Experiment 3 - Function Generator

Student Name : Shahzad Momayez , Mohammad Amanlou
Student ID : 810100272 , 810100084

***Abstract* — This document is a student report to experiment #3 of Digital Logic Laboratory course at ECE Department, University of Tehran. The goal of this experiment is to design an arbitrary function generator which is capable of generating many waveforms such as Rhomboid, Square, sine and etc with wide range for frequency selection**.

***Keywords*— Function Generator, Waveforms, Frequency Selector**

I. INTRODUCTION

As we mentioned before, we are to design a function generator. An overall view of the function generator system is showed in figure 1.

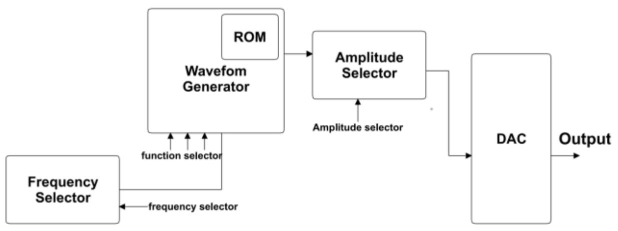


Figure 1: Block diagram of the Arbitrary Generator (AFG)

It consists of 3 main units:
1. Waveform Generator.
2. PWM
3. Frequency Selector.
4. Amplitude Selector

II. WAVEFORM GENERATOR

This component produces desired functions. Output of this module is an 8-bit digital representing the amplitude of signal. The supported functions are sine, square, reciprocal, triangle, full-wave and half-wave rectified signals.

Waveform generator inputs 3 bits indicating which function (waveform) needs to be generated by this unit. There are 7 choices:

Table 1: Function selection

`func[2:0]`	Function
`3'b000`	Reciprocal
`3'b001`	Square
`3'b010`	Triangle
`3'b011`	Sine
`3'b100`	Full-wave rectified
`3'b101`	Half-wave rectified
`3'b110`	DDS

We implemented waveforms square, reciprocal and triangle based on a counter that counts up or down with each clock for the period of the waveform. The output of the frequency selector is the input clock for this module that determines the discrete incremental values of this signal (resolution).

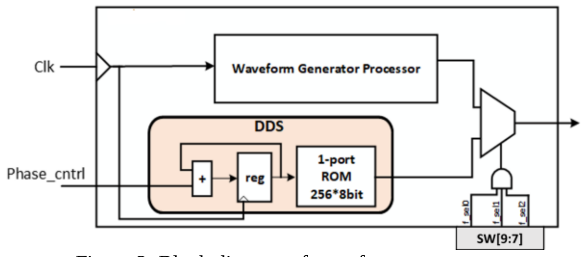


Figure 2: Block diagram of waveform generator

The waves that we have to generate are like below:

