Experiment 3 - Function Generator

Erfan Mokhtari 810100208
AmirHosein Samoudi 810100108

***Abstract*— In this experiment we have designed an Arbitrary Function Generator (AFG), an electronic test instrument that generates reciprocal, square, triangular, sine, full-waved rectified and full-waved rectified waves with different amplitudes and frequencies.**

***Keywords*— Arbitrary Function Generator (AFG) – Waveform generator - Digital to Analog Converter (DAC) – PWM – ROM – DDS - ModelSim simulation - FPGA**

I. Introduction

In this experiment, we have designed an Arbitrary Function Generator (AFG), to generate varied signals with different values for amplitude and frequency. Based on these specifications there is a main component that generates one of the desired waveforms based on the function selectors' value (Waveform generator), a frequency selector that sets the output signal frequency, and an amplitude selector and DAC module that is used to convert the digital output to an analog signal.

II. Waveform Generator

This module produces desired functions. The output of this module is an 8-bit digital representing the amplitude of the signal. The supported functions, shown in figure 1, are sine, square, reciprocal, triangle, full-wave, and half-wave rectified signals.

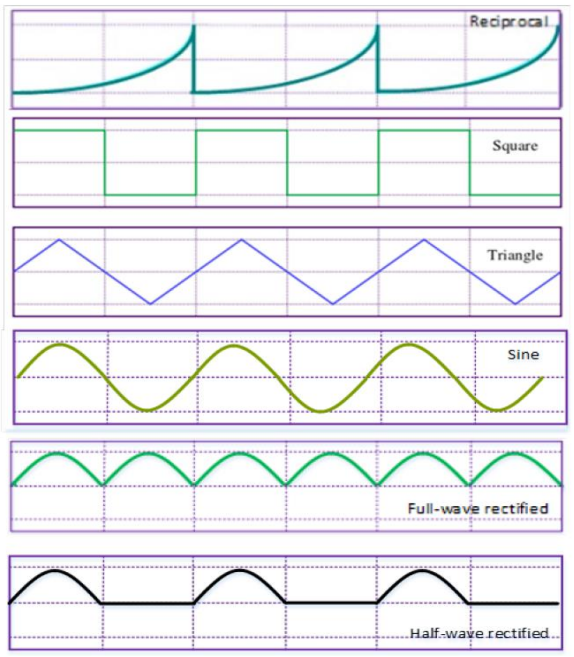


Fig 1. Different waveforms

Waveforms square, reciprocal, and triangle are 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.

Direct Digital Synthesis (DDS) module is used for generating arbitrary phase tunable output from a single fixed-frequency reference clock. The output of the DDS module is a quantized version of the output files (usually a sinusoid). To generate the DDS signal, we have used a 1-port ROM memory (with 64 bytes of data) to store the value of a sine wave for several clock cycles.

A phase accumulator module, PA, can generate the address location of this memory. The PA will enumerate all the data point positions of the entire period of the waveform. Two bits of the PA output denoted as "signbit" and "phasepos", are used to indicate the quadrant information. The next 6 bits of the PA output, denoted as "addr", are fed to the ROM address input.

Waveform_Generator.v

```
module Waveform_Generator(input clk, rst, input [2:0] sel, input [8:0] sign_DDS, output reg [7:0] out_signal);

    reg [7:0] counter_out;
    reg [7:0] square , reciprocal , triangle, sine, full_wave, half_wave;


    always @(posedge clk, posedge rst) begin
        if (rst) counter_out <= 8'b0;
        else counter_out <= counter_out + 1'b1;
    end

    always @(posedge clk) begin
        if(counter_out < 128)
            square <= 8'b0;
        else
            square <= 8'b11111111;

    end

    always @(posedge clk) begin
        if(counter_out < 128)
            triangle <= counter_out << 1;
        else
            triangle <= 255 - (counter_out <<1);
    end

    always @(posedge clk) begin

        reciprocal <= (255)/(255-counter_out);
    end

    always @(posedge clk) begin   ///Sine

        sine <= sign_DDS[7:0];
    end

    always @(posedge clk) begin  ///full_wave

        if (sign_DDS < 128 )
            full_wave <= ~sign_DDS[7:0] + 1 +255;
        else
            full_wave <= sign_DDS[7:0];
    end

    always @(posedge clk) begin  ///half_wave

        if (sign_DDS < 128 )
            half_wave <= 8'b01111111;
        else
            half_wave <= sign_DDS[7:0];
    end


    always @(sel, reciprocal, triangle, square, sine, full_wave, half_wave) begin
        out_signal <= 8'b0;
        case (sel)
        3'b000: out_signal <= reciprocal;
        3'b001: out_signal <= triangle;
        3'b010: out_signal <= square;
        3'b011: out_signal <= sine;
        3'b100: out_signal <= full_wave;
        3'b101: out_signal <= half_wave;
        default: out_signal <= 8'b0;
    endcase


    end

endmodule
```

