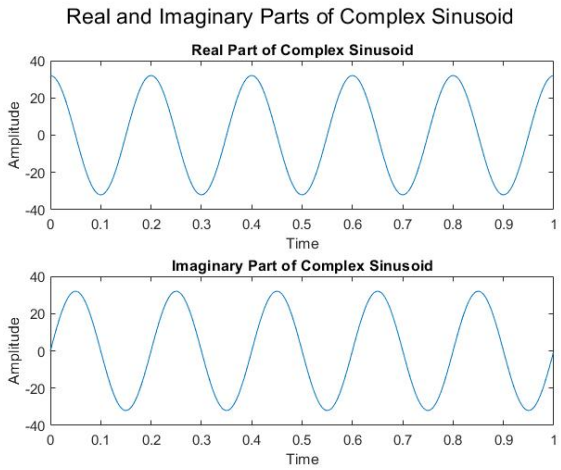

Figure 1: Sine wave

```
% Generate 1024 sample complex sinusoid and save in a text file
fs = 1024; % Sampling frequency
f = 5; % Frequency of the sinusoid
t = 0:1/fs:1-1/fs; % Time vector
complex_sinusoid = 32 * exp(1i * 2 * pi * f * t); % Generate complex sinusoid

% Extract real and imaginary parts
x_real = real(complex_sinusoid);
x_imag = imag(complex_sinusoid);

% Convert real and imaginary parts to 8-bit integers
x_real_int = uint16(x_real); % Convert real part to 8-bit integer
x_imag_int = uint16(x_imag); % Convert imaginary part to 8-bit integer

% Concatenate real and imaginary parts into 16-bit format
data = bitor(bitshift(x_real_int, 8), x_imag_int);
```

```
17
18                  % Save the data to a txt file
19                  fid = fopen('sinusoid_samples.txt', 'wt');
20                  fprintf(fid, '%04X\n', data);
21                  fclose(fid);
22                  % Plot the real part
23                  subplot(2,1,1);
24                  plot(t, x_real);
25                  title('Real Part of Complex Sinusoid');
26                  xlabel('Time');
27                  ylabel('Amplitude');
28
29                  % Plot the imaginary part
30                  subplot(2,1,2);
31                  plot(t, x_imag);
32                  title('Imaginary Part of Complex Sinusoid')
                        ;
33                  xlabel('Time');
34                  ylabel('Amplitude');
35
36                  sgtitle('Real and Imaginary Parts of
                        Complex Sinusoid');
37                  disp('Data saved to complex_sinusoid.txt');
```

2.2 Pulse

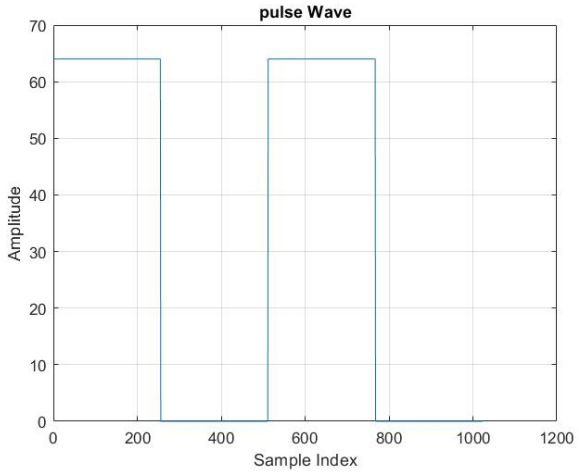

Figure 2: Pulse wave

```
% Generate 1024 samples of a pulse signal with higher frequency
t = 0:1023;
x = zeros(1, 1024);
x(1:256) = 64;
x(513:768) = 64;

% Convert the samples to 16-bit values
samples = uint16(x);

% Combine the real and imaginary parts into a single 16-bit value
samples_combined = bitshift(samples, 8);

% Write the samples to a text file
fid = fopen('pulse_samples.txt', 'w');
fprintf(fid, '%04X\n', samples_combined);
fclose(fid);
% Plot the pulse wave
figure;
plot(t, x);
grid on;
xlabel('Sample Index');
```

```
ylabel('Amplitude');
title('pulse Wave');
```