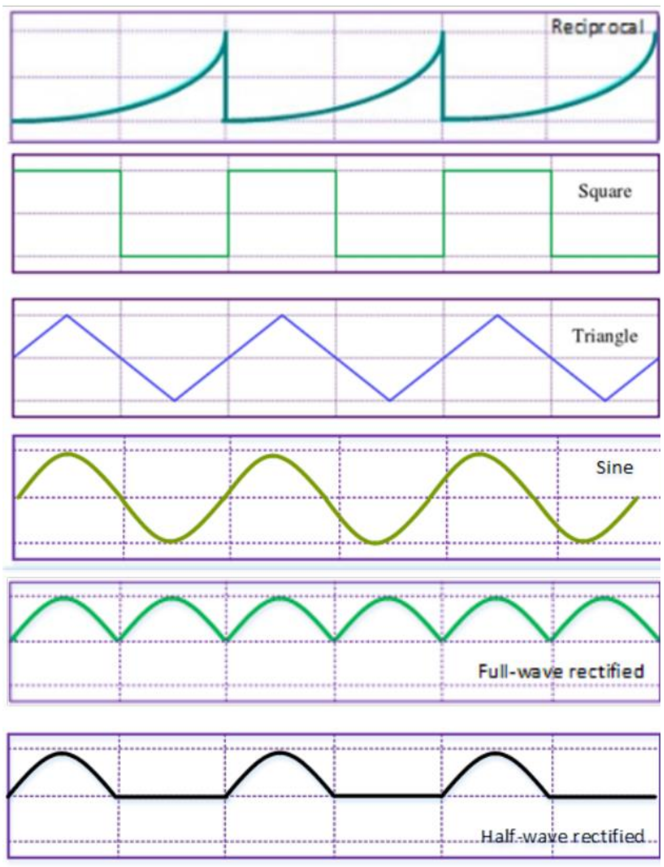
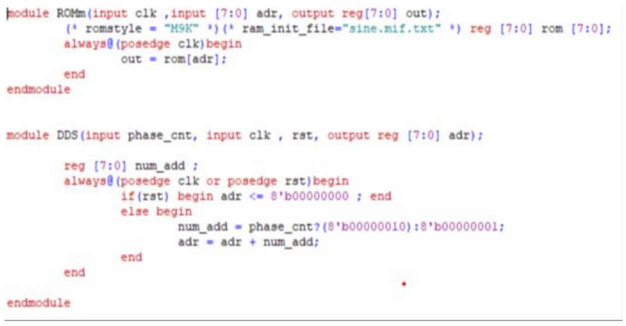


Figure 3: Different waveforms of function generator

The last case (3'b110) will generate a waveform stored in a ROM. The ROM is initialized using a sine.mif file with Quartus MegaWizard.

The fig below shows the Verilog of Rom and DDS:



```
module ROMm(input clk ,input [7:0] adr, output reg[7:0] out);
        (* romstyle = "M9K" *)(* ram_init_file="sine.mif.txt" *) reg [7:0] rom [7:0];
        always@(posedge clk)begin
                out = rom[adr];
        end
endmodule


module DDS(input phase_cnt, input clk , rst, output reg [7:0] adr);

        reg [7:0] num_add ;
        always@(posedge clk or posedge rst)begin
                if(rst) begin adr <= 8'b00000000 ; end
                else begin
                        num_add = phase_cnt?(8'b00000010):8'b00000001;
                        adr = adr + num_add;
                end
        end

endmodule
```

Fig4. Verilog of ROM and DDS

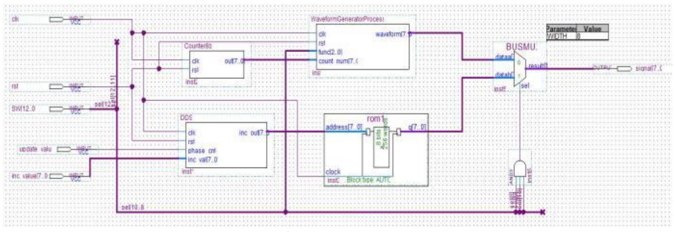


Fig 5. Block Diagram of WaveForm Generator unit

Processor unit in Fig. 5 outputs the desired function (waveforms) based on input SW[10..8]. 256 points are generated for each function. Each wave is represented by a function *f(x)* which *x* is the output of counters. An 8 bit counter is used for Processor unit and a DDS is used for the ROM since it stores 256 words (8 bit values). The multiplexer
puts the desired wave (generated by processor or ROM) on the
output. This output will later be used as an input for Amplitude Selector which we discuss later in this report.

In order to implement half-wave rectified and full-wave rectified , first we need to implement sine wave. And to implement sine wave we have some mathematic formulas that are usable in our Verilog code.

$\sin(n) = \sin(n-1) + a.\cos(n-1)$
$\cos(n) = \cos(n-1) - a.\sin(n)$
$\sin(n) = \sin(n-1) + 1/64 . \cos(n-1)$
$\cos(n) = \cos(n-1) - 1/64 .\sin(n)$

we also assume values are between -30000 to 30000 for sin and cos. Also:
sin(0) =0 , cos(0)= 30000

Initialization of first values in differential equations is necessary. Use 0 for sin(0) and 30000 for cos(0). The results of sine and cosine operations are signed and between -127 to +128. However, for simplification and compatibility with other parts of this experiment, we add an offset of 127, making the range of our signal between 0 and 256.