Fig 2. Waveform generator Verilog code

```
module DDS(input clk, rst, output reg [8:0] sign_DDS);

    reg [7:0] mem_out;
     wire [7:0] mag;

    wire sign_bit,Phase_pos;
    wire[5:0] Addr;
    Phase_Accumulator_TOP PA(clk, rst, sign_bit, Phase_pos, Addr);


    wire [5:0] mem_in;
    assign mem_in = (Phase_pos) ? (~Addr +1):(Addr);

    //reg [7:0] LUT [0:63]; //ROM Memory
    //initial begin
    //$readmemb("sine.mem", LUT);
    // end
     (* romstyle = "M9K" *) (* ram_init_file = "sine.mif" *) reg [7:0] LUT [0:63];

    always @(mem_in) begin
        mem_out <= LUT[mem_in];
    end

    wire sel_mag;
    assign sel_mag = Phase_pos &(~|Addr);
    assign mag = (sel_mag)? (8'b11111111):(mem_out);




    always @(sign_bit, mag) begin
        if(~sign_bit)
            sign_DDS = {sign_bit, mag};
        else
            sign_DDS = ~mag +1 +255;
    end
endmodule
```

Fig 3. DDS module Verilog code

```
module Phase_Accumulator_DP(input clk, rst, output co, output reg[5:0] Addr);

    always @(posedge clk, posedge rst) begin
       if (rst) Addr <= 6'b0;
       else Addr <= Addr + 1'b1;
    end

    assign co = &Addr;

endmodule
```

Fig 4. Phase Accumulator Data path Verilog code

```
module Phase_Accumulator_CU(input clk , rst, co , output reg sign_bit , Phase_pos);

    reg [1:0] ps, ns;
    parameter [1:0] quarter1 = 0, quarter2 = 1, quarter3 = 2, quarter4 = 3;

    always @(ps, co) begin

        case (ps)
           quarter1 : ns = co ? quarter2 : quarter1;
           quarter2 : ns = co ? quarter3 : quarter2;
           quarter3 : ns = co ? quarter4 : quarter3;
           quarter4 : ns = co ? quarter1 : quarter4;
            default: ns = quarter1;
        endcase
    end

    always @(ps) begin
        {sign_bit, Phase_pos} = 2'b0;
        case (ps)
           quarter1 : {sign_bit, Phase_pos} = 2'b0;
           quarter2 : {sign_bit, Phase_pos} = 2'b01;
           quarter3 : {sign_bit, Phase_pos} = 2'b10;
           quarter4 : {sign_bit, Phase_pos} = 2'b11;
        endcase

    end

    always@(posedge clk , posedge rst) begin
        if (rst) ps <= quarter1;
        else ps <= ns;
    end
endmodule
```

Fig 5. Phase Accumulator controller Verilog code

```
module Phase_Accumulator_TOP(input clk , rst , output sign_bit , Phase_pos , output [5:0]Addr);

    wire co;

    Phase_Accumulator_DP DP(clk, rst, co, Addr);
    Phase_Accumulator_CU CU(clk , rst, co , sign_bit , Phase_pos);

endmodule
```

Fig 6. Phase Accumulator Top module Verilog code

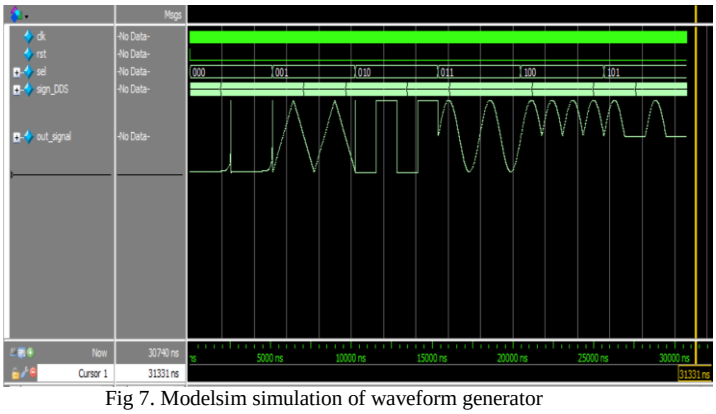


Fig 7. Modelsim simulation of waveform generator

III. Digital to Analog Conversion using PWM

In the following parts we will explain every component in the RTL design. For converting digital to analog (DAC) we use Pulse Width Modulation (PWM).

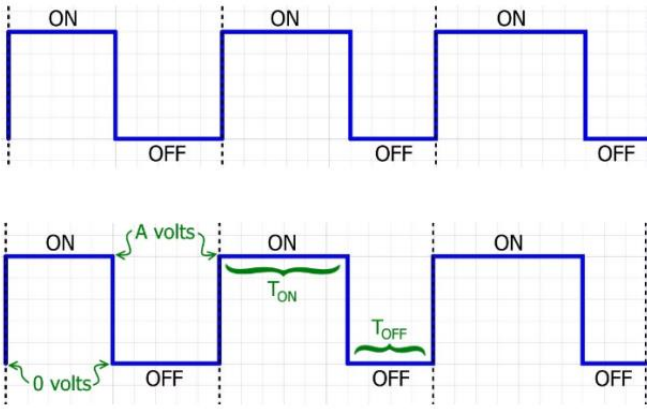
A PWM signal is a sequence of periods in which the duration of the logic-high (or logic-low) voltage varies according to external conditions, and these variations can be used to transmit information.

The duty cycle of the PWM signal is equal to:

$$duty\ cycle = \frac{T_{on}}{T_{on} + T_{off}}$$

Fig 8. Duty cycle of PWM

Figure 9: Pulse Width Modulation (PWM)



The nominal DAC voltage observed at the output of the low-pass filter is determined by just two parameters, namely, the duty cycle and the PWM signal's logic-high voltage; in the figure 9, A denotes this logic-high voltage for "amplitude." 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.
When the input signal is larger than counter the output is 1 and when the input signal is less than counter the output is 0.

```
module PWM(input clk, rst, input [7:0]
In_PWM, output Out_PWM);
    reg [7:0] count;
    always@(posedge clk, posedge rst) begin
        if(rst) count <= 8'b0;
        else count <= count + 1'b1;

    end
    assign Out_PWM = (count <= In_PWM) ?
1'b1 : 1'b0;

endmodule
```

Fig 10. PWM module

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.

```
module Freq_Selector(input clk , rst ,
key_0 , input [4:0] load , output co);

    wire ld ;
    assign ld = key_0 | co ;

    reg [8:0] count;
    always @(posedge clk , posedge rst)
begin
        if (rst) count <= 9'b0;
        else begin
            if (ld) count <= {load,
4'b0110};
            else count <= count +1;
        end
    end
    assign co = &count;
endmodule
```

Fig 11. Frequency selector module

V. AMPLITUDE SELECTOR

This module it uses to dividing the output amplitude by a number that is chosen by 2-bit input (SW[6:5]).

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

Fig 12. Amplitude selection

```
module Amp_Sel(input [7:0] In_signal, input
[1:0] sel, output reg [7:0] Out_signal);

    always @(sel, In_signal) begin
        //Out_signal = 8'b0;
        case (sel)
           2'b00 : Out_signal <= In_signal;
           2'b01 : Out_signal <= In_signal
>> 1;
           2'b10 : Out_signal <= In_signal
>> 2;
           2'b11 : Out_signal <= In_signal
>> 3;
        endcase

    end

endmodule
```

Fig 13. Amplitude selector module

IV. The Total design

Now we should in Quartus connecting modules then we assign ports of the FPGA using pin planner. Finally we connect our circuit to the function and observe the output on the screen.