2.3 Triangle

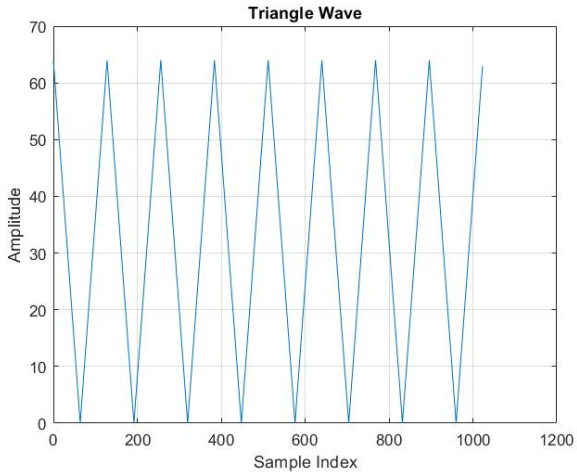

Figure 3: Triangle wave

```
% Generate 1024 samples of a non-negative triangle wave
t = 0:1023;
x = abs(mod(t, 128) - 64);

% Convert the samples to 16-bit values
real_part = uint16(x);
imag_part = uint16(zeros(1, 1024));

% Combine the real and imaginary parts into a single 16-bit value
samples_combined = bitor(bitshift(real_part, 8), imag_part);

% Write the samples to a text file
fid = fopen('triangle_samples.txt', 'w');
fprintf(fid, '%04X\n', samples_combined);
fclose(fid);

% Plot the triangle wave
```

```
figure;
plot(t, x);
grid on;
xlabel('Sample Index');
ylabel('Amplitude');
title('Triangle Wave');
```

2.4 Sawtooth

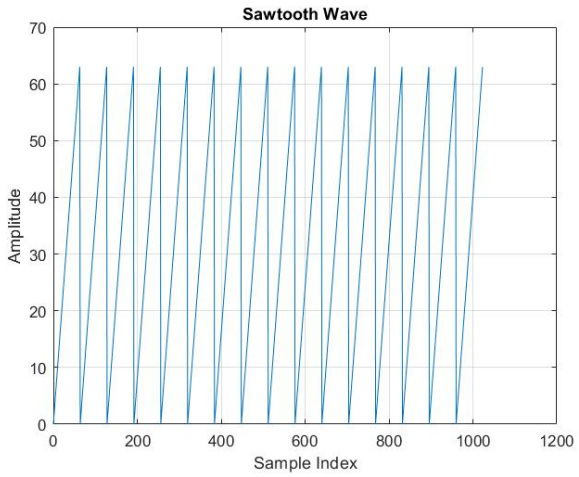

Figure 4: Sawtooth wave

```
clc, clear
% Generate 1024 samples of a sawtooth wave
t = 0:1023;
x = mod(t,64);

% Convert the samples to 16-bit values
real_part = uint16(x);
imag_part = uint16(zeros(1, 1024));

% Combine the real and imaginary parts into a single 16-bit value
samples_combined = bitor(bitshift(real_part, 8), imag_part);

% Write the samples to a text file
fid = fopen('sawtooth_samples.txt', 'w');
```

```
fprintf(fid, '%04X\n', samples_combined);
fclose(fid);

% Plot the sawtooth wave
figure;
plot(t, x);
grid on;
xlabel('Sample Index');
ylabel('Amplitude');
title('Sawtooth Wave');
```