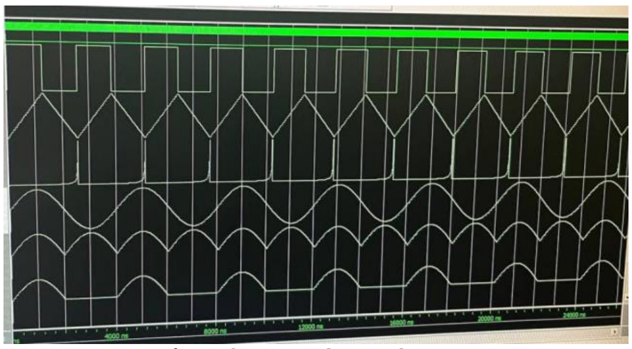
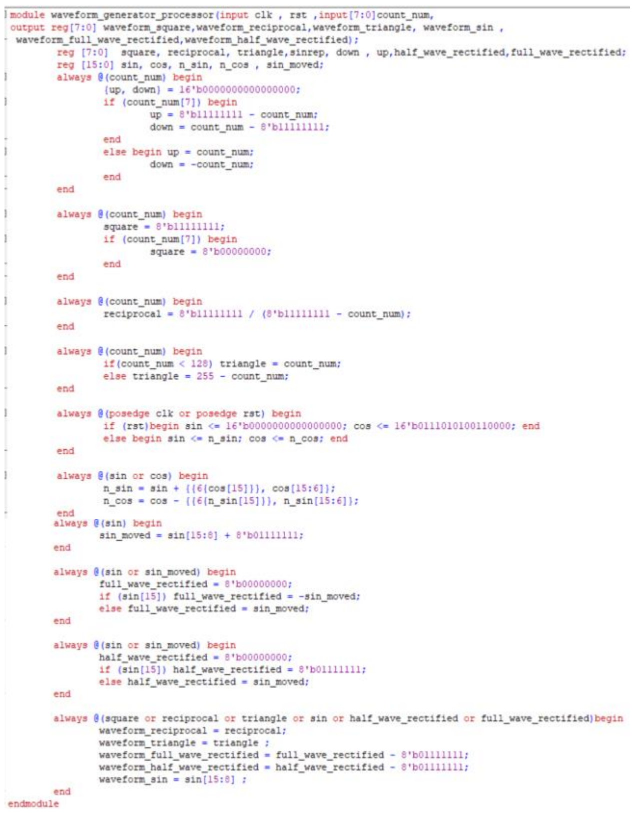
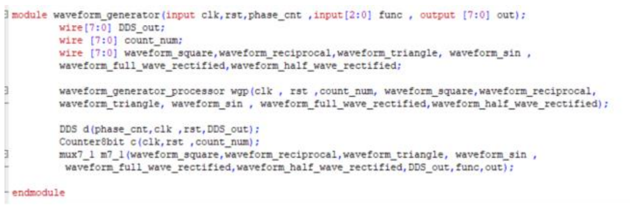


Fig6. Waveforms that we implemented.

