Experience #3 – Function Generator

Mohammad Sadeghi /810100175 Ali Ghahari/810100201

Introduction

Goal: the main purpose of this laboratory is to design an Arbitrary Function Generator that it can generates wide range of waveforms with a different frequency selection. Here is a top level for this module:

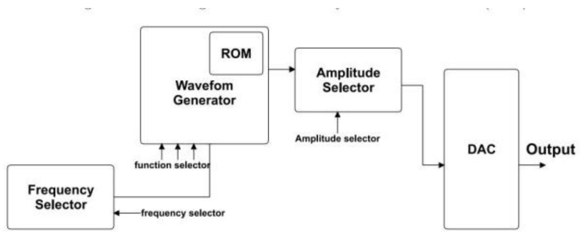


Fig1-Block diagram of the Arbitrary Function Generator

We have 4 main prat in this implementation:

1- Frequency selector
2- Wave generator
3- Amplitude selector
4- DAC(digital to analog converter)

Also we have an ROM (read only memory) to read our sinus function values for simulating sinus wave.

Waveform Generator

this module will create out desired waves and its output is 8 bit digital which is represent the signal amplitude.

Figure 6: Block diagram of waveform generator

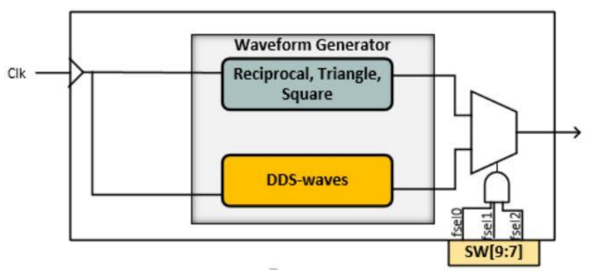
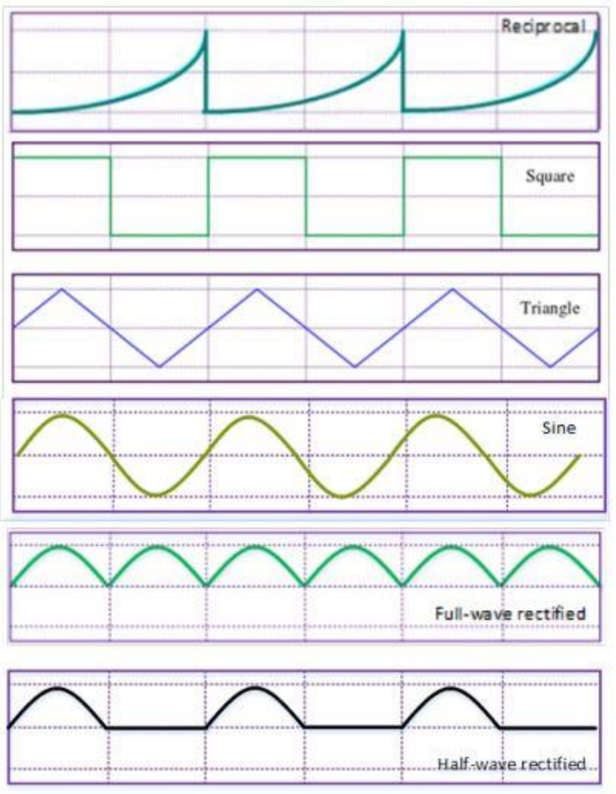


Fig2-Block diagram of the waveform generator

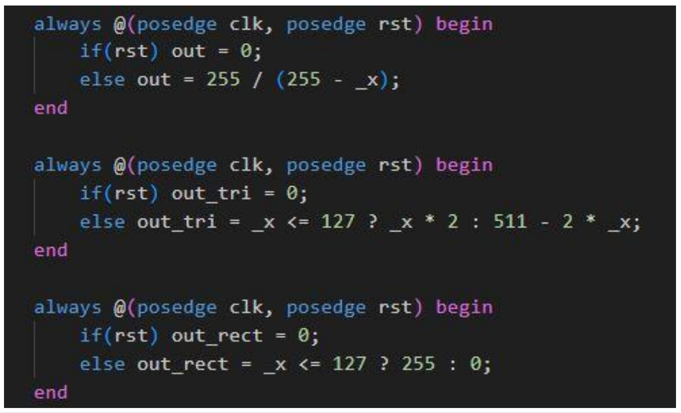
The

Supported functions, shown in figure3, are sine, square, reciprocal, triangle, full-wave, and half-wave rectified signals.

