Experiment 3 – Function Generator

Amirhosein Yavarikhoo 810199514

***Abstract*— In this experiment we use our knowledge of logic circuits to design a function generator. Function generator is used for testing circuits and plays a crucial part in verifying a design.**

***Keywords:* Function Generator, Digital circuit design, sine wave, reciprocal, FPGA, Quartus**

I. Introduction

FPGA boards are used in many different areas in circuit implementation. In this experiment, we design a function generator and implement it on FPGA board. Fig.1 shows a block diagram of this design.

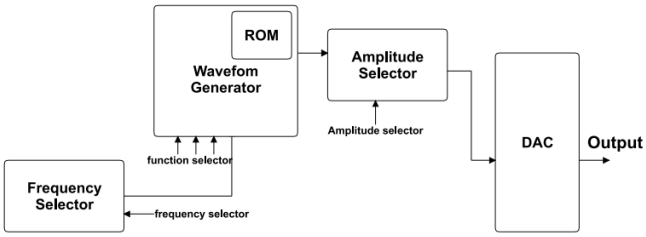

Fig. 1 Block diagram of the AFG

II. Waveform Generator

To design a waveform generator, first we need a counter to act as x-axis for waveform formulas. Verilog code for this counter is shown below:

```
Ln#
1   module counter(input clk,rst, output reg [7:0] count_out);
2   always @(posedge clk) begin
3           if (rst==1'b1) count_out=8'd0;
4           else count_out=count_out+1;
5   end
6   endmodule
7
```

Fig. 2 Counter Verilog code

Now we use this counter to implement square, reciprocal and triangle signal.

```
always @(posedge clk) begin
        if(count_in == 8'd0) triangle = 8'd0;
        if (count_in<8'd127) begin
        square=8'd255;
        triangle = triangle + 1'b1;
        end
        if (count_in>8'd127) begin
        square=8'd0;
        triangle = triangle - 1'b1;
        end
        reciprocal=(255/(256-count_in));
end
```

Fig. 3 Waveform generator Verilog code

To create a sine wave, we use 2 registers to capture and hold previous value of sine and cos. The equation used for calculating sine is shown below:

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

Fig. 4 Sine and cosine equations

Also, we have to initialize sin and cos registers. We use 0 for sin (0) and 30000 for cos (0). Note that inner calculations for sin and cos are 16 bits and we use left 8 bits (most significant bits) as output.

According to these formulas, we implement sine and cos in verilog as shown below:

```
always@(sin_old,cos_old) begin
        sin_new=sin_old+{{6{cos_old[15]}},cos_old[15:6]};
        cos_new=cos_old-{{6{sin_new[15]}},sin_new[15:6]};
end
always @(posedge clk) begin
        sin_old=sin_new;
        cos_old=cos_new;
end
```

Fig. 5 Sine and cosine Verilog code

To create half wave rectified, we use 127 decimal number as a threshold. If sine wave is lower than this, the value will be 127. Verilog code for this design is shown below:

```
always @(sin_new) begin
if (sin_out<8'd127)
        halfrect=8'd127;
else
halfrect=sin_out;

end
```

Fig. 6 Half wave rectifier Verilog code

To create full wave rectified, we have to flip the signal (which means that we subtract it from the double of the signal) and add 127 to it so the whole signal would be above 127. Verilog code for this design is shown below:

```
always @(sin_new) begin
if (sin_out<8'd127)
        fullrect=sin_out+2*(8'd127-sin_out);
else
fullrect=sin_out;

end
```

Fig. 7 Full wave rectifier Verilog code

Flow Summary	
Flow Status	Successful - Wed May 10 08:51:08 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	DDS
Top-level Entity Name	DDS
Family	Cyclone IV E
Total logic elements	8 / 6,272 (< 1 %)
Total registers	8
Total pins	11 / 92 (12 %)
Total virtual pins	0
Total memory bits	2,048 / 276,480 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)
Device	EP4CE6E22C6
Timing Models	Final

Fig. 8 DDS synthesis report using quartus ROM

Flow Status	Successful - Wed May 10 09:16:11 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	DDS
Top-level Entity Name	DDS
Family	Cyclone IV E
Total logic elements	2 / 6,272 (< 1 %)
Total registers	2
Total pins	11 / 92 (12 %)
Total virtual pins	0
Total memory bits	32 / 276,480 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)
Device	EP4CE6E22C6
Timing Models	Final

Fig. 9 DDS synthesis report using Verilog code

Finally, we design a testbench for waveform generator. Fig. 10 shows results of the waveform generator testbench.

