Experiment 3 – Function Generator

Mehdi Jamalkhah, 810100111

Mobina Mehrazar, 810100216

***Abstract*— This document is a report for experiment #3 of Digital Logic Design Laboratory at ECE department, University of Tehran. The purpose of this experiment is to generate a wide variety of waveforms with different amplitude and frequency with an Arbitrary Function Generator (AFG).**

***Keywords*— Sine, reciprocal, frequency, FPGA, DDS, waveform, clock, module, Verilog, ROM.**

I. Introduction

The goal of this report is to design an Arbitrary Generator that is capable of generating reciprocal, square, triangle, sine, full-wave rectified, and half-wave rectified.

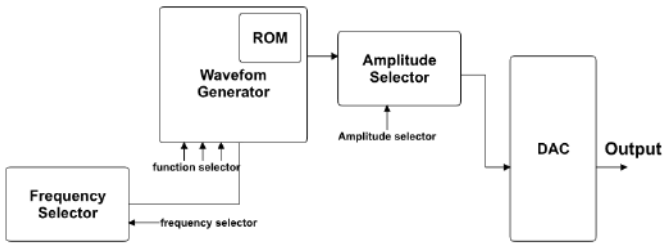

Fig. 1 Block diagram of the Arbitrary Generator (AFG) : there is a main component that generates one of the desired waveform based on the function selectors' value, a frequency selector that sets the output signal frequency, and amplitude selector. DAC is also used to convert the digital output to an analog signal, which can be observed and evaluated via an oscilloscope. A ROM memory is usually embedded to store any other arbitrary waveform.

II. Waveform Generator

This part of project produces desired functions. Output of this module is an 8-bit digital representing the amplitude of signal.

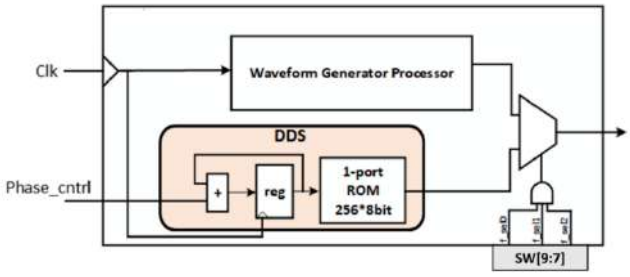

Fig. 2 Block diagram of waveform generator

Waveforms square, reciprocal and triangle are based on a counter that counts up or down with each clock for the period of the waveform.

A. Reciprocal

This waveform will be created by following formula:

$$Rep(x) = \frac{255}{(255 - x) + 1}$$

Where x is the output of counter.

```
1   `timescale 1ns/1ns
2
3   module ReciprocalWave(clk, rst, ReciprocalWave);
4           input clk, rst;
5           output reg [7:0] ReciprocalWave =  8'd0;
6
7           reg [7:0] count_out = 8'b0;
8
9           always @(posedge clk) begin
10
11                  if (rst) begin
12                          ReciprocalWave = 8'b0;
13                  end
14
15                  else if (count_out == 8'b0) begin
16                          ReciprocalWave = 8'b0;
17                  end
18
19                  else begin
20
21                          ReciprocalWave = 8'd255 / (8'd255 - count_out + 8'd1);
22                  end
23          end
24
25          always @(posedge clk) begin
26
27                  if (rst) begin
28                          count_out = 8'b0;
29                  end
30
31                  else begin
32                          count_out = count_out + 1;
33                  end
34          end
35
36  endmodule
37
```

Fig. 3 Reciprocal Verilog code

B. Square

```
`timescale 1ns/1ns

module SquareWave(clk, rst, squareWave);
        input clk, rst;
        output reg [7:0] squareWave =  8'b1111_1111;

        reg [7:0] count_out = 8'b0;

        always @(posedge clk) begin

                if (rst) begin
                        squareWave = 8'b1111_1111;
                end

                else if (count_out % 8'd128 == 8'd0) begin
                        squareWave = ~squareWave;
                end
        end

        always @(posedge clk) begin

                if (rst) begin
                        count_out = 8'b0;
                end

                else begin
                        count_out = count_out + 1;
                end
        end

endmodule
```

Fig. 3 Square Verilog code

C. Triangle

This wave can be created by following formula:

$$Triangle(x) = x$$

Where x is output of counter.

```
`timescale 1ns/1ns

module TriangleWave(clk, rst, TriangleWave);
        input clk, rst;
        output reg [7:0] TriangleWave =  8'd0;

        reg [7:0] count_out = 8'b0;

        always @(posedge clk) begin

                if (rst) begin
                        TriangleWave = 8'd0;
                end

                else if (count_out > 8'd128) begin
                        TriangleWave = 8'd128 - (count_out - 8'd128);
                end

                else begin
                        TriangleWave = count_out;
                end
        end

        always @(posedge clk) begin

                if (rst) begin
                        count_out = 8'b0;
                end

                else begin
                        count_out = count_out + 1;
                end
        end

endmodule
```