We start with square signal and increase amplitude gradually then change the frequency :

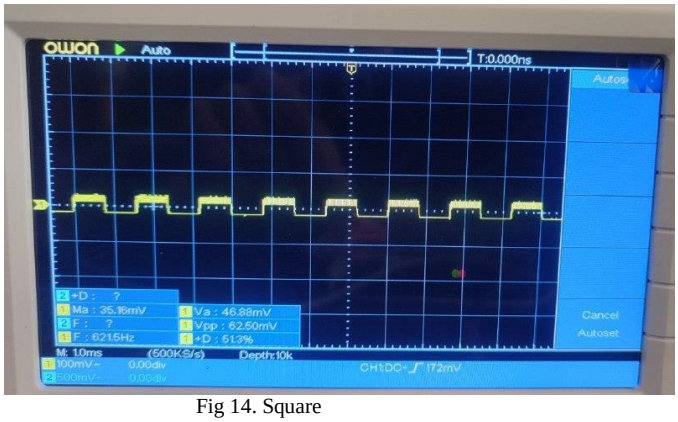


Fig 14. Square

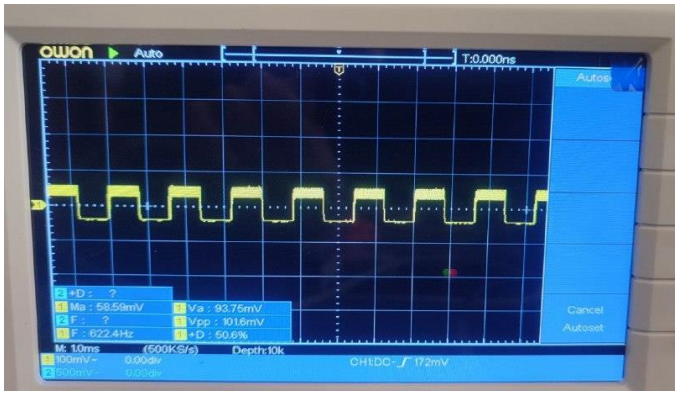


Fig 15. Square (increase amplitude)

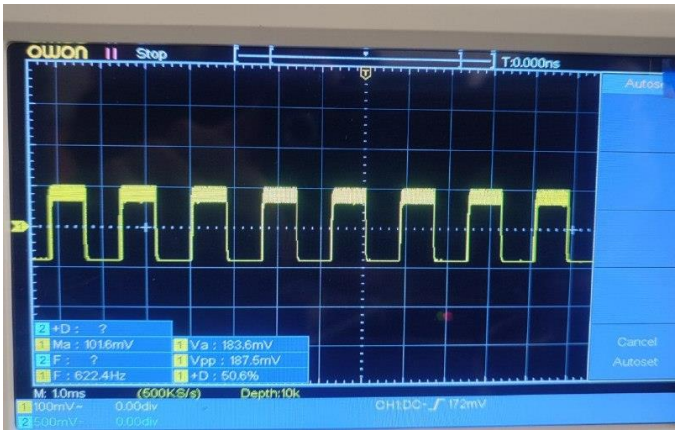


Fig 16. Square (increase amplitude)

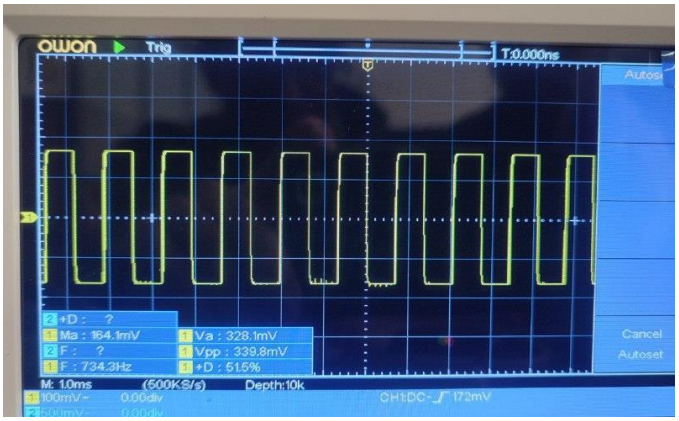


Fig 17. Square (increase frequency)