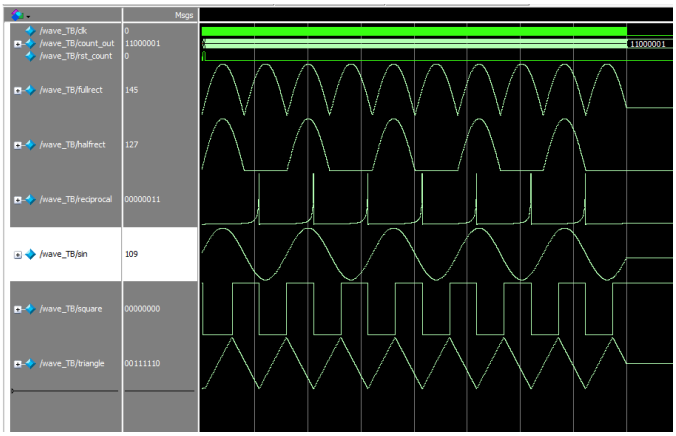

Fig. 10 Waveform of the waveform generator

Flow Status	Successful - Tue May 23 11:01:47 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	LAB3final
Top-level Entity Name	wave_gen
Family	Cyclone IV E
Device	EP4CE6E22A7
Timing Models	Final
Total logic elements	191 / 6,272 (3 %)
Total registers	49
Total pins	57 / 92 (62 %)
Total virtual pins	0
Total memory bits	0 / 276,480 (0 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)

Fig. 11 synthesis report of the waveform generator

III. DAC USING PWM

Until this part, we've been creating discrete signals. We need to convert these signals to analog signals. In this experiment, we use a PWM that functions with a much faster clock (at least 256 times faster). This device is a counter that compares input signal (meaning waveform generator output) with it's counter. If the counter is higher, it issues a one bit 0 as output. Verilog code for PWM is shown below:

```
1  module PWM( input clk,rst,input [7:0] data, output reg out);
2          reg [7:0] count_out;
3          counter cnt (clk,rst,count_out);
4          always@(posedge clk) begin
5          if (count_out>data) out=1'b0;
6          else out=1'b1;
7          end
8  endmodule
9
```

Fig. 12 PWM Verilog code

IV. AMPLITUDE SELECTOR

We use an amplitude selector to scale down the amplitude of the waveform. This function is implemented by shifting the input as shown in Fig.13.

```
module AmpSel ( input [7:0] in ,input [1:0] sw65, output [7:0] out);
assign out=(sw65==2'd0)?in:
          (sw65==2'd1)?(in/2):
          (sw65==2'd2)?(in/4):
          (sw65==2'd3)?(in/8):in;
endmodule
```

Fig. 13 Amplitude selector Verilog code

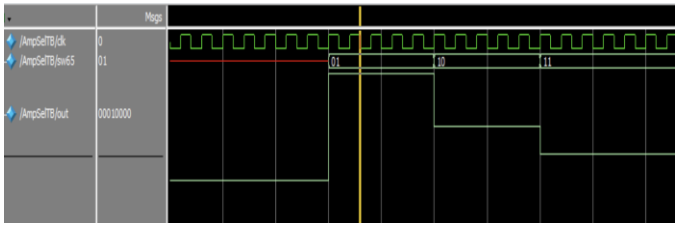

Fig. 14 Amplitude selector waveform

V. Frequency Selector

The function generator also needs a frequency selector to generate waves with different frequencies. We use a 9-bit counter and set 4 least significant bits to 10 and 5 most significant to the SW as shown in Fig. 15.

```
module freq_sel (input rst, input [4:0] SW, input clk, output reg out_clk,output reg [8:0] count);
wire [8:0] count_init;
assign count_init[3:0]=4'd10;
assign count_init[8:4]=SW;
always @(posedge clk) begin
if (rst==1'b1) count=9'd0;
if (count==9'd0) count=count_init;
count=count+1;
if (count==9'd511) out_clk=1'b1;
else out_clk=1'b0;
end
endmodule
```

Fig. 15 Frequency selector Verilog code

VI. Finalizing Design

Now that we have designed all the components, we use Quartus block diagram to connect them all together. Schematic is shown below in Fig. 16.