Figure 3: Different waveforms of function generator

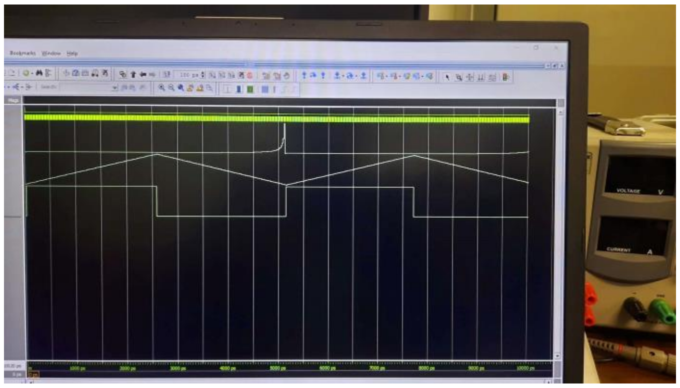


Waveforms square, reciprocal, and triangle are based on a 8bit counter. The output of the frequency selector is the input clock for this module . we have separated logic for calculating value of each wave:



```
always @(posedge clk, posedge rst) begin
    if(rst) out = 0;
    else out = 255 / (255 - _x);
end

always @(posedge clk, posedge rst) begin
    if(rst) out_tri = 0;
    else out_tri = _x <= 127 ? _x * 2 : 511 - 2 * _x;
end

always @(posedge clk, posedge rst) begin
    if(rst) out_rect = 0;
    else out_rect = _x <= 127 ? 255 : 0;
end
```

And this is our digital result for these three waves:



For generating sine wave we use module named **DDS**

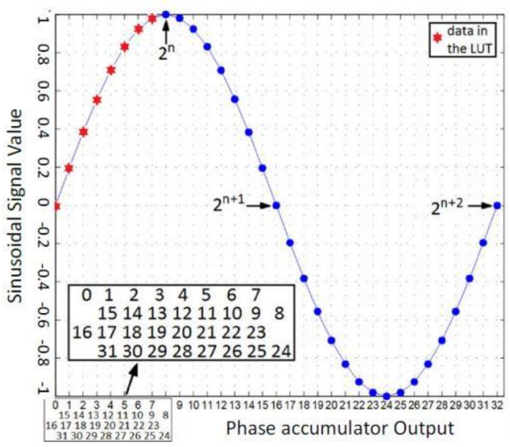
This module has main part calles **phase accumulator** that it has 8 bit output as bellow:

- One bit for *phasepos*
- One bit as *singbit*
- 6 bit as an ROM *address*

For one period of sine wave , we have 4 quadrant which in first and second one our sign is positive and in two other quadrants is negative. The *signbit* output will clear for us the sign for wave value.

Also in first and third quadrant our sine value magnitudes are increasing from 0 to 63. But in two other parts it will decrease from 63 to 0. So the *phasepos* will show us that values must be accumulative or regressive. the next figure will enlighten us the logic:

Figure 4: Phase accumulator output generation

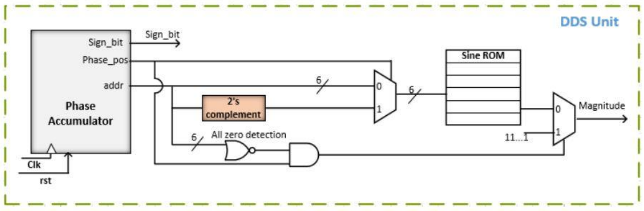


In other part of DDS according to the *phasepos* , we decide to either get 6 bit main address to ROM for reading sine value or its 2's complement.

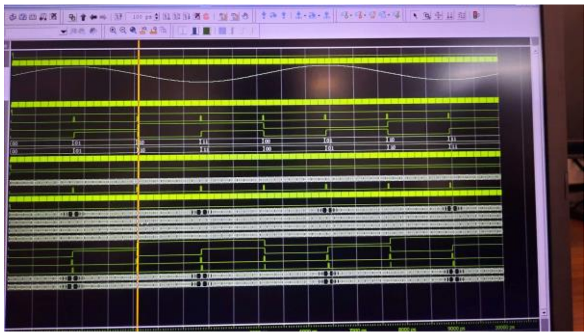
Also for maximum points in sine wave we check if all bit of address was zero , the wave magnitude for sine will be 6'b1.