2.5 Scrambler

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

Figure 5: Scrambling method

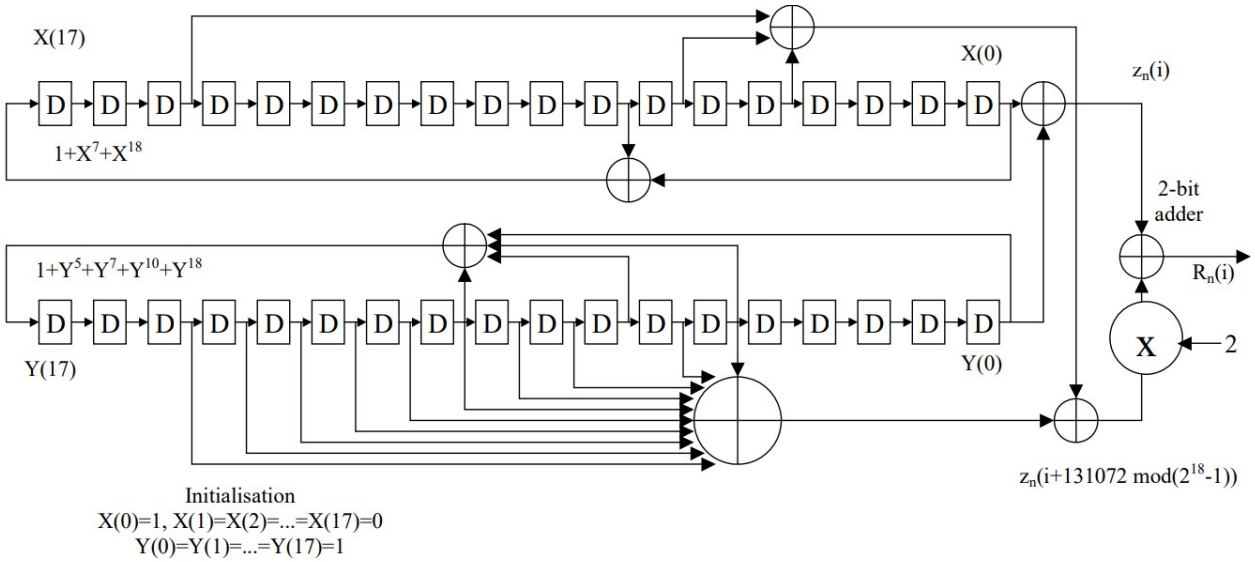

Figure 6: Scrambling structure

```
clc, clear
% Open the input text file for reading
fileID = fopen('pulse_samples.txt', 'r');
```

```
X = zeros(1,18);
Y = zeros(1,18);
% Read the data from the file
data = textscan(fileID, '%s', 'Delimiter', '\n');
fclose(fileID);

% Extract the hex values from the data
Values = data{1};

% Initialize output data
outputData = cell(1024, 1);
% Process each value and generate the output
for i = 1:1024
% Generate a random 2-bit number R using RandomGen function
[R,X,Y] = RandomGen(i,X,Y);  % Replace with the actual RandomGen function call

I = int8(hex2dec(Values{i}(1:2)));
Q = int8(hex2dec(Values{i}(3:4)));
switch R
case 0
Is = I;
Qs = Q;
case 1
Is = bitcmp(Q) + 1;  % Two's complement for negative value
Qs = I;
case 2
Is = bitcmp(I) + 1 ;  % Two's complement for negative value
Qs = bitcmp(Q) + 1;  % Two's complement for negative value
case 3
```

```
33                  Is = Q;
34                  Qs = bitcmp(I) + 1;  % Two's complement for
                        negative value
35                  end
36
37                  % Concatenate Is and Qs to form the output
                        value
38                  outputData{i} = [Char(dec2hex(int16(Is))),
                        Char(dec2hex(int16(Qs)))];
39                  end
40
41                  % Write the output data to a new text file
42                  outputFileID = fopen('scrambled.txt', 'w');
43                  for i = 1:1024
44                  fprintf(outputFileID, '%s\n', outputData{i
                        });
45                  end
46                  fclose(outputFileID);
47                  %%
48                  function ch = Char(in)
49                  ch = in;
50                  if(length(in)==1)
51                  ch = ['0',in];
52                  end
53                  end
54                  %%
55                  function [R,X,Y] = RandomGen(i,X,Y)
56                  if(i==1)
57                  X(1) = 1;
58                  Y = ones(1,18);
59                  else
60                  X18 = bitxor(X(1),X(8));
61                  Y18 = bitxor(bitxor(Y(1),Y(6)),bitxor(Y(8),
                        Y(11)));
62                  for j=1:17
63                  X(j) = X(j+1);
```

```
64          Y(j) = Y(j+1);
65          end
66          X(18) = X18;
67          Y(18) = Y18;
68          end
69          T1 = bitxor(X(1),Y(1));
70          T2 = bitxor(bitxor(X(5),X(7)),X(16));
71          T3 = bitxor(Y(6),Y(7));
72          for k=9:16
73          T3 = bitxor(T3,Y(k));
74          end
75          T4 = bitxor(T3,T2);
76          R = T4*2 + T1;
77          end
```

2.6 Header

Frame structure is as follows:

- ASM:Length = 4 Content = 4×*100*s
- Frame:Length = 1024

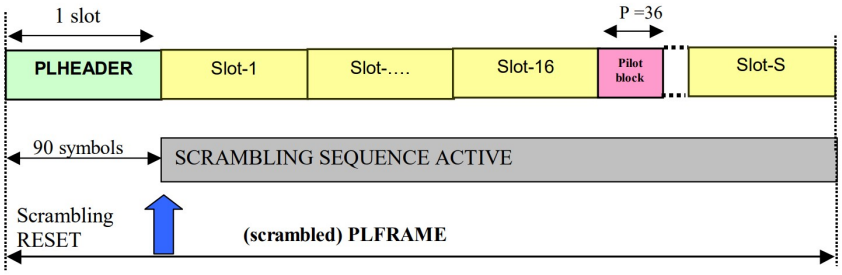

Figure 7: Header style

```
1           numbers = [100, 100, 100, 100];
2
3           % Open the original file in append mode and
                copy content to a temporary file
4           copyfile('scrambled.txt', 'temp_file.txt');
5
6           % Open the original file in write mode
7           fileID = fopen('header.txt', 'w');
```

```
8
9                   % Write the 4 numbers at the beginning
                        again
10                  fprintf(fileID, '%04X\n', numbers);
11
12                  % Append the content of the temporary file
                        to the original file
13                  tempFileID = fopen('temp_file.txt', 'r');
14                  while ~feof(tempFileID)
15                  line = fgetl(tempFileID);
16                  fprintf(fileID, '%s\n', line);
17                  end
18
19                  % Close the files
20                  fclose(fileID);
21                  fclose(tempFileID);
22
23                  % Delete the temporary file
24                  delete('temp_file.txt');
```

3 C:PS

Let's see what happens in PS for Scrambling(pretty much like matlab):

```
1               #include <stdio.h>
2               #include "scrambler.h"
3
4               static u32 x, y;
5
6               static void resetState() {
7                       x = XStart;
8                       y = YStart;
9               }
10
11              static u8 calculateZ(void) {
12                      return ((x & 0x1) + (y & 0x1)) % 2;
13              }
14
```

```
15  static u8 calculateZPrime(void) {
16          return (((y >> 15) & 0x1) + ((y >> 14) & 0
                x1) + ((y >> 13) & 0x1) + ((y >> 12) & 0
                x1) +
17          ((y >> 11) & 0x1) + ((y >> 10) & 0x1) + ((y
                >> 9) & 0x1) + ((y >> 8) & 0x1) +
18          ((y >> 6) & 0x1) + ((y >> 5) & 0x1) + ((x
                >> 15) & 0x1) + ((x >> 6) & 0x1) +
19          ((x >> 4) & 0x1)) % 2;
20  }
21
22  static u8 calculateR(void) {
23          u8 z = calculateZ();
24          u8 zPrime = calculateZPrime();
25          zPrime <<= 1;
26          return z + zPrime;
27  }
28
29  static void updateXY(void) {
30          u8 xAdd = (((x) & 0x1) + ((x >> 7) & 0x1))
                % 2;
31          u8 yAdd = (((y) & 0x1) + ((y >> 5) & 0x1) +
                ((y >> 7) & 0x1) + ((y >> 10) & 0x1)) %
                2;
32
33          x >>= 1;
34          y >>= 1;
35          x = (xAdd << 17) | x;
36          y = (yAdd << 17) | y;
37  }
38
39  void scramble(u8 realPart, u8 imagPart, u16 *dout)
        {
40          u8 r = calculateR();
41          printf("x = %d, y = %d\n", x, y);
42          printf("r = %d\n", r);
```

```
43
44                      u16 realPart2 = realPart;
45                      u16 imagPart2 = imagPart;
46
47                      switch (r) {
48                              case 0:
49                              *dout = (imagPart2 << 8) |
                                   realPart2;
50                              break;
51                              case 1:
52                              *dout = (realPart2 << 8) | ((~
                                   imagPart2 + 1) & 0xFF);
53                              break;
54                              case 2:
55                              *dout = (((~imagPart2 + 1) & 0xFF)
                                   << 8) | ((~realPart2 + 1) & 0xFF
                                   );
56                              break;
57                              case 3:
58                              *dout = (((~realPart2 + 1) & 0xFF)
                                   << 8) | (imagPart2 & 0xFF);
59                              break;
60                              default:
61                              *dout = (imagPart2 << 8) |
                                   realPart2;
62                              break;
63                      }
64
65                      updateXY();
66              }
67
68              void reset() {
69                      resetState();
70              }
```