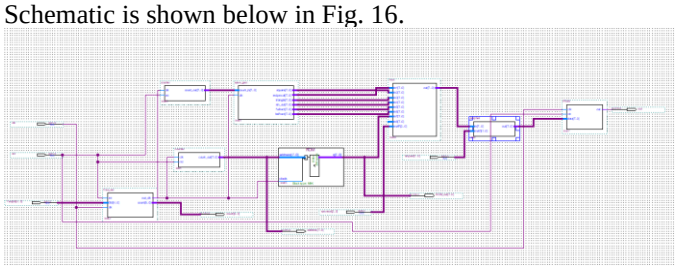

Fig. 16 Final design in quartus

Synthesis report is shown below:

Flow Status	Successful - Tue May 23 11:22:37 2023
Quartus Prime Version	20.1.0 Build 711 06/05/2020 SJ Lite Edition
Revision Name	LAB3final
Top-level Entity Name	LAB3final
Family	Cyclone IV E
Device	EP4CE6E22A7
Timing Models	Final
Total logic elements	44 / 6,272 (< 1 %)
Total registers	14
Total pins	38 / 92 (41 %)
Total virtual pins	0
Total memory bits	16 / 276,480 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 30 (0 %)
Total PLLs	0 / 2 (0 %)

Fig. 17 Synthesis report of the final design

Following pictures show results produced by FPGA board and observation made by oscilloscope.