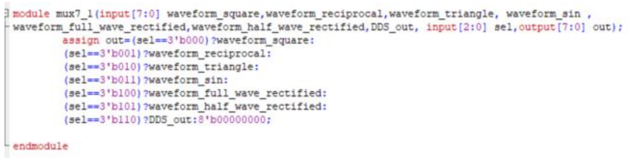
Below is the Verilog of function generator.



```verilog
module waveform_generator_processor(input clk , rst ,input[7:0]count_num,
output reg[7:0] waveform_square,waveform_reciprocal,waveform_triangle, waveform_sin ,
 waveform_full_wave_rectified,waveform_half_wave_rectified);
        reg [7:0]  square, reciprocal, triangle,sinrep, down , up,half_wave_rectified,full_wave_rectified;
        reg [15:0] sin, cos, n_sin, n_cos , sin_moved;
        always @(count_num) begin
                {up, down} = 16'b0000000000000000;
                if (count_num[7]) begin
                        up = 8'b11111111 - count_num;
                        down = count_num - 8'b11111111;
                end
                else begin up = count_num;
                        down = -count_num;
                end
        end

        always @(count_num) begin
                square = 8'b11111111;
                if (count_num[7]) begin
                        square = 8'b00000000;
                end
        end

        always @(count_num) begin
                reciprocal = 8'b11111111 / (8'b11111111 - count_num);
        end

        always @(count_num) begin
                if(count_num < 128) triangle = count_num;
                else triangle = 255 - count_num;
        end

        always @(posedge clk or posedge rst) begin
                if (rst)begin sin <= 16'b0000000000000000; cos <= 16'b0111010100110000; end
                else begin sin <= n_sin; cos <= n_cos; end
        end

        always @(sin or cos) begin
                n_sin = sin + {{6{cos[15]}}, cos[15:6]};
                n_cos = cos - {{6{n_sin[15]}}, n_sin[15:6]};
        end
       always @(sin) begin
               sin_moved = sin[15:8] + 8'b01111111;
       end

       always @(sin or sin_moved) begin
               full_wave_rectified = 8'b00000000;
               if (sin[15]) full_wave_rectified = -sin_moved;
               else full_wave_rectified = sin_moved;
       end

       always @(sin or sin_moved) begin
               half_wave_rectified = 8'b00000000;
               if (sin[15]) half_wave_rectified = 8'b01111111;
               else half_wave_rectified = sin_moved;
       end

       always @(square or reciprocal or triangle or sin or half_wave_rectified or full_wave_rectified)begin
               waveform_reciprocal = reciprocal;
               waveform_triangle = triangle ;
               waveform_full_wave_rectified = full_wave_rectified - 8'b01111111;
               waveform_half_wave_rectified = half_wave_rectified - 8'b01111111;
               waveform_sin = sin[15:8] ;
       end
endmodule
```



```verilog
module waveform_generator(input clk,rst,phase_cnt ,input[2:0] func , output [7:0] out);
        wire[7:0] DDS_out;
        wire [7:0] count_num;
        wire [7:0] waveform_square,waveform_reciprocal,waveform_triangle, waveform_sin ,
        waveform_full_wave_rectified,waveform_half_wave_rectified;

        waveform_generator_processor wgp(clk , rst ,count_num, waveform_square,waveform_reciprocal,
        waveform_triangle, waveform_sin , waveform_full_wave_rectified,waveform_half_wave_rectified);

        DDS d(phase_cnt,clk ,rst,DDS_out);
        Counter8bit c(clk,rst ,count_num);
        mux7_1 m7_1(waveform_square,waveform_reciprocal,waveform_triangle, waveform_sin ,
         waveform_full_wave_rectified,waveform_half_wave_rectified,DDS_out,func,out);

endmodule
```



```verilog
module mux7_1(input[7:0] waveform_square,waveform_reciprocal,waveform_triangle, waveform_sin ,
waveform_full_wave_rectified,waveform_half_wave_rectified,DDS_out, input[2:0] sel,output[7:0] out);
        assign out=(sel==3'b000)?waveform_square:
        (sel==3'b001)?waveform_reciprocal:
        (sel==3'b010)?waveform_triangle:
        (sel==3'b011)?waveform_sin:
        (sel==3'b100)?waveform_full_wave_rectified:
        (sel==3'b101)?waveform_half_wave_rectified:
        (sel==3'b110)?DDS_out:8'b00000000;

endmodule
```

III. Digital to Analog conversion using PWM

PWM is abbreviation for Pulse Width Modulation . Figure 6 shows the DAC circuit. As can be seen for realizing an external DAC chip a serial to parallel interface is required between the FPGA board and the chip. In this experiment, we used the this method to have a digital to analog conversion. The following is a brief description of how PWM works as a DAC

A PWM signal is a sequence of periods in which the duration of the logic-high (or logic-low) voltage varies according to external conditions, and these variations can be used to transmit information. The PWM carrier frequency is constant, so the active and inactive state duration increase.