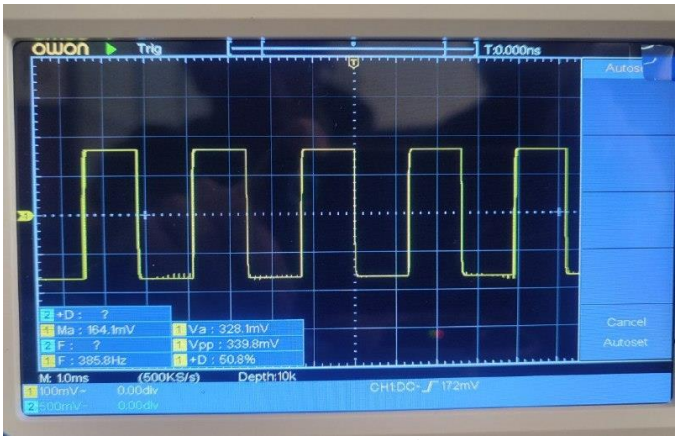


Fig 18. Square (decrease frequency)

We continue with reciprocal :

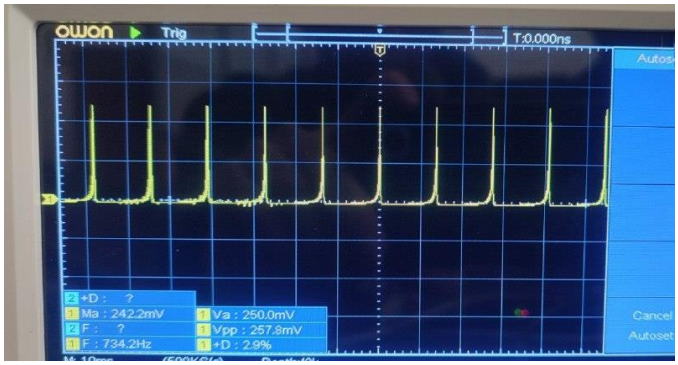


Fig 19. reciprocal

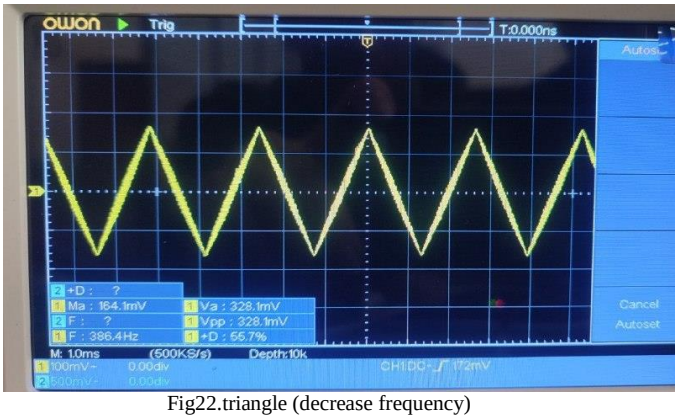


Fig22.triangle (decrease frequency)

And triangle :

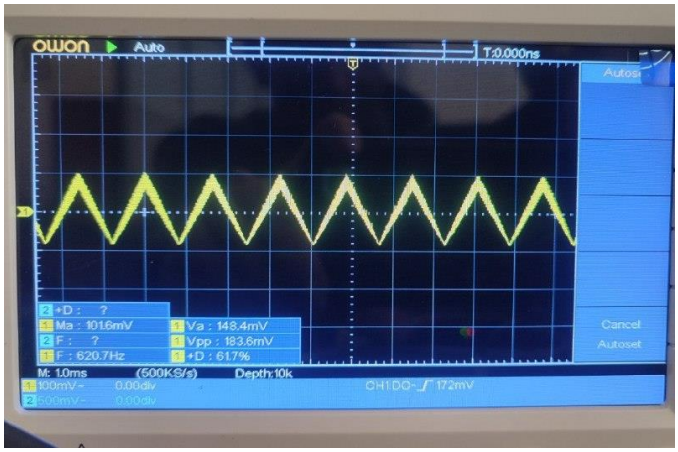


Fig 20. triangle

The next one is sine :