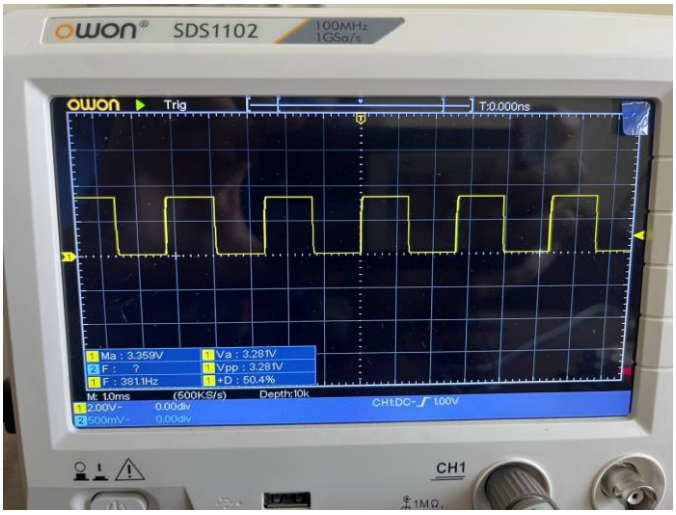

Fig. 18 Square wave

Fig. 19 Triangular wave

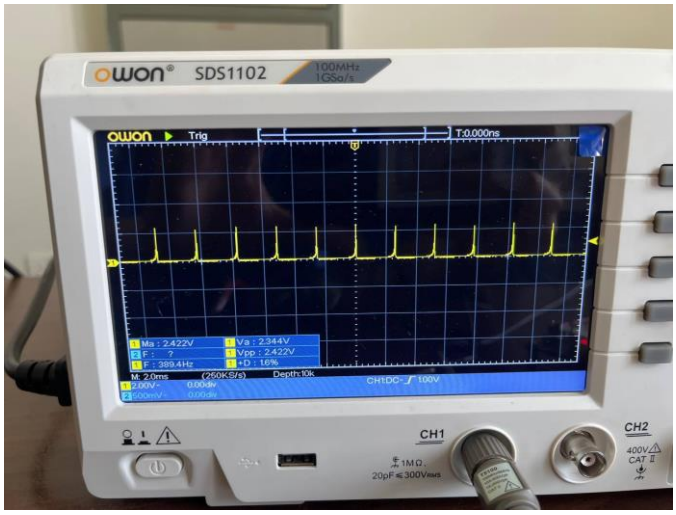

Fig. 20 Reciprocal wave

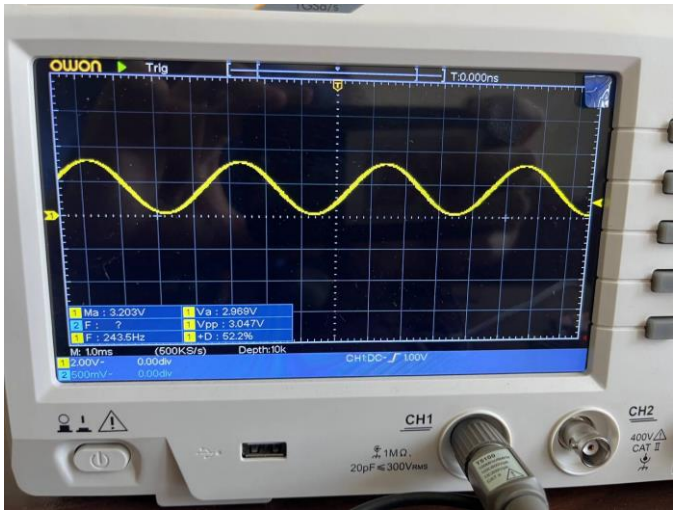

Fig. 21 Sine wave

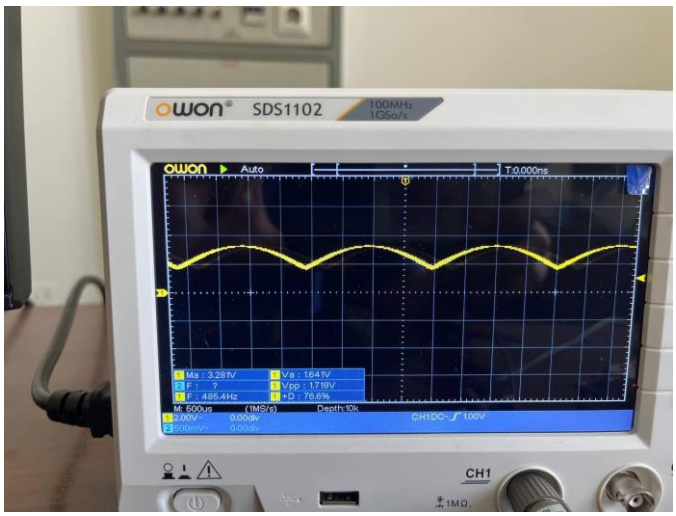

Fig. 22 Full-wave rectified

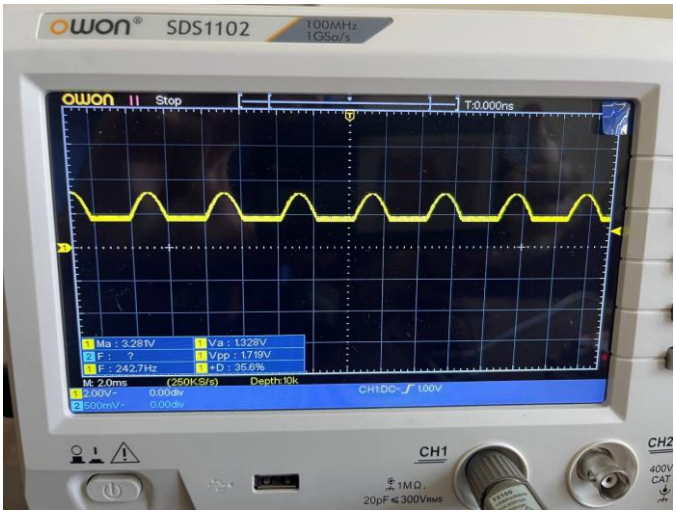

Fig. 23 Half-wave rectified

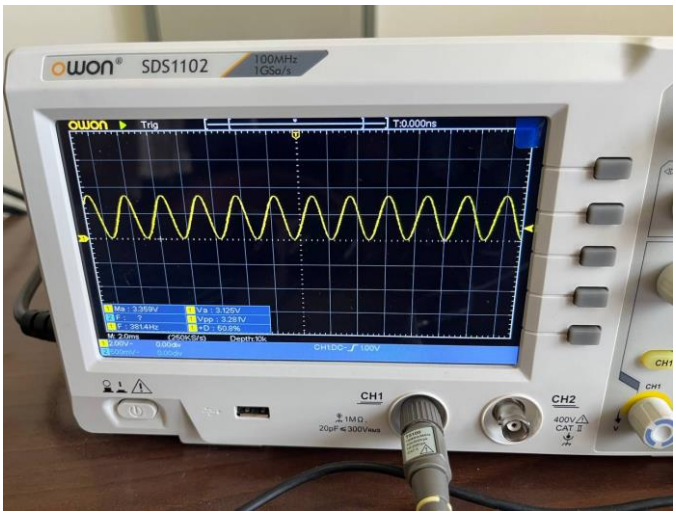

Fig. 24 DDS output wave

Now we test amplitude selector:

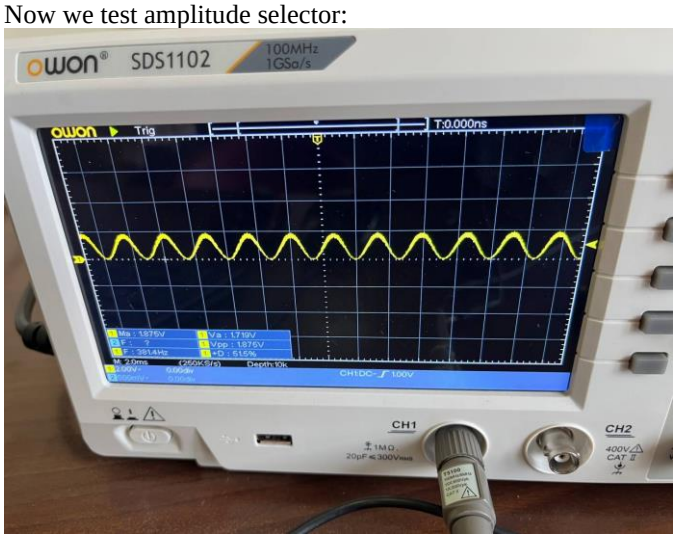

Fig. 25 Sine wave with different amplitudes

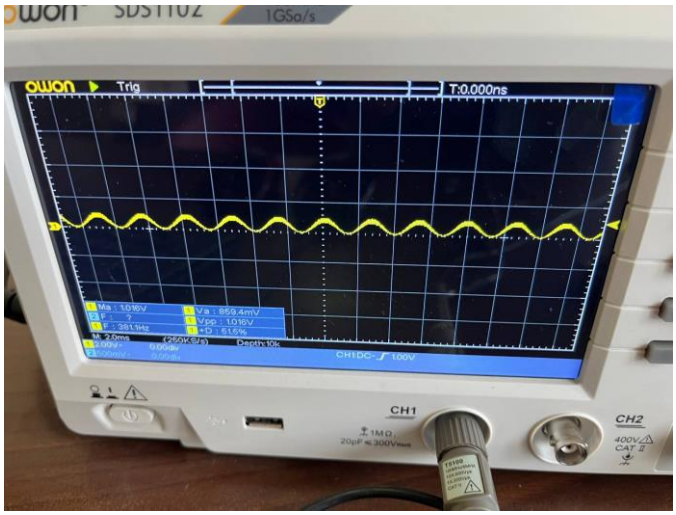

Fig. 26 Sine wave with different amplitudes

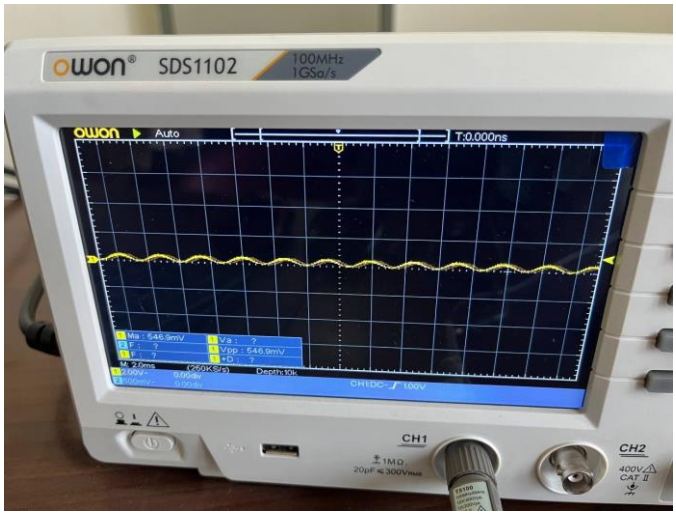

Fig. 27 Sine wave with different amplitudes

At last we use frequency selector to verify its functionality:

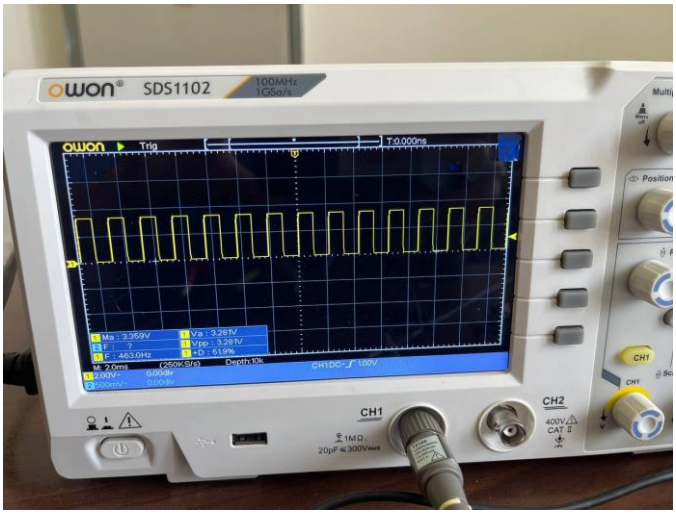

Fig. 28 Square wave with different frequencies

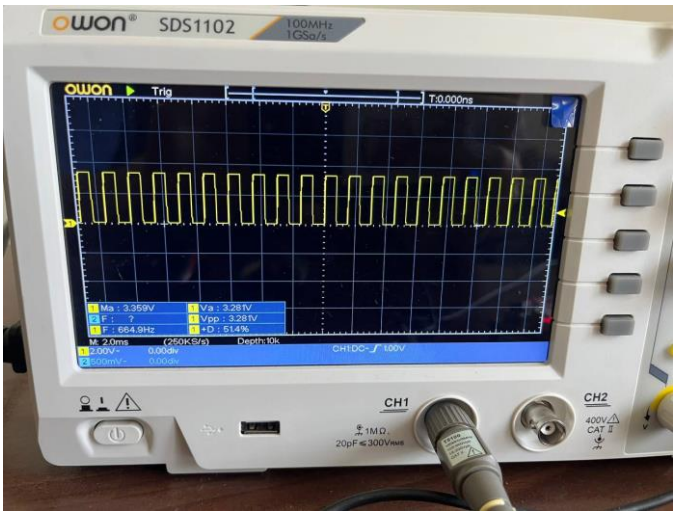

Fig. 29 Square wave with different frequencies

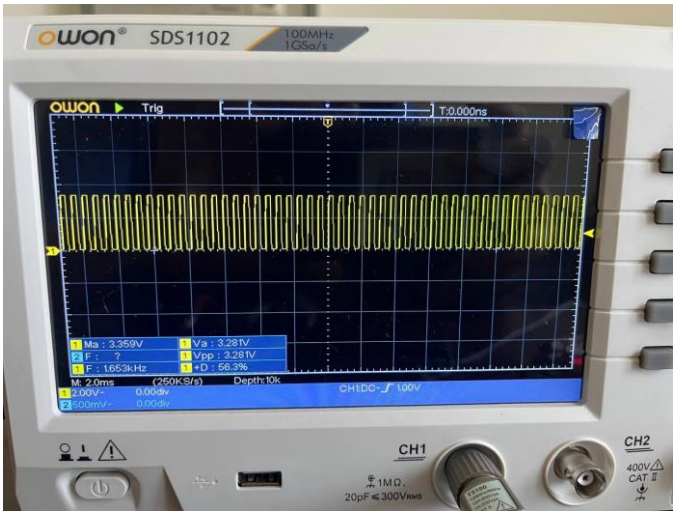

Fig. 30 Square wave with different frequencies

Now we change phase_cntrl to verify its functionality:

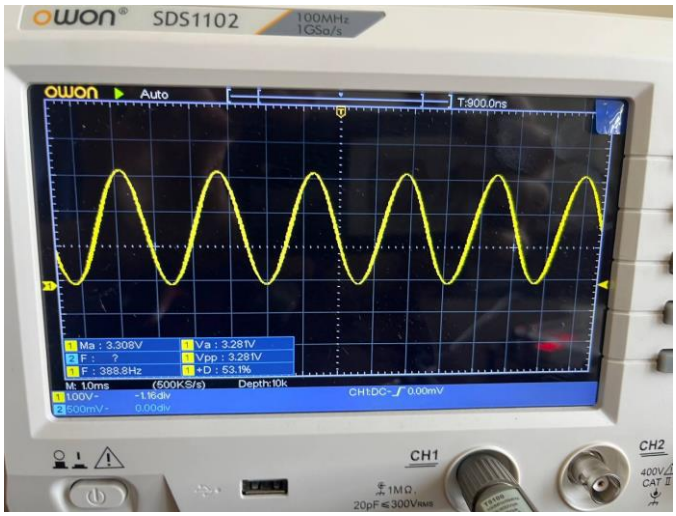

Fig. 31 Generating sine wave with different frequencies using DDS

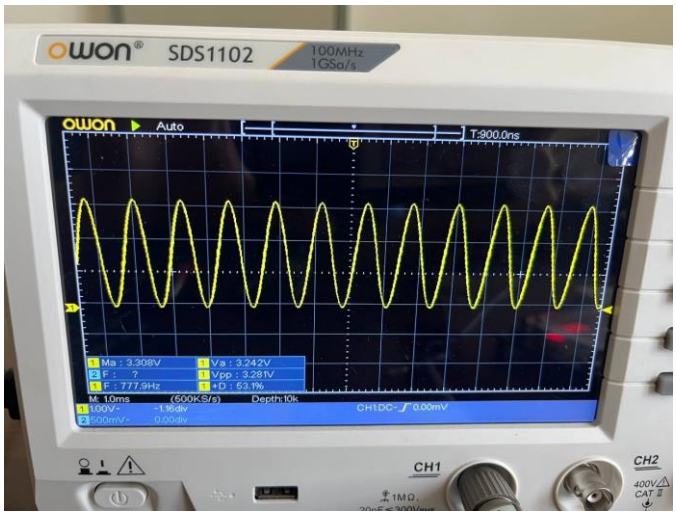

Fig. 32 Generating sine wave with different frequencies using DDS

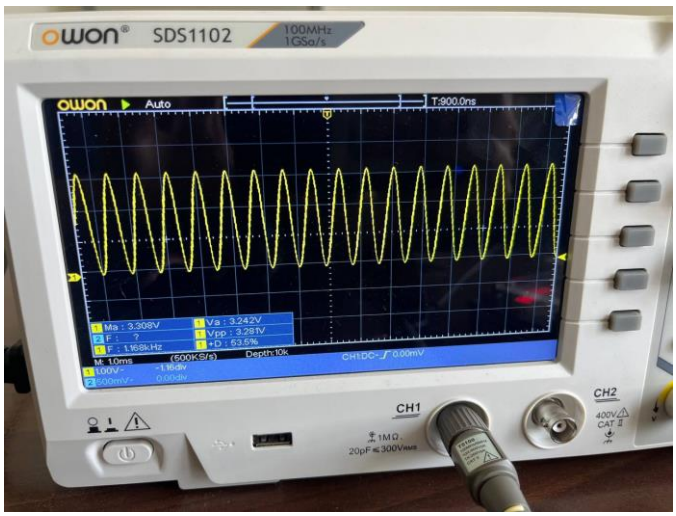

Fig. 33 Generating sine wave with different frequencies using DDS