The scrambler module is implemented in the C programming language, as it will be used in

the Software Development Kit (SDK) of the system at a later stage. The header file "scrambler.h" and the implementation file "scrambler.c" contain the necessary code to implement the scrambler based on the DVB-S (Digital Video Broadcasting - Satellite) standard.

The code includes two main functions:

The reset() function: This function must be called explicitly whenever a full packet is transmitted. It resets the internal state of the scrambler, specifically the X and Y shift registers, to their original values. This ensures that the scrambling process starts from a known state for each new packet.

The scrambler() function: This function accepts the real part and the imaginary part of the input signal at each time step. It then performs the scrambling operation on these values and writes the modified values directly to the output variable, which is passed as a reference argument.

To test the scrambler module, it is integrated with another wave generator function from a previous phase. This wave generator function is used to create 1024 samples of a sine wave. The scrambler function is then applied to these samples, and the scrambled data is written to a text file. This text file is later used as input to the Verilog testbench, which is responsible for testing the performance of the descrambler module.

By implementing the scrambler in C and providing a well-defined interface through the header file, the module can be easily integrated into the SDK and used in the overall system design. The explicit reset function and the direct modification of the output variable in the scrambler() function ensure that the scrambling process is controlled and can be seamlessly incorporated into the data processing pipeline.

This approach of using C for the scrambler and Verilog for the descrambler allows for a modular and flexible design, where the scrambler can be updated or modified independently without affecting the rest of the system. The testing process, which involves generating a sine wave, scrambling it, and using the scrambled data to test the descrambler, helps to validate the correctness and performance of the overall scrambling and descrambling system.

There's also the header file:

```
1  #include <stdint.h>
2
3  typedef uint8_t u8;
4  typedef uint16_t u16;
5  typedef uint32_t u32;
```

```
6
7                    #define XStart 1U
8                    #define YStart 262143U
9
10                   void scrambler(u8 realPart, u8 imagPart,
                        u16 *dout);
11                   void reset();
```

4 HDL:PL

Now we try to implement the opposing structure meaning *Deheader* and *Descrambler*.

4.1 Deheader

Here we try to detect 4 consecutive *16'h0064* or *16'd0100*:

4.1.1 Main Module

```
1                    `timescale 1ns/1ns
2                    module Deheader(
3                    //-----------------------Port directions
                        and deceleration
4                    input clk,
5                    input [15:0]in,
6                    output reg valid,
7                    output [15:0]out
8                    );
9                    reg start = 0;
10                   reg [10:0]Counter = 0;
11                   reg [2:0] next = 0;
12                   reg [2:0] current = 0;
13                   assign out = valid ? in : 16'bx;
14                   always @(in,current) begin
15                   case (current)
16                   0:begin
17                   next = 0;
18                   if (in == 100)
19                   next = 1;
20                   end
21                   1:begin
22                   next = 0;
```

```
23                          if (in == 100)
24                          next = 2;
25                          end
26                          2:begin
27                          next = 0;
28                          if (in == 100)
29                          next = 3;
30                          end
31                          3:begin
32                          next = 0;
33                          if (in == 100)
34                          next = 4;
35                          end
36                          4:begin
37                          next = 0;
38                          if (in == 100)begin
39                          start = 1;
40                          Counter = 0;
41                          end
42                          end
43                          default: next = 0;
44                          endcase
45                          end
46                          always @(posedge clk) begin
47                          if(start)begin
48                          Counter <= Counter + 1;
49                          valid <= 1;
50                          end
51                          current <= next;
52                          if (Counter==1023)begin
53                          valid <= 0;
54                          next <= 0;
55                          start <= 0;
56                          end
57                          end
58                          endmodule
```

4.1.2 Simulation

```
`timescale 1ns/1ns

module Deheader_tb();
//------------------generating clock signal in 50MHz
reg clk = 1'b0;
always @(clk)
clk <= #10 ~clk;
wire valid;
wire [15:0] out;
reg [15:0] in;
integer i;
//-------------------------------------reg declaration
initial begin
in = 0;
#30;
in = 1000;
#20;
in = 100;
#20;
in = 100;
#20;
in = 100;
#20;
in = 100;
#20;
for(i=0;i<1024;i=i+1) begin
in = i;
#20;
end
end
Deheader uut (
.clk(clk),
.valid(valid),
.in(in),
.out(out)
```

```
36                                  );
37                                  endmodule
```

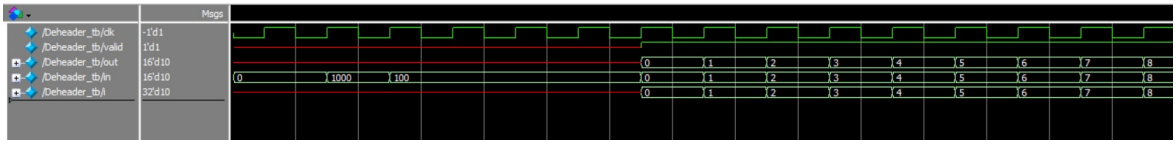

Figure 8: Header detection

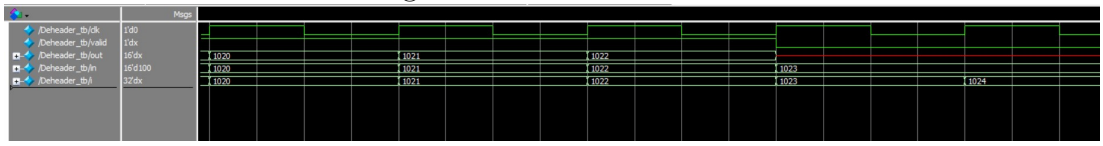

Figure 9: Frame end

4.2 Descrambler

4.2.1 Main Module

```
1                                   `timescale 1ns / 1ps
2                                   module Descrambler(clk, rst, in, out, R);
3                                   input wire clk;
4                                   input wire rst;
5                                   input wire [15:0] in;
6                                   output reg [15:0] out;
7
8                                   wire [7:0] Real;
9                                   wire [7:0] Imag;
10
11                                  assign Real = in[7:0];
12                                  assign Imag = in[15:8];
13
14                                  reg [17:0] X;
15                                  reg [17:0] Y;
16                                  reg T1 ;
17                                  reg T2;
18
19                                  wire [1:0] T3;
20                                  assign T3 = T2 << 1;
21
22                                  output wire [1:0] R;
23                                  assign R = T1 + T3;
24
```

```
always @(posedge clk, negedge rst) begin
if (!rst) begin
out <= 16'bx;
X = 18'd1;
Y = 18'd262143;
end
else begin
X[16:0] <= X[17:1];
X[17] <= (X[0] + X[7]);
Y[16:0] <= Y[17:1];
Y[17] <= (Y[0] + Y[5] + Y[7] + Y[10]);

T1 <= X[0] + Y[0];
T2 <= Y[15] + Y[14] + Y[13] + Y[12] + Y[11] + Y[
   10] + Y[9] + Y[8] + Y[6] + Y[5] + X[4] + X[6]
   + X[15];

case (R)
0: begin
out <= {{Real},{Imag}};
end
1: begin
out <= {{Imag},{~Real + 8'd1}};
end
2: begin
out <= {{~Real + 8'd1},{~Imag + 8'd1}};
end
3: begin
out <= {{~Imag + 8'd1},{Real}};
end
endcase
end
end
endmodule
```