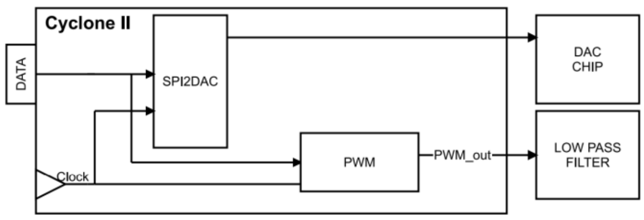


Figure 7. Block diagram of Digital to Analog Converter

In brief, PWM converts a 8-bit digital number to a 1-bit digital number. From its input it receives a number. If this input number was greater than “N” the output becomes 1 otherwise it becomes 0.

Figure below shows the clock cycle of PWM:

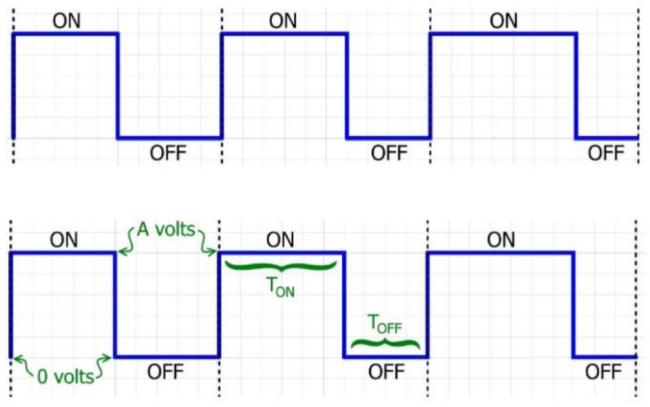
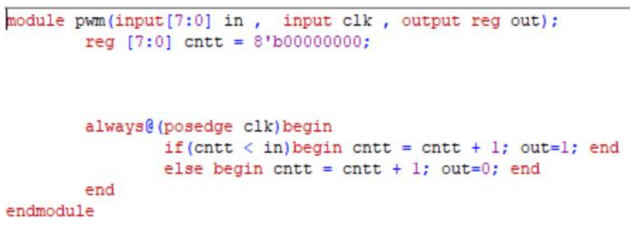


Fig8. Clock cycle of PWM

or decrease vice versa. The duty cycle of the PWM signal is equal to: duty cycle = $T_{on} / (T_{on} + T_{off})$

The relationship between duty cycle, amplitude, and nominal DAC voltage is fairly intuitive: In the frequency domain, a low-pass filter suppresses higher frequency components of an input signal. The time-domain equivalent of this effect is smoothing, or averaging. Thus, by low-pass filtering a PWM signal we are extracting its average value. In this experiment, period of PWM is fixed to 256 clocks and its pulse width is the value on 8-bit input of module. Figure 7 shows sample PWM waves. In each cycle, first PW clock output value is 1 and it is 0 for rest of cycle. The output of the PWM module will go to an external board with an RC low pass filter.



```
module pwm(input[7:0] in ,  input clk , output reg out);
        reg [7:0] cntt = 8'b00000000;



        always@(posedge clk)begin
                if(cntt < in)begin cntt = cntt + 1; out=1; end
                else begin cntt = cntt + 1; out=0; end
        end
endmodule
```

IV. FREQUENCY SELECTOR

The scheme below is a general scheme of frequency selector:

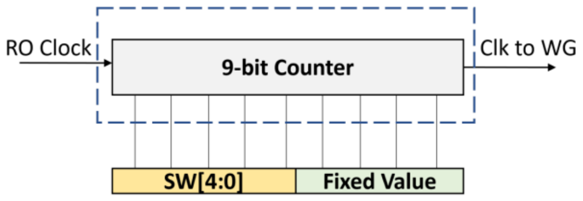
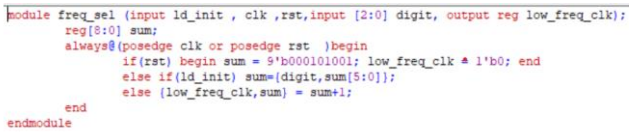


Fig9. Frequency selector

This section divides the FPGA clock (50MHz) and the divided lock is used for the WaveGenerator unit clock. The division is done by building a 256-counter with parallel load
inputs. Counting starts from PL[7..0] and ends at 255 = $(11111111)_2$. When the counter reaches 255, the carry out bar
bit of a 74193 IC becomes high, therefore there is a toggle on
the T Flip Flop (Built using a 7476 JK Flip Flop). The output
of the T Flip Flop is the divided clock. The divided clock has
a duty cycle of 50%. It toggles whenever the counter reaches
255, therefore $T_{\text{divided\_clock}} = T_{\text{FPGA}} \times (255 - PL[7..0])$ where T
indicates the period of a clock signal. The desired frequency
can be set using the suitable parallel load values using the formula above.



```
module freq_sel (input ld_init , clk ,rst,input [2:0] digit, output reg low_freq_clk);
        reg[8:0] sum;
        always@(posedge clk or posedge rst  )begin
                if(rst) begin sum = 9'b000101001; low_freq_clk = 1'b0; end
                else if(ld_init) sum={digit,sum[5:0]};
                else {low_freq_clk,sum} = sum+1;
        end
endmodule
```

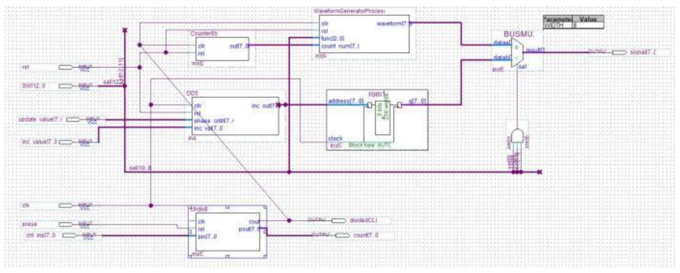
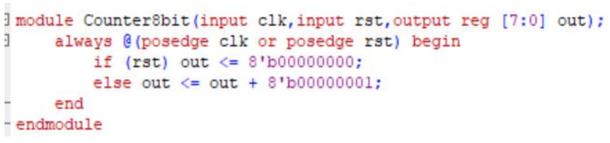


Fig. 10: Block Diagram of Wave Generator with a Frequency Selector (Clock divider)



```
module Counter8bit(input clk,input rst,output reg [7:0] out);
    always @(posedge clk or posedge rst) begin
        if (rst) out <= 8'b00000000;
        else out <= out + 8'b00000001;
    end
endmodule
```

The unit with least parallel load (240) will have the highest clock period (It takes longer for the counters to reach 255 and for the T Flip Flop to toggle), therefore less number of positive edges in a fixed length of time leading to a lower frequency wave. The unit with highest parallel load (250) will have the lowest clock period, therefore higher number of positive edges in a fixed length of time leading to a higher frequency wave. This is can be validated in Figure 12.

$T_{\text{divided\_clock}} = \text{TFPGA} \times (255 - PL[7..0])$

$T_{\text{FPGA}} = 1 / f_{\text{FPGA}} = 1 / 50\text{MHz} = 20\text{ns}$

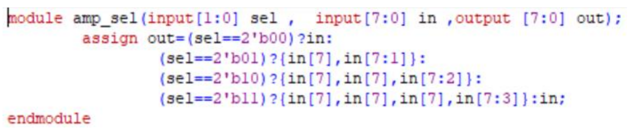
V. AMPLITUDE SELECTOR

This is an option in the Function Generator to scale down the generated waveform. The 8 bit output of the WaveForm Generator unit is passed to this unit and scaling is executed based on a 2 bit input indicating the scaling intensity. Based on the 2 bit input, the output is divided by a number shown in the following table:

Table 2: Amplitude selection

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

Dividing each waveform by 2^i is equivalent to shifting i units to the right, therefore sign extension needs to be applied to each signal.



```
module amp_sel(input[1:0] sel ,  input[7:0] in ,output [7:0] out);
        assign out=(sel==2'b00)?in:
                (sel==2'b01)?{in[7],in[7:1]}:
                (sel==2'b10)?{in[7],in[7],in[7:2]}:
                (sel==2'b11)?{in[7],in[7],in[7],in[7:3]}:in;
endmodule
```

A block symbol of the Amplitude Selector section is added to
the previous block diagrams. The final (complete) block diagram is as followed:

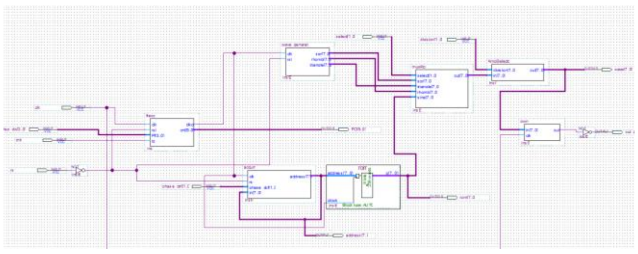
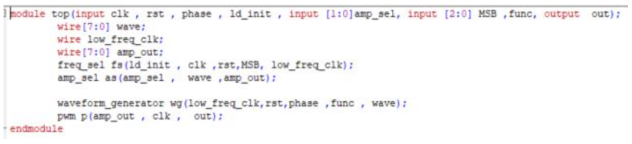
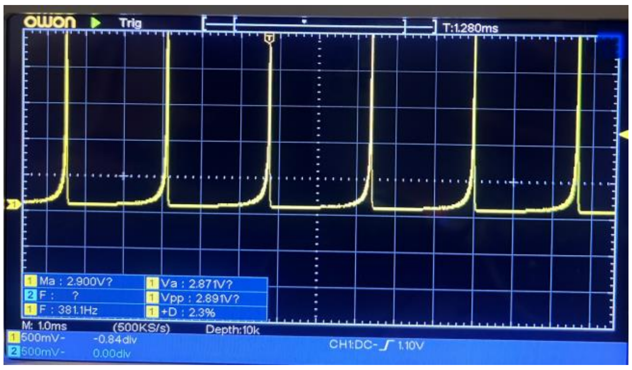


Fig. 11: Complete Block Diagram of Function Generator.

The final module has 13 bit inputs (SW[12..0]). 3 bits are used for function selection. 2 bits are used to amplitude selection.
The remaining 8 bits are used for parallel load of counters.



```
module top(input clk , rst , phase , ld_init , input [1:0]amp_sel, input [2:0] MSB ,func, output  out);
        wire[7:0] wave;
        wire low_freq_clk;
        wire[7:0] amp_out;
        freq_sel fs(ld_init , clk ,rst,MSB, low_freq_clk);
        amp_sel as(amp_sel ,  wave ,amp_out);

        waveform_generator wg(low_freq_clk,rst,phase ,func , wave);
        pwm p(amp_out , clk ,  out);
endmodule
```