The fig 5 shows DDS block diagram:

Figure 5: DDS hardware design

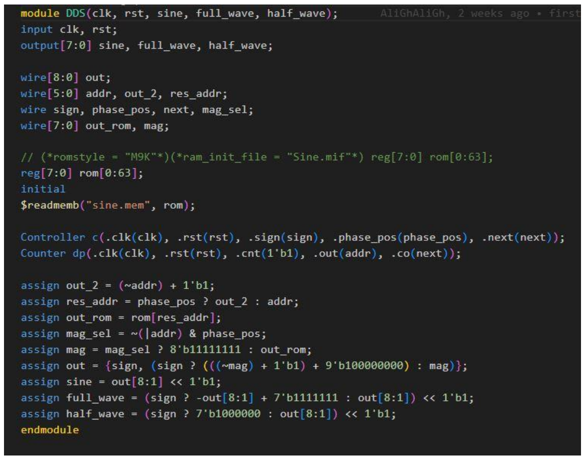


Here is the digital result for sine wave:



Now that we have sine wave we can easily generate full and half form of sine wave. For full wave if *signbit* was 1 means our sine is negative we just set final signal as (-out) and for half way we set it to zero.

So the complete code for DDS is as bellow:



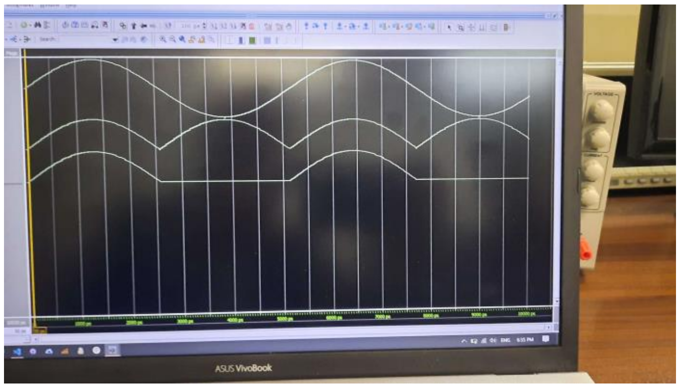
```verilog
module DDS(clk, rst, sine, full_wave, half_wave);      AliGhAliGh, 2 weeks ago • first
input clk, rst;
output[7:0] sine, full_wave, half_wave;

wire[8:0] out;
wire[5:0] addr, out_2, res_addr;
wire sign, phase_pos, next, mag_sel;
wire[7:0] out_rom, mag;

// (*romstyle = "M9K"*)(*ram_init_file = "Sine.mif"*) reg[7:0] rom[0:63];
reg[7:0] rom[0:63];
initial
$readmemb("sine.mem", rom);

Controller c(.clk(clk), .rst(rst), .sign(sign), .phase_pos(phase_pos), .next(next));
Counter dp(.clk(clk), .rst(rst), .cnt(1'b1), .out(addr), .co(next));

assign out_2 = (~addr) + 1'b1;
assign res_addr = phase_pos ? out_2 : addr;
assign out_rom = rom[res_addr];
assign mag_sel = ~(|addr) & phase_pos;
assign mag = mag_sel ? 8'b11111111 : out_rom;
assign out = {sign, (sign ? (((~mag) + 1'b1) + 9'b100000000) : mag)};
assign sine = out[8:1] << 1'b1;
assign full_wave = (sign ? -out[8:1] + 7'b1111111 : out[8:1]) << 1'b1;
assign half_wave = (sign ? 7'b1000000 : out[8:1]) << 1'b1;
endmodule
```

For reading sine values which are saved in `sine.mem` file we use command bellow:

```
$readmemb(“sine.mem”,rom);
```

Which rom is name of our memory with 64 rows and 8 bit value in each row.

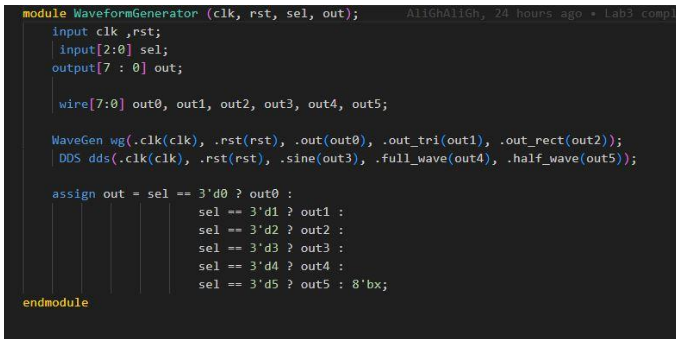
And here is digital wave of different sine wave form:



So now we have all our 6 desired waves and we just can select through them with one MUX shown in fig-2 according to table follow:

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

And the final code for Waveform Generator is:

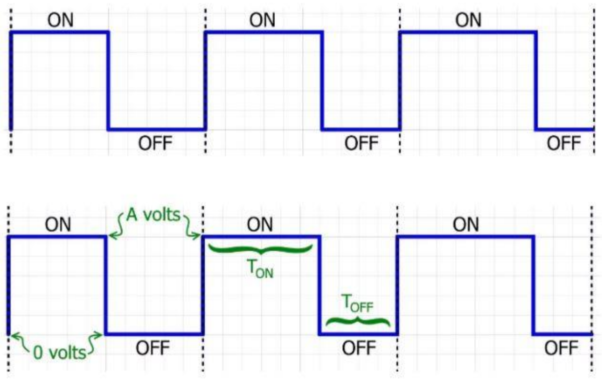


```
module WaveformGenerator (clk, rst, sel, out);        AliGhAliGh, 24 hours ago • Lab3 compl
    input clk ,rst;
     input[2:0] sel;
    output[7 : 0] out;

     wire[7:0] out0, out1, out2, out3, out4, out5;

    WaveGen wg(.clk(clk), .rst(rst), .out(out0), .out_tri(out1), .out_rect(out2));
     DDS dds(.clk(clk), .rst(rst), .sine(out3), .full_wave(out4), .half_wave(out5));

    assign out = sel == 3'd0 ? out0 :
                   sel == 3'd1 ? out1 :
                   sel == 3'd2 ? out2 :
                   sel == 3'd3 ? out3 :
                   sel == 3'd4 ? out4 :
                   sel == 3'd5 ? out5 : 8'bx;
endmodule
```

- Out0 : Reciprocal
- Out1 : Triangle
- Out2 : rectangle
- Out3 : DDS sine
- Out4: full_wave sine
- Out5 : half_wave sine

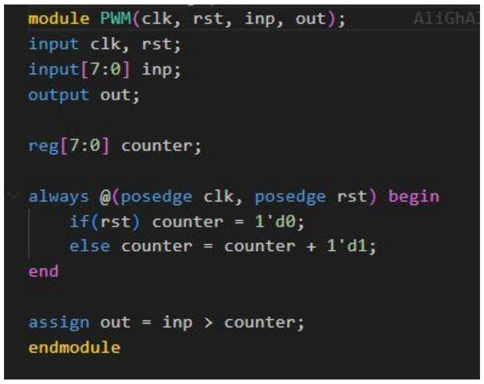
DAC using PWM

For converting digital signal to analog we use PWM. This module has 256 clk so it is doing as 8bit counter and while our input signal is greater than its count value this module will generate 1 analog output which it can use in RC circuit.



So we can easily convert our 8bit digital signal to 1 or 0 analog value.

Here is code for this module:



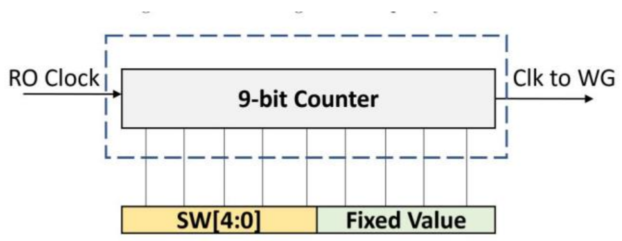
```
module PWM(clk, rst, inp, out);        AliGhA
input clk, rst;
input[7:0] inp;
output out;

reg[7:0] counter;

always @(posedge clk, posedge rst) begin
    if(rst) counter = 1'd0;
    else counter = counter + 1'd1;
end

assign out = inp > counter;
endmodule
```