Fig. 4 Triangle Verilog code

D. Sine types

The full-wave and half-wave rectified waveforms are both generated based on the sine wave. The following second order differential equation can be used to generate the sine function:

$$\sin(n) = \sin(n-1) + \mathrm{a} \times \cos(n-1)$$

$$\cos(n) = \cos(n-1) - \mathrm{a} \times \sin(n)$$

In order to do the mathematical operations with reasonable accuracy, operations are done in 16-bit fixed point. Also, considering the period of about 256 clock cycles from frequency selector, the equations turn to:

$$\sin(n) = \sin(n-1) + \frac{1}{64} \times \cos(n-1)$$

$$\cos(n) = \cos(n-1) - \frac{1}{64} \times \sin(n)$$

0 was used for sin(0) and 30000 was used for cos(0).The values are between -32768 to 32767 for sin and cos.

```
`define SIN_0 16'd0
`define COS_0 16'd30000
`define OFFSET 8'd127

module SinandRectifiedWave(clk, rst, sinWave, fullRectifiedWave, halfRectifiedWave);
        input clk, rst;
        output reg [7:0] sinWave = `SIN_0,
                         fullRectifiedWave = `SIN_0,
                         halfRectifiedWave = `SIN_0;

        reg [15:0] sin_out = `SIN_0 , cos_out = `COS_0, fullRectified = `SIN_0, halfRectified = `SIN_0;
        reg [7:0] sin_out_offset, cos_out_offset ;
        reg [7:0] count_out = 8'b0;

        always @(posedge clk) begin
                if (rst) begin
                        sin_out = `SIN_0;
                        cos_out = `COS_0;
                        fullRectified = `SIN_0;
                        halfRectified = `SIN_0;
                end
                else begin
                        sin_out = sin_out + {{6{cos_out[15]}}, cos_out[15:6]};
                        cos_out = cos_out - {{6{sin_out[15]}}, sin_out[15:6]};
                        fullRectified = (sin_out[15]) ? -sin_out : sin_out;
                        halfRectified = (sin_out[15]) ? 16'b0 : sin_out;
                end
                sin_out_offset = sin_out[15:8] + `OFFSET;
                cos_out_offset = cos_out[15:8] + `OFFSET;
                sinWave = sin_out_offset;
                fullRectifiedWave = fullRectified[15:8];
                halfRectifiedWave = halfRectified[15:8];
        end
        always @(posedge clk) begin
                if (rst) begin
                        count_out = 8'b0;
                end

                else begin
                        count_out = count_out + 1;
                end
        end
endmodule
```

Fig. 5 Sine, full-rectified, and half-rectified Verilog code

The top level of this part can be seen in following picture.

```
`define SIN_0 16'd0
`define COS_0 16'd30000
`define OFFSET 8'd127

`timescale 1ns/1ns
module WaveformGenerator(clk, rst, waveSelector, dataOut, romIn);
        input clk, rst;
        input [2:0] waveSelector;
        input [7:0] romIn;
        output reg [7:0] dataOut;


        wire [7:0] reciprocalWave, squareWave, triangleWave, sinWave, fullRectifiedWave,
                halfRectifiedWave;

        ReciprocalWave Reciprocal(.clk(clk), .rst(rst), .ReciprocalWave(reciprocalWave));
        SquareWave Square(.clk(clk), .rst(rst), .squareWave(squareWave));
        TriangleWave Triangle(.clk(clk), .rst(rst), .TriangleWave(triangleWave));
        SinandRectifiedWave Sin(.clk(clk), .rst(rst), .sinWave(sinWave), .fullRectifiedWave(fullRectifiedWave),
                                .halfRectifiedWave(halfRectifiedWave));

        always @(waveSelector, reciprocalWave, squareWave, triangleWave, sinWave, fullRectifiedWave, halfRectifiedWave) begin
                dataOut = 8'b0;
                case(waveSelector)
                        3'b000: dataOut = reciprocalWave;
                        3'b001: dataOut = squareWave;
                        3'b010: dataOut = triangleWave;
                        3'b011: dataOut = sinWave;
                        3'b100: dataOut = fullRectifiedWave;
                        3'b101: dataOut = halfRectifiedWave;
                        3'b110: dataOut = romIn;
                        default: dataOut = 8'b0;
                endcase
        end
endmodule
```

Fig. 6 waveform generator Verilog code

The results of the sine and cosine operations are signed and range between -127 to +128. However, for simplification and compatibility with other parts of this experiment, an offset of 127 was added to make the range of our signal between 0 and 256.

Most modern function generators use Direct Digital Synthesis (DDS) to produce their output waveforms. DDS generates a tunable output with arbitrary phase from a fixed-frequency reference clock, such as an oscillator. The output of the DDS module is a quantized version of the output waveform, typically a sinusoid, controlled by a phase control value. To generate the DDS signal, a 1-port ROM memory is used to store the values of a sine wave for several clock cycles, which are initialized using a file named sine.mif. An address location is generated using a register and adder, which increments the phase of the signal by the value of the Phase_cntrl. This module is commonly known as a phase accumulator. The DDS module requires a stable clock frequency, which in this case is a 50-MHz clock frequency of the FPGA. However, the clock frequency will be changed to the ring oscillator divided clock in the subsequent sections of the design.

```
1   `timescale 1ns/1ns
2
3   module DDS(clk, rst, romSinWave);
4           input clk, rst;
5           output romSinWave;
6           reg [7:0] counter = 8'b0;
7           always @(posedge clk) begin
8                   if (rst) begin
9                           counter = 8'b0;
10                  end
11                  else begin
12                          counter = counter + 1;
13                  end
14          end
15          assign romSinWave = counter;
16  endmodule
17
```

Fig. 7 DDS Verilog code (without memory part)

We are going to use different method to implement memory for DDS module.
In first attempt we simply use the ROM megafunction, lpmrom, in the Quartus Megawizard Plug-In.

We can also write Verilog description of memory. By default the Quartus put our memory in logic elements(Fig. 11), but this part of FPGA is so important in some designs. So we have this option to forced Quartus to put that in memory part of FPGA(Fig. 13).

```
1   `timescale 1ns/1ns
2
3   module ROM3(clk, address, dOut);
4     input clk;
5     input [7:0] address;
6     output reg [7:0] dOut;
7     (* ram_init_file = "sine.mif" *) reg [7:0] rom [255:0];
8     always @(posedge clk) begin
9       dOut = rom[address];
10    end
11  endmodule
12
```

Fig. 8 Memory Verilog description

```
1   `timescale 1ns/1ns
2
3   module ROM3(clk, address, dOut);
4     input clk;
5     input [7:0] address;
6     output reg [7:0] dOut;
7     (* ROMSTYLE = "M4K" *)(* ram_init_file = "sine.mif" *) reg [7:0] rom [255:0];
8     always @(posedge clk) begin
9       dOut = rom[address];
10    end
11  endmodule
12
```

Fig. 9 Memory Verilog description (ROMSTYLE is M4K)

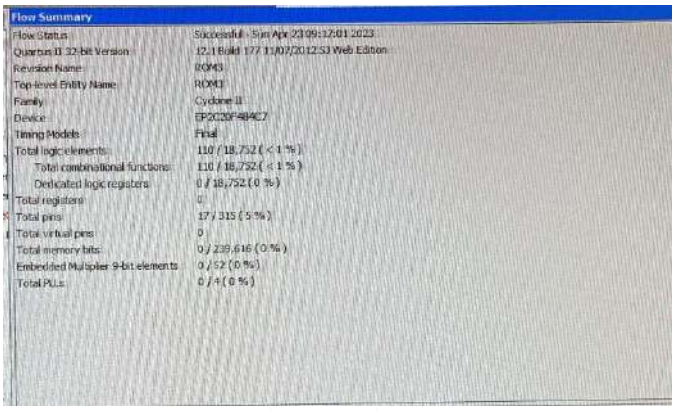

Fig. 10 Flow summary of syntesies of memory without memtoning ROMSTYLE: the total memory bits is zero which means that no memory block is used

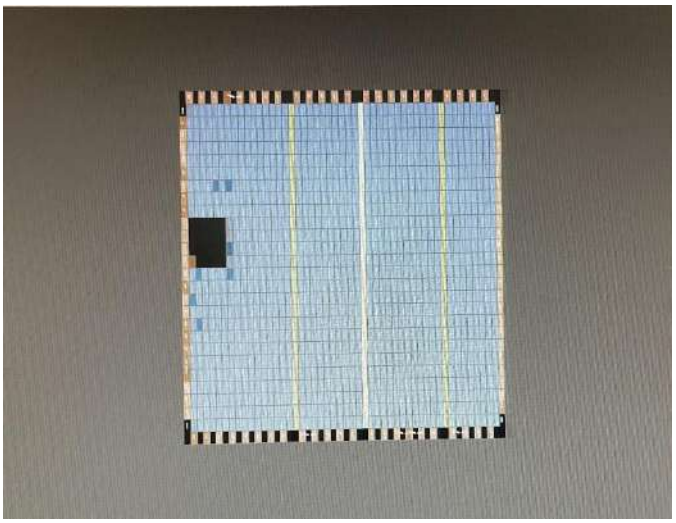

Fig. 11 Chip planner: Blue blocks are LAB and dark blue blocks show used LAB

Fig. 12 Flow summary of syntesies of memory with memtoning that ROMSTYLE is M4K: the total memory bits is 2048 which means that memory block is used. And the total logic element zero which mean no LAB is used.

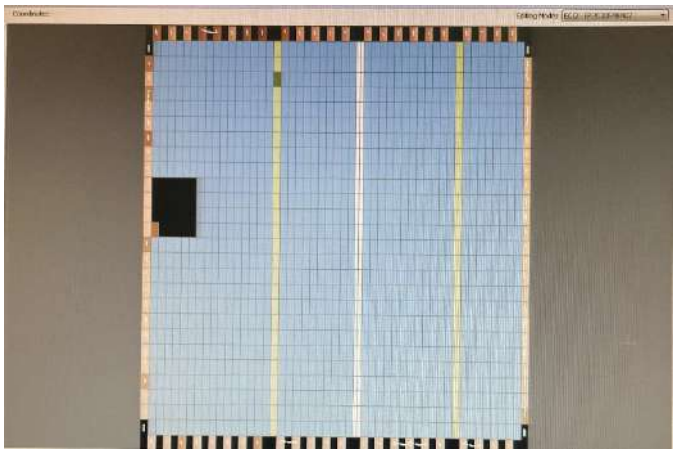

Fig. 13 Chip planner: green block show the used memory block

For verifying the functionality of our design, we will test our design in Modelsim.

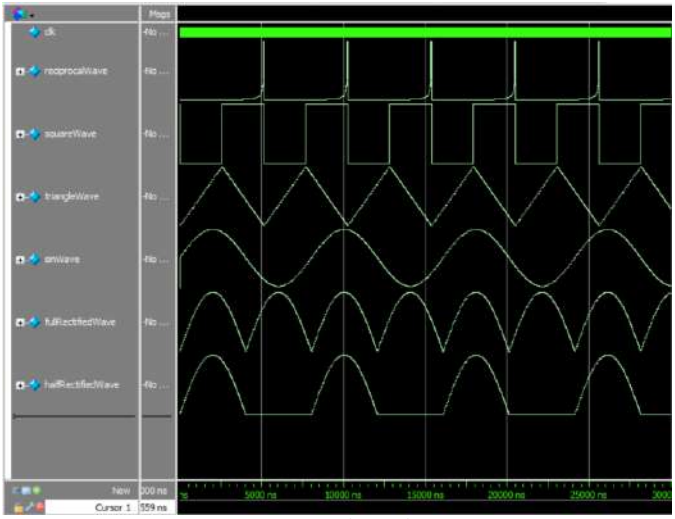

Fig. 14 waveform of signals

III. DIGITAL TO ANALOG CONVERSION USING PWM

A) Verilog code for DAC module is shown in Fig. 15.

```
1   `timescale 1ns/ 1ns
2
3   module PWM(clk, rst, data, PWM_out);
4           input clk, rst;
5           input [7:0] data;
6           output reg PWM_out;
7
8           reg [7:0] pulse_width = 8'b0;
9
10          always @(posedge clk) begin
11
12                  if (rst) begin
13                          pulse_width = 8'b0;
14                  end
15
16                  else begin
17
18                          if (pulse_width == 8'b0) begin
19                                  PWM_out = 1'b1;
20                          end
21
22                          if (pulse_width == data) begin
23                                  PWM_out = 1'b0;
24                          end
25
26                          pulse_width = pulse_width + 1;
27                  end
28          end
29  endmodule
30
```

Fig. 15 PMW Verilog code

B) The result of simulation of Modelsim is shown in Fig. 16.

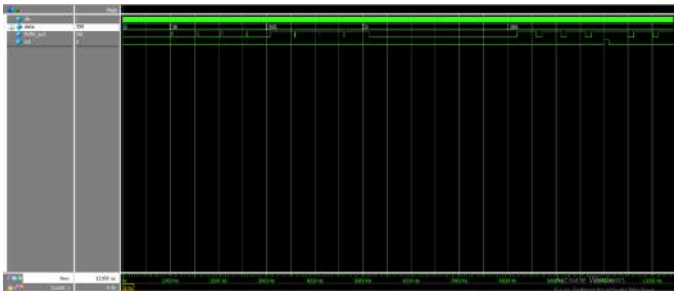

Fig. 16 PWM waveform

C) After synthesising the design and programming FPGA, we connected output of the FPGA to oscilloscope and receive these waves by testing it.

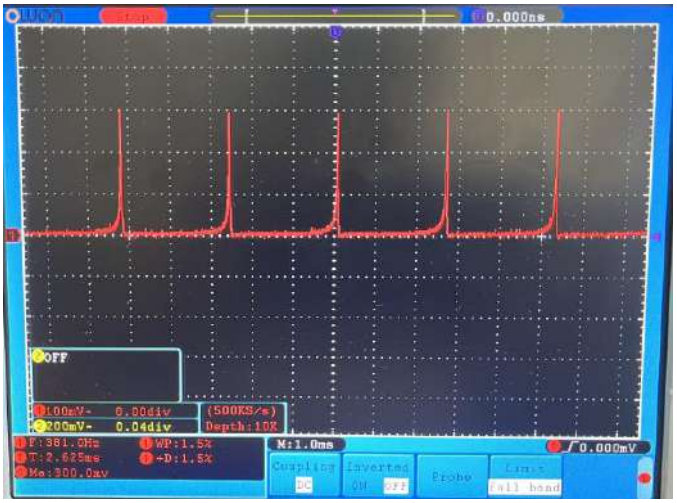

Fig. 17 Reciprocal waveform

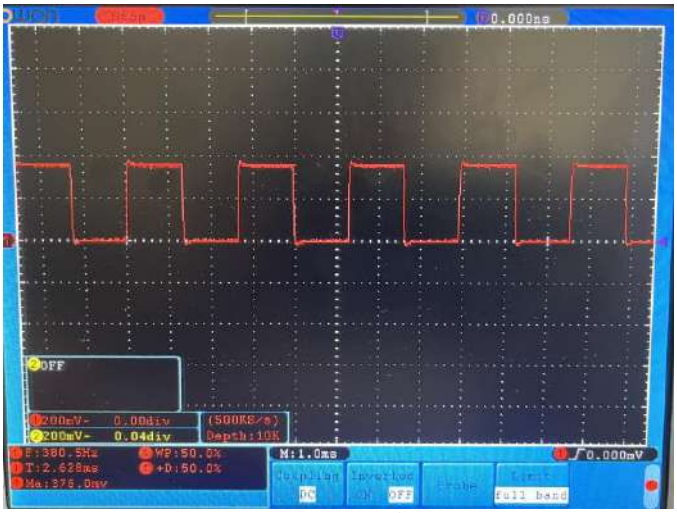

Fig. 18 Square waveform

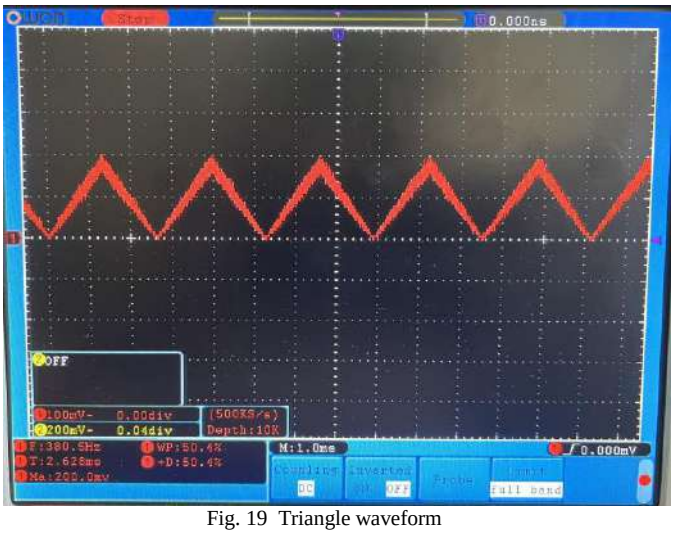

Fig. 19 Triangle waveform

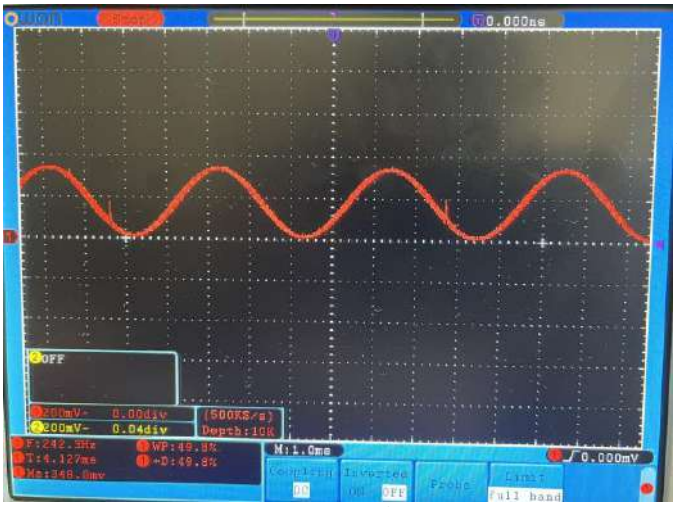

Fig. 20 Sine waveform

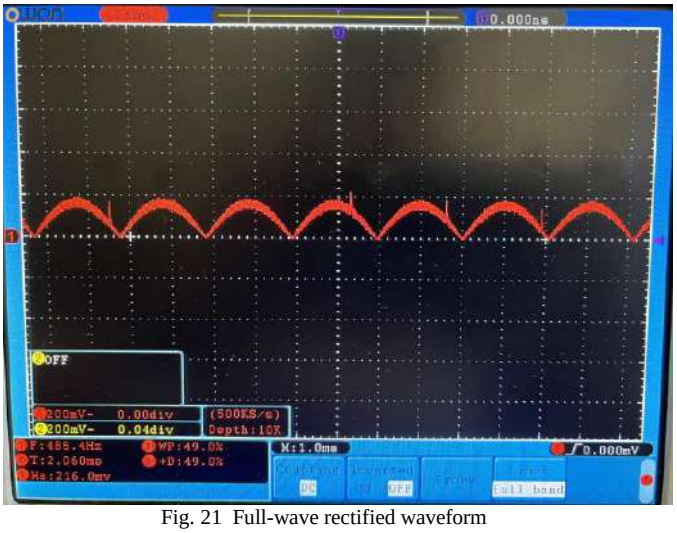

Fig. 21 Full-wave rectified waveform

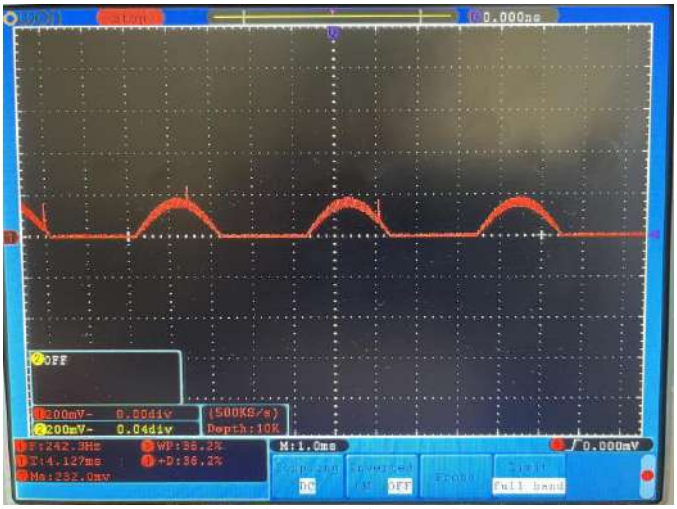

Fig. 22 Half-wave rectified waveform

IV. Frequency Selector

In order to set the frequency of the output signal a frequency selector is required. The frequency selector consists of a counter that divides a high source input signal to the desired value. For this purpose, we will write a Verilog description of a simple 9 bit counter like Fig. 23.

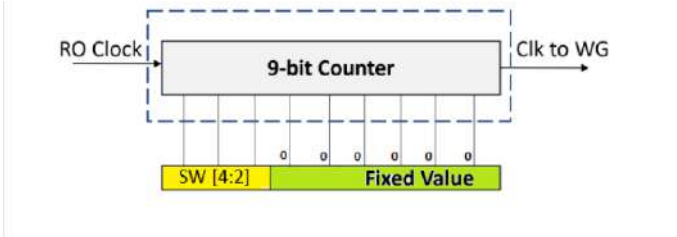

Fig. 23 Block diagram of frequency selector

1) We will turn the 9 bit counter to frequency divider by wiring carry out signa to load enable signal.

```
1   `timescale 1ns/1ns
2
3   module FrequencySelector(clk, rst, ld, parIn, clkOut);
4           input clk, rst, ld;
5           input [2:0] parIn;
6           output clkOut;
7           wire carryOut;
8           NineBitCounter counter(.clk(clk), .rst(rst), .ldEn(ld || carryOut),
9                                  .parIn({parIn, 6'b0}), .cOut(carryOut));
10          assign clkOut = carryOut;
11  endmodule
12
```

Fig. 24 frequency selector Verilog code

2) for testing the design, we show the sine wave in four different frequencies.

Each period of sine wave will be completed by 400 clock cycles.

So the frequency of sine wave can be calculated by following formula:

$$f_{sine} = \frac{f_{clk}}{400 \times (512 - PI - 1)}$$

Where PI is parallel input of frequency selector.

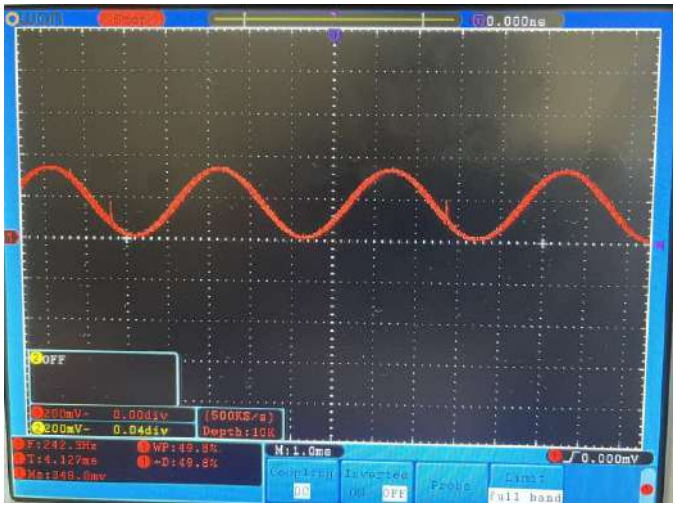

Fig. 25 Sine wave ($PI = 0\_0000\_0000_b$)