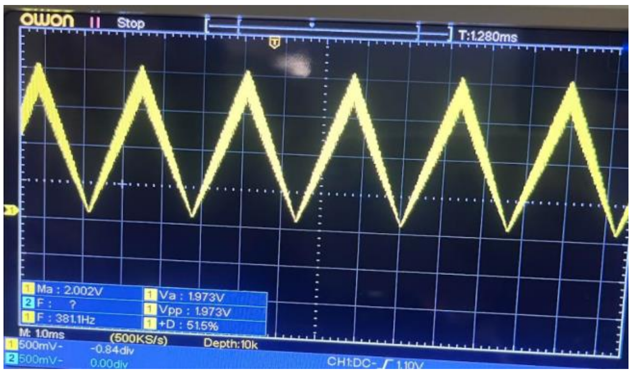


Fig12. triangle waveform

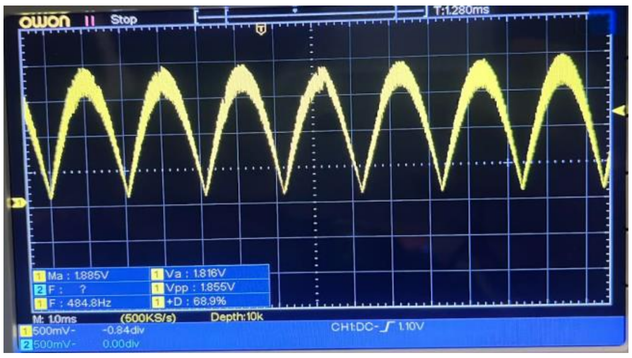


Fig13. Full wave rectified waveform

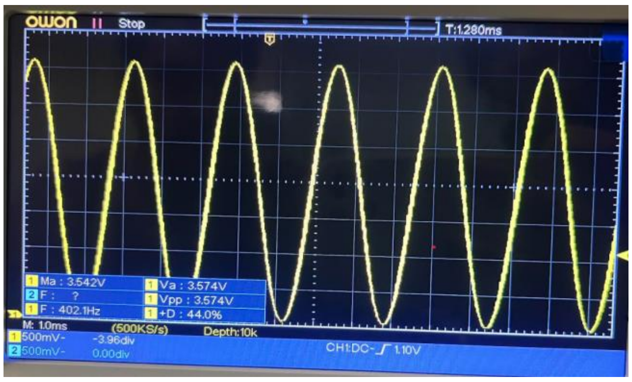


Fig14 . sine wave foem

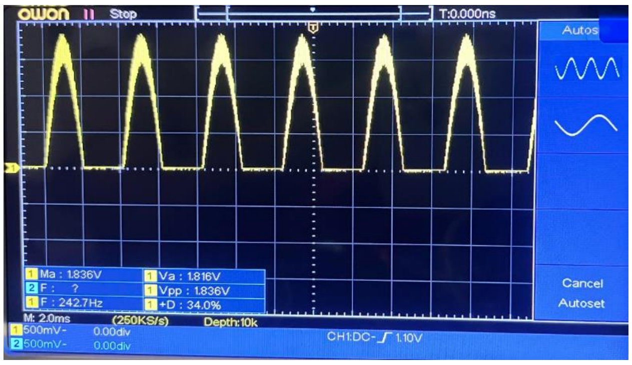


Fig 15. Half-wave rectified wave

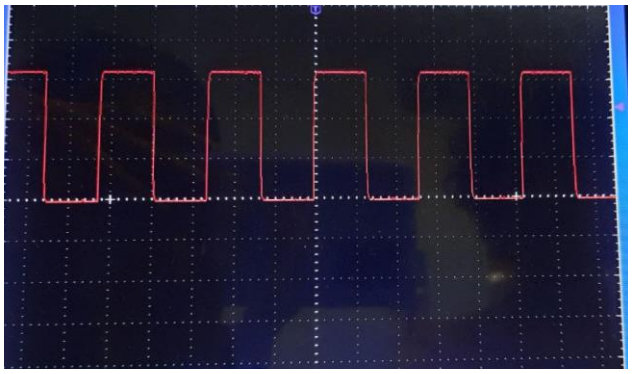


Fig 16. Square waveform

VI.CONCLUSIONS

All in all, we implementes these waves with Modelsim at first and then we moved our code to Quartus and we got the result. By the end of this experiment we learnt to design a Function Generator.

ACKNOWLEDGMENT

This report was prepared and developed by Shahzad Momayez(SID:810100272) and Mohammad Amanlou(SID:810100084) ,bachelor students of Computer engineering at University of Tehran, under the supervision of Professor Zain Navabi.