4.2.2 Simulation

```
`timescale 1ns / 1ps
module Descrambler_tb();
```

```
3                       reg clk;
4                       reg rst;
5                       reg [31:0] in;
6                       wire [15:0] out;
7                       wire [1:0] R;
8
9                       reg [31:0] samples [0:1023];
10                      reg [31:0] wave_Samples [0:1023];
11                      reg [31:0] wave_data;
12
13                      Descrambler uut(.clk(clk), .rst(rst), .in
                           (in), .out(out), .R(R));
14
15                      always #5 clk = ~clk;
16
17                      integer i;
18
19                      initial begin
20                      $dumpfile("Descrambler.vcd");
21                      $dumpvars;
22
23                      $readmemh("triangle_scrambled.txt",
                           samples);
24                      $readmemh("triangle_samples.txt",
                           wave_Samples);
25
26                      clk = 0;
27                      rst = 0;
28
29                      #15
30                      rst = 1;
31
32                      for (i = 0; i < 1024; i = i+1) begin
33                      #10;
34                      in = samples[i];
35                      wave_data = wave_Samples[i];
36                      end
37
```

```
38          #10;
39          end
40          endmodule
```

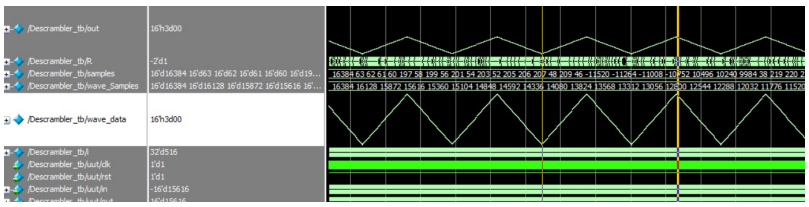

Figure 10: triangle wave reconstructing(They're equivalant,scaling decieves)

4.3 Block Design Enhancement

In order to integrate the new scrambling and descrambling functionality into our overall system block design, we must first consider the existing interfaces and protocols used in the system.

We note that the output of the Direct Memory Access (DMA) IP core in our system follows the Stream AXI protocol. This is a common interface used for streaming data in AXI-based systems. On the other hand, the input of the Fast Fourier Transform (FFT) IP core also expects data in the Stream AXI format.

To seamlessly integrate the descrambler module between the DMA and the FFT, we have created and packaged a custom AXI4 peripheral IP with two master and slave interfaces. This custom IP serves as an intermediary component in the data processing pipeline.

The slave interface of the custom IP will read the data from the DMA IP core, which is in the Stream AXI format. The custom IP will then store this data in an internal First-In-First-Out (FIFO) buffer.

Next, the master interface of the custom IP will retrieve the data from the FIFO and forward it to the FFT IP core, also in the Stream AXI format. However, before passing the data to the FFT, the custom IP will utilize the descrambler module to decode the input values.

By inserting this custom AXI4 peripheral IP between the DMA and the FFT, we can intercept the data stream, apply the descrambling operation, and then pass the descrambled data to the FFT IP core for further processing.

This approach allows us to seamlessly integrate the new descrambler functionality into the existing block design without modifying the DMA or the FFT IP cores. The custom AXI4 peripheral IP acts as a bridge, handling the data transfer and the descrambling process transparently to the rest of the system.

By leveraging the AXI4 protocol and creating a custom IP with the necessary interfaces, we can ensure a smooth integration of the descrambler module into the overall system architecture, enabling the system to process the descrambled data correctly before passing it to the subsequent signal processing blocks, such as the FFT.

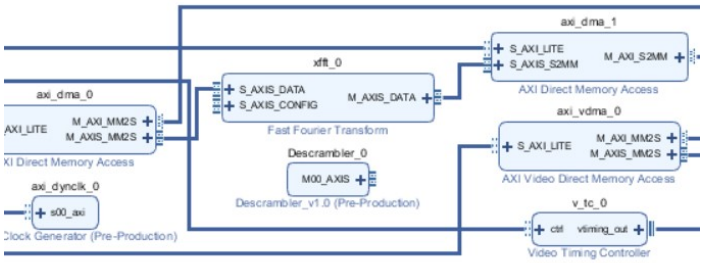

Figure 11: Final Block design:input:scrambled,output:descrambled