$$PI = 0_d$$

$$f_{sine} = \frac{50\ MHz}{400 \times (512 - 0 - 1)} = 244 \cdot 61\ Hz$$

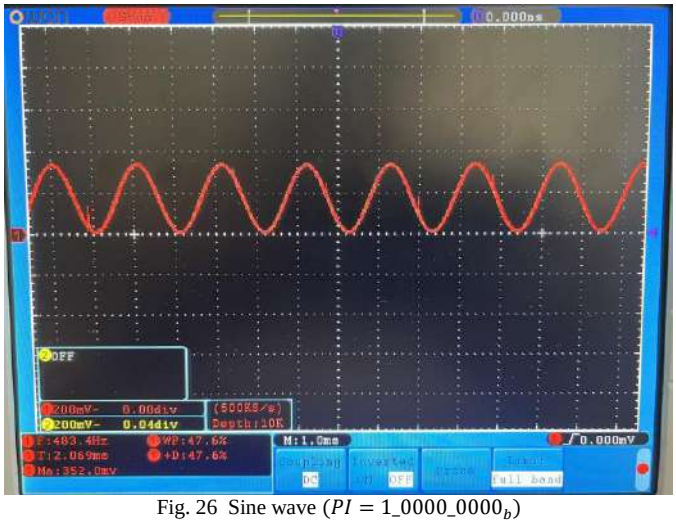

Fig. 26 Sine wave ($PI = 1\_0000\_0000_b$)

$$PI = 256_d$$

$$f_{sine} = \frac{50\ MHz}{400 \times (512 - 256 - 1)} = 490 \cdot 19\ Hz$$

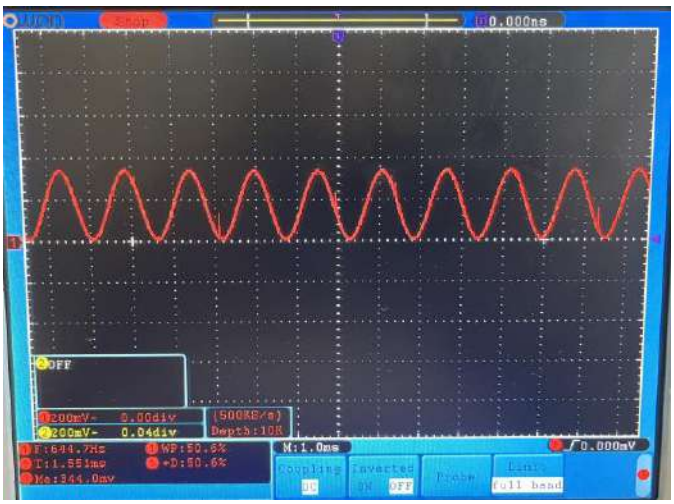

Fig. 27 Sine wave ($PI = 1\_0100\_0000_b$)

$$PI = 320_d$$

$$f_{sine} = \frac{50\ MHz}{400 \times (512 - 320 - 1)} = 654 \cdot 45\ Hz$$

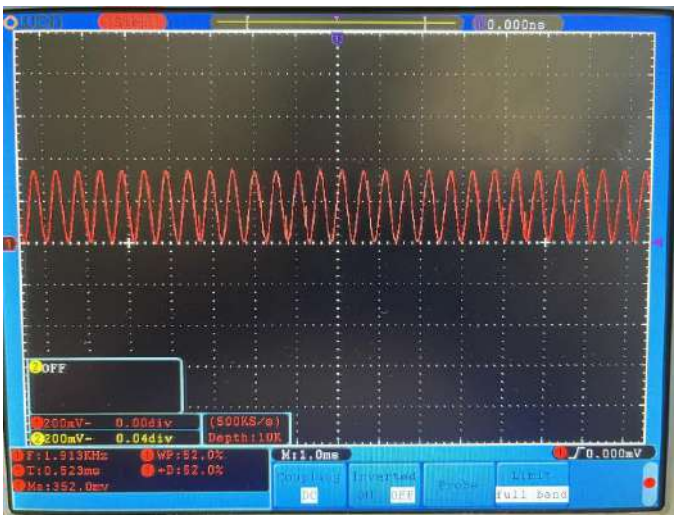

Fig. 28 Sine wave ($PI = 1\_1100\_0000_b$)

$$PI = 448_d$$

$$f_{sine} = \frac{50\ MHz}{400 \times (512 - 448 - 1)} = 1 \cdot 98\ KHz$$

Calculated frequencies are approximately equal to observed ones.

V. Amplitude Selector

One option in function generator is the amplitude of generated wave. The task of this module is to scale down the amplitude of the waveforms. This can be done by dividing the output amplitude by a number. The value of divisor is chosen by a 2-bit input. Dedicate two bits of input SW (SW[6:5]) to this selector inputs. This module divides the amplitude of the output wave by the numbers of following table.

Table 1: Amplitude selection

SW[6:5]	Amplitude
2'b00	1
2'b01	2
2'b10	4
2'b11	8

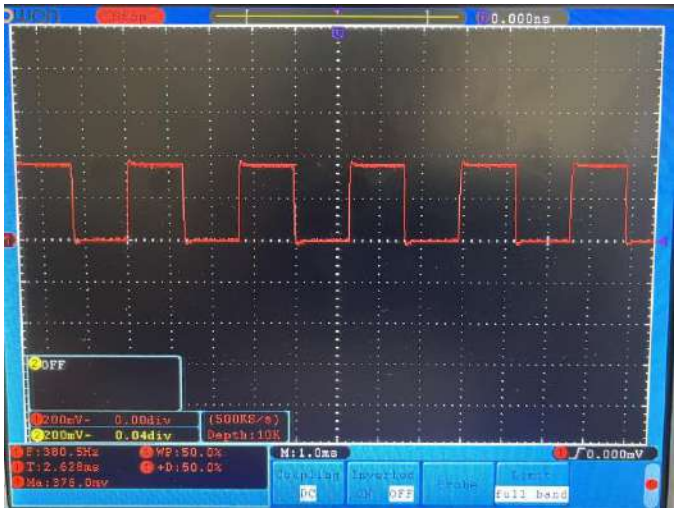

Fig. 29 Square wave (Amplitude: 1)

$$Domain = 360 \text{ mV}$$

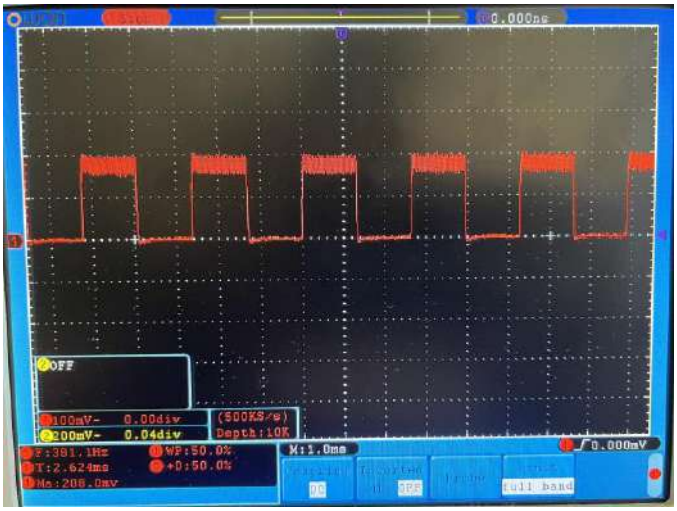

Fig. 30 Square wave (Amplitude: 2)

$$Domain = \frac{360}{2} = 180\ mV$$

Fig. 31 Square wave (Amplitude: 4)

$$Domain = \frac{360}{4} = 90\ mV$$

Fig. 32 Square wave (Amplitude: 8)

$$Domain = \frac{360}{8} = 40\ mV$$

VI. The Total Design

The final block diagram is shown if Fig. 33.

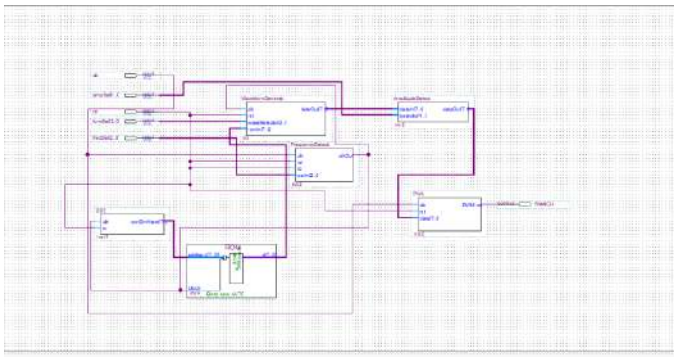
Fig. 33 Final block diagram in Quartus II

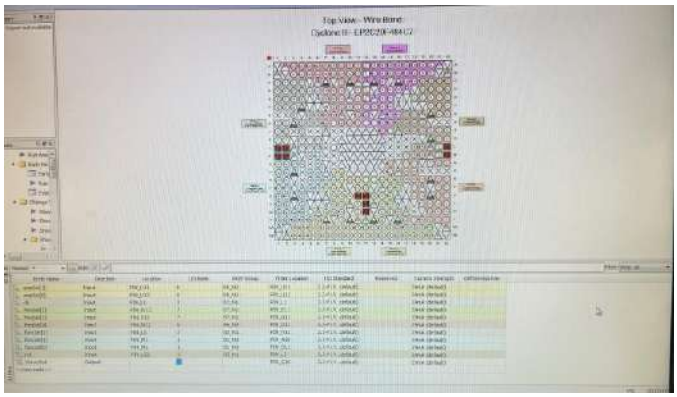
Fig. 34 Pin planner

VII. Conclusions

By designing the components of the waveform generator, frequency selector, amplitude selector, and DAC modules in Verilog, and synthesizing the project in Quartus II, we programmed the FPGA. The project successfully achieved its objectives of generating various waveforms with adjustable frequency and amplitude, and the final design was able to integrate all the modules seamlessly. We tested and verified the functionality of the design through the use of oscilloscope.

Acknowledgment

This report was prepared and developed by Mehdi Jamalkhah and Mobina Mehrazar, bachelor students of Computer engineering at University of Tehran, under the supervision of professor Zain Navabi.