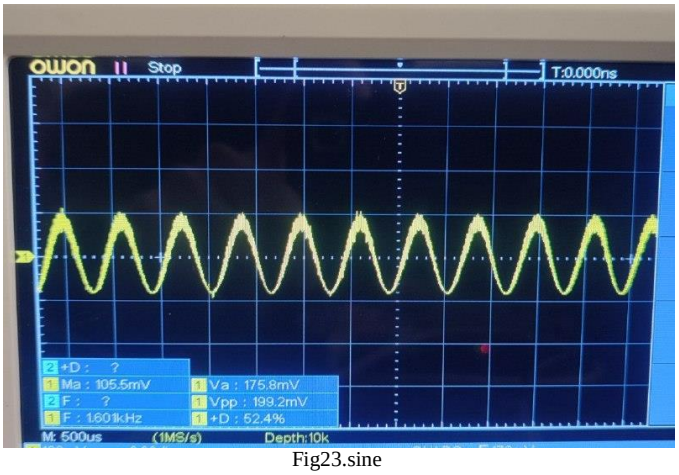


Fig23.sine

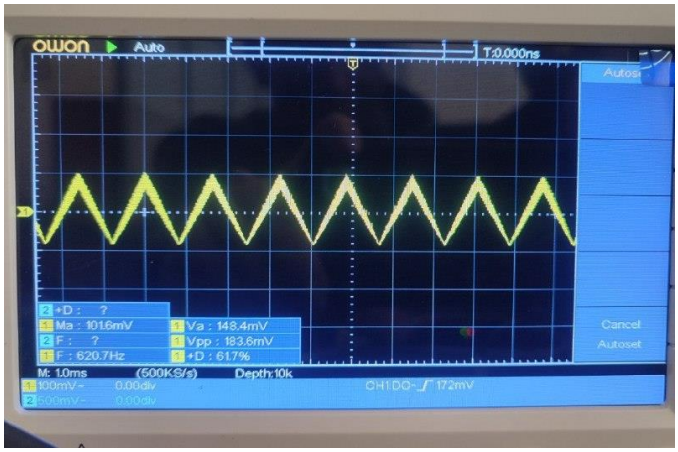


Fig 21. triangle (increase amplitude)

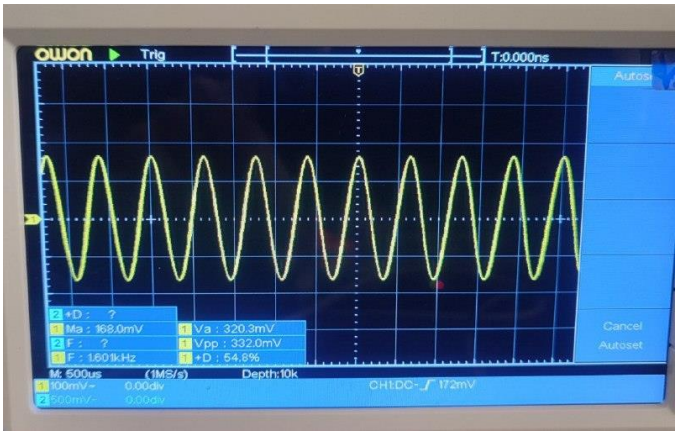


Fig24.sine(increase amplitude)

After sine we have full sine :

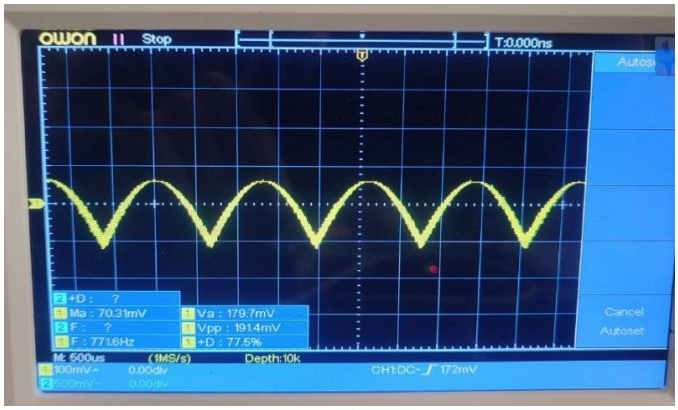


Fig25.full sine

And the last one is half sine :