Frequeny selector

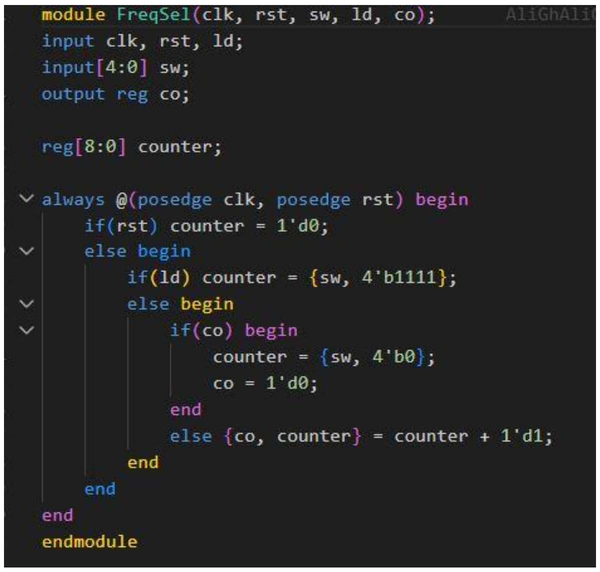
This module will create the clk for waveform generator and DDS. It is only a 9 bit counter which 5 left bit of its loaded value setting by SW[4:0] of board switches and other 4 bit are fixed numers.

Also we control loading data with OR ld signal with key[0] which if it become one the loading can done.

This is block diagram for frequency selector:



Also we have Verilog code for that:



```
module FreqSel(clk, rst, sw, ld, co);
input clk, rst, ld;
input[4:0] sw;
output reg co;

reg[8:0] counter;

always @(posedge clk, posedge rst) begin
    if(rst) counter = 1'd0;
    else begin
        if(ld) counter = {sw, 4'b1111};
        else begin
            if(co) begin
                counter = {sw, 4'b0};
                co = 1'd0;
            end
            else {co, counter} = counter + 1'd1;
        end
    end
end
endmodule
```

Amplitude Selector

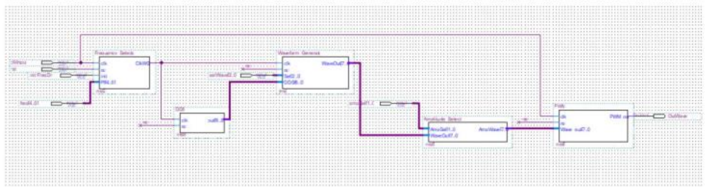
This module will change the frequency of input signal and divide it by 2 , 4 or 8 or don't change frequency.

It just do it by simple 4 to 1 MUX and we select our divide value with SW[6:5] according to this table:

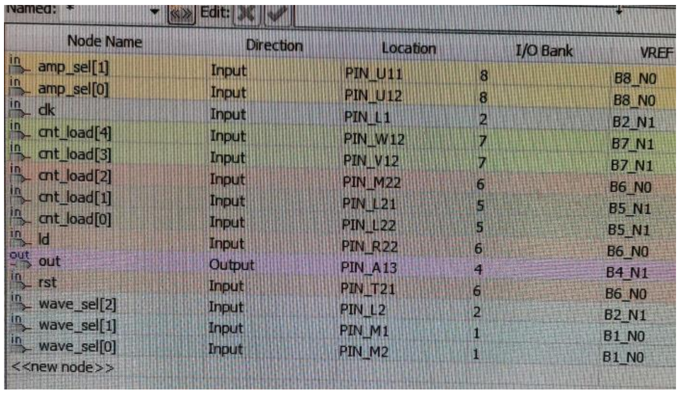
SW[6:5]	Amplitude
2'b00	1
2'b01	2
2'b10	4
2'b11	8

The Total design

At last we add all Verilog code to Quartus and make block symbol for all modules and the final design for AFG is as bellow:



At last part we assigned our board PINs to out input and MUX selectors:



Node Name	Direction	Location	I/O Bank	VREF
amp_sel[1]	Input	PIN_U11	8	B8_N0
amp_sel[0]	Input	PIN_U12	8	B8_N0
clk	Input	PIN_L1	2	B2_N1
cnt_load[4]	Input	PIN_W12	7	B7_N1
cnt_load[3]	Input	PIN_V12	7	B7_N1
cnt_load[2]	Input	PIN_M22	6	B6_N0
cnt_load[1]	Input	PIN_L21	5	B5_N1
cnt_load[0]	Input	PIN_L22	5	B5_N1
ld	Input	PIN_R22	6	B6_N0
out	Output	PIN_A13	4	B4_N1
rst	Input	PIN_T21	6	B6_N0
wave_sel[2]	Input	PIN_L2	2	B2_N1
wave_sel[1]	Input	PIN_M1	1	B1_N0
wave_sel[0]	Input	PIN_M2	1	B1_N0
<<new node>>				

And we have these three analog signals for instance: