Experiment 3 – Function Generator

Matin Bazrafshan, 810100093

Mohammad Amin Yousefi, 810100236

Abstract— **This document is a report for experiment 3 of Digital Design Laboratory at ECE department at University of Tehran. The purpose of this experiment is to design a sequence detector and synthesize on an FPGA .**

Keywords—**DE1 FPGA, Wave Generator, PWM, Oscilloscope, Frequency Selector, Amplitude Selector**

I. Introduction

In this experiment we will design a Wave Generator, First we simulate waves with Verilog in Modelsim, Them using Quartus we implement it on an FPGA and finally we see Waves on an oscilloscope. After that we will see different between using LUT and memory in FPGA.

II. Simulation on Verilog

A. Wave Generator

This module is the heart of this project. It 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.

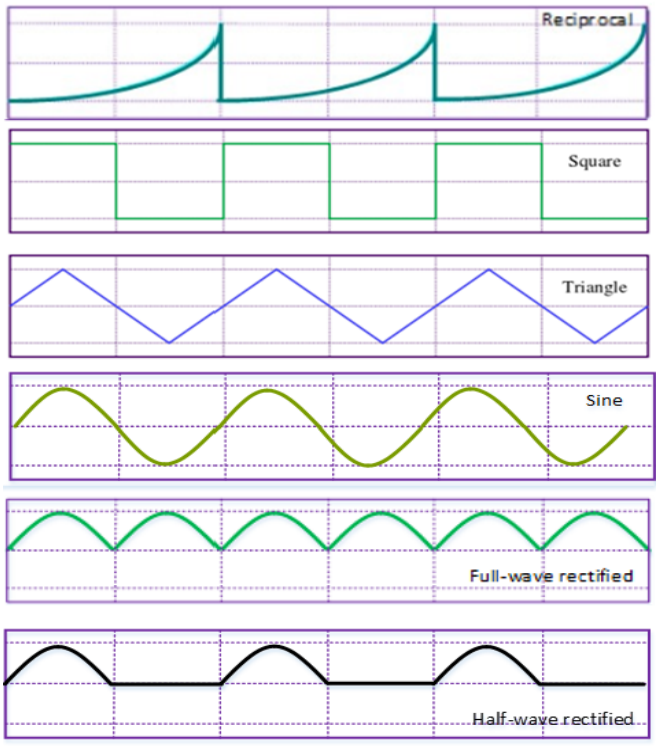


Fig. 1 Waveforms

```
module WaveGenerator (
    input clock,reset,
    input [7:0] count,
    output [7:0] square, triangle, reciprocal, sin, full, half
);
    assign square = (count < 128) ? 0 : 255;
    assign triangle = (count < 128) ? count << 1 : (255 - count) << 1;
    assign reciprocal = 255 / (256 - count);

    reg [15:0] snn = 0, cnn = 30000;
    reg [15:0] sn, cn;
    always @(posedge clock, posedge reset) begin
        if (reset) begin
            snn = 0;
            cnn = 30000;
            cn = 0;
            sn = 30000;
        end
        else begin
            sn = snn + {cnn[15],cnn[15],cnn[15],cnn[15],cnn[15],cnn[15],cnn[15:6]};
            cn = cnn - {sn[15],sn[15],sn[15],sn[15],sn[15],sn[15],sn[15:6]};
            snn = sn;
            cnn = cn;
        end

    end

    assign sin = sn[15:8] + 127;
    assign full = (sin >= 127) ? sin : sin + 2*(127-sin);
    assign half = (sin >= 127) ? sin : 127;
endmodule
```

Fig. 2 Wave Generator Verilog Code

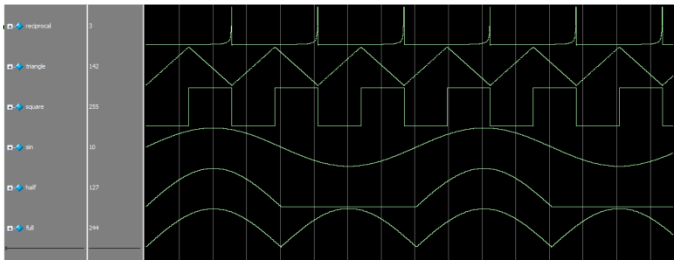


Fig. 3 Waveforms Simulated on Modelsim

B. Digital to Analog conversion using PWM

There are different methods for digital to analog conversion. You can either use an external chip like using an add-on-board card that consists of both ADC and DAC or it can be implemented using Pulse Width Modulation (PWM). 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 or decrease vice versa. The duty cycle of the PWM signal is equal to:

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

```verilog
module PWM (
    input [7:0] signal_in,
    input clock, reset,
    output reg signal_out
);
    reg [7:0] counter = 0;
    always@(posedge clock, posedge reset) begin

        if (reset)
            counter = 0;

        else begin

            if (counter < signal_in)
                signal_out <= 1;
            else
                signal_out <= 0;

            counter = counter + 1;
        end

    end
endmodule
```

Fig. 4 PWM Verilog Code

C. 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.

We use a 9-bit counter with 3 most-left bits can be loaded and 6 bits are fixed.

```verilog
module FrequencySelector (
    input clock, reset, load,
    input [3:0] par_in,
    output reg low_freq_clock
);

reg [4:0] fixed = 0;
reg [8:0] count;
wire carry;
initial begin
    low_freq_clock = 0;
    count = {par_in, fixed};
end
always @(posedge clock, posedge reset) begin

    if(reset)
        count = {par_in, fixed};
    else if((~load) || carry)
            count = {par_in, fixed};
    else
        count = count + 1;

end

assign carry = &{count};

always @(posedge carry) begin
    low_freq_clock = ~low_freq_clock;
end

endmodule
```

Fig. 5 9-bit counter with 6 fixed bits and 3 loadable bits

D. 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

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

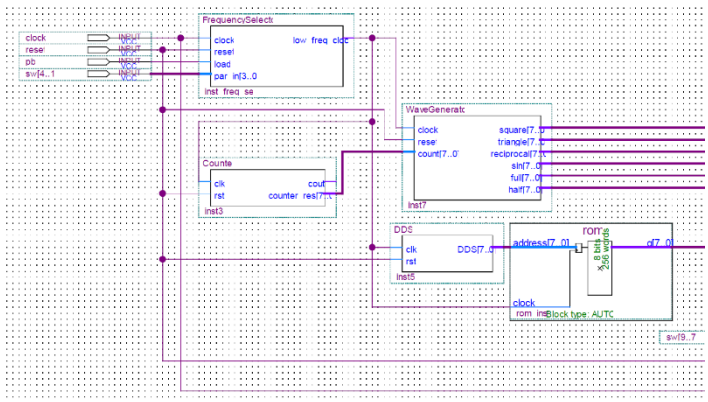
Fig. 6 Divide Select in Amplitude Selector

```verilog
module AmpSelector (
    input [7:0] wave_in,
    input [1:0] select,
    output reg [7:0] wave_out
);

always begin
    case (select)
        0: wave_out = wave_in;
        1: wave_out = {1'b0,wave_in[7:1]};
        2: wave_out = {2'b0,wave_in[7:2]};
        3: wave_out = {3'b0,wave_in[7:3]};
        default: wave_out = wave_in;
    endcase
end

endmodule
```

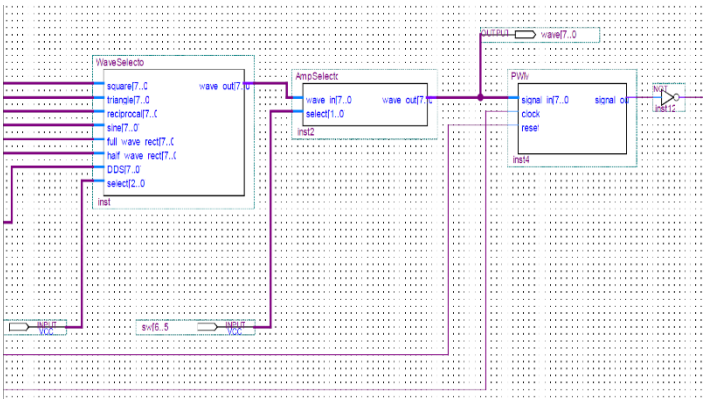
Fig. 7 Amplitude Selector Verilog Code

III. Synthesis

After designing every module, we get an instantiate in Quartus, and then we connect them properly.



Fig, 8 Block Diagram Part I



Fig, 9 Block Diagram Part II (Right Hand Side of Part I)

IV. Waves on Oscilloscope

The output will be given to a Low pass filter (An RC circuit) and then connected to a oscilloscope.

Take note that the output is connected to a NOT, so it is shown in reverse.

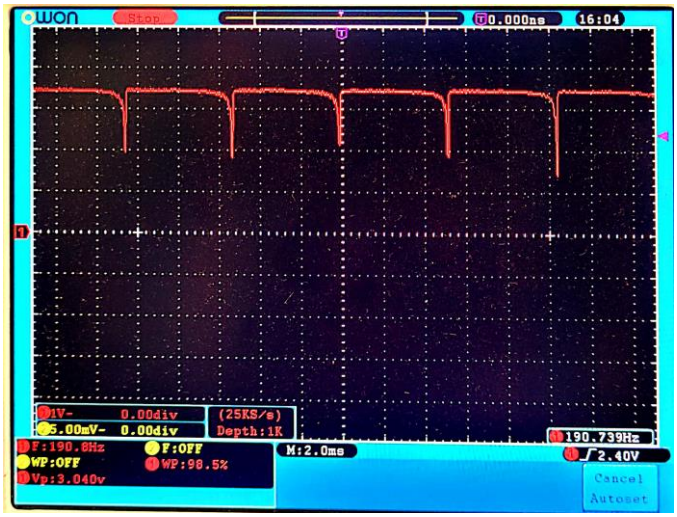


Fig.10 Reciprocal Wave

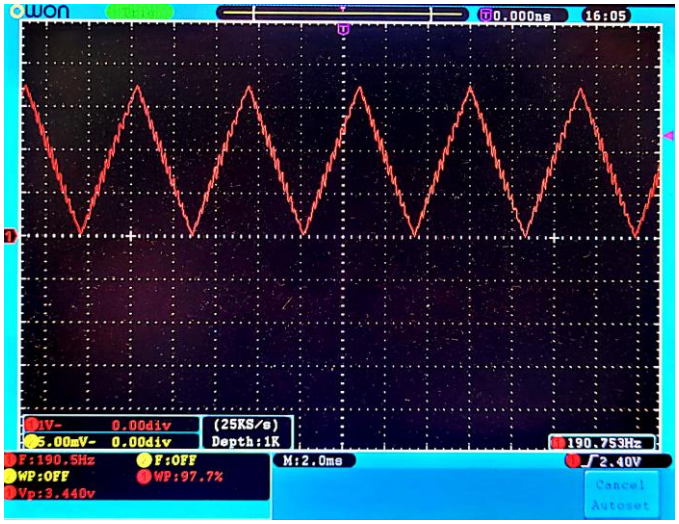


Fig.11 Triangle Wave

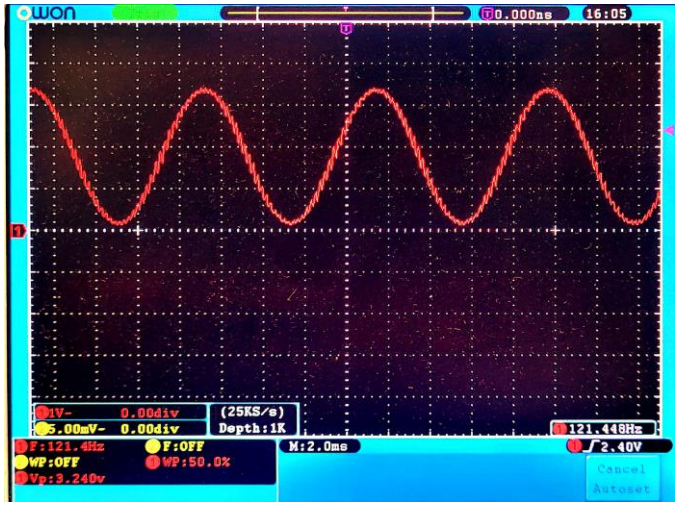


Fig.12 Sine Wave

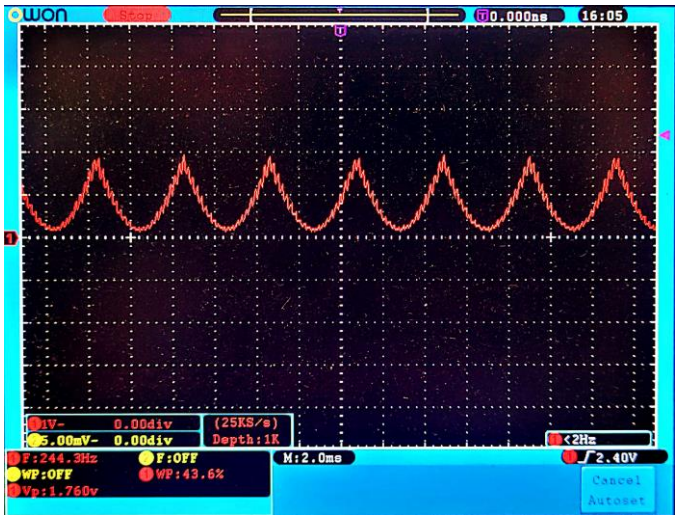


Fig.13 Sine Full-Wave Rectified Wave

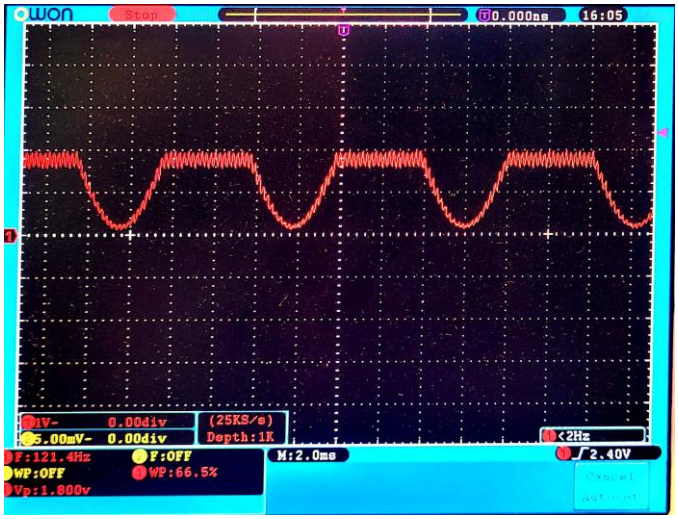


Fig.114 Sine Half-Wave Rectified Wave

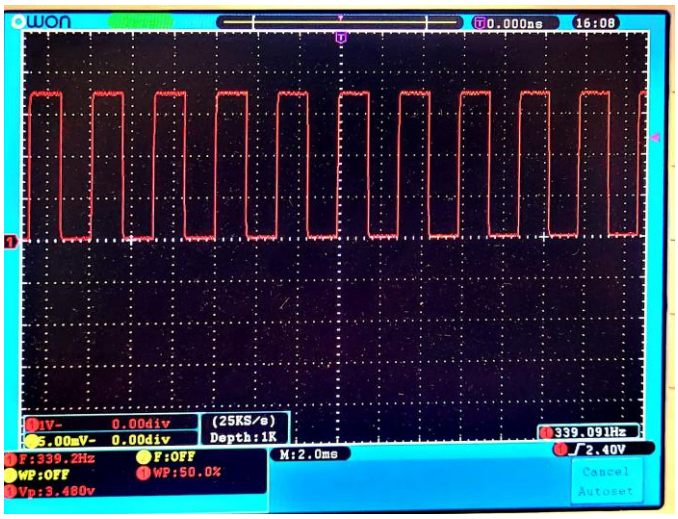


Fig.15 Square Wave

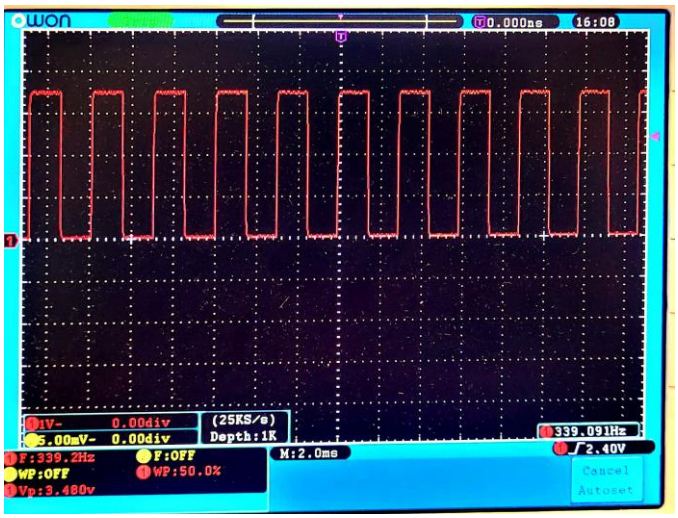


Fig.16 Square Wave with Higher Frequency

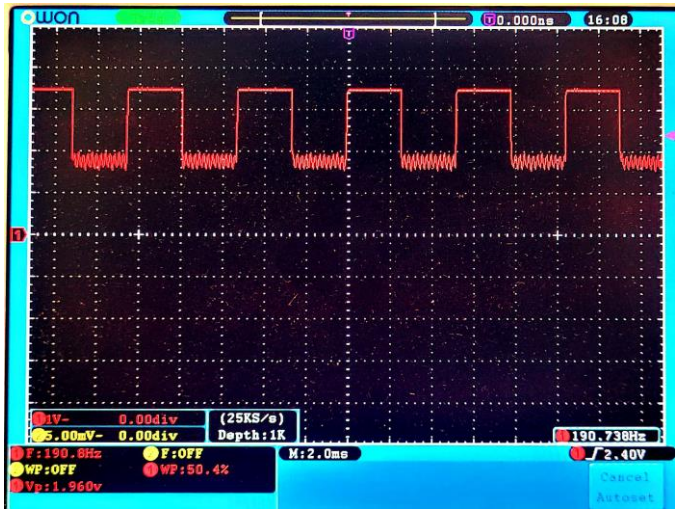


Fig.17 Square Wave With Amp 1/2

Fig.18 Square Wave with Amp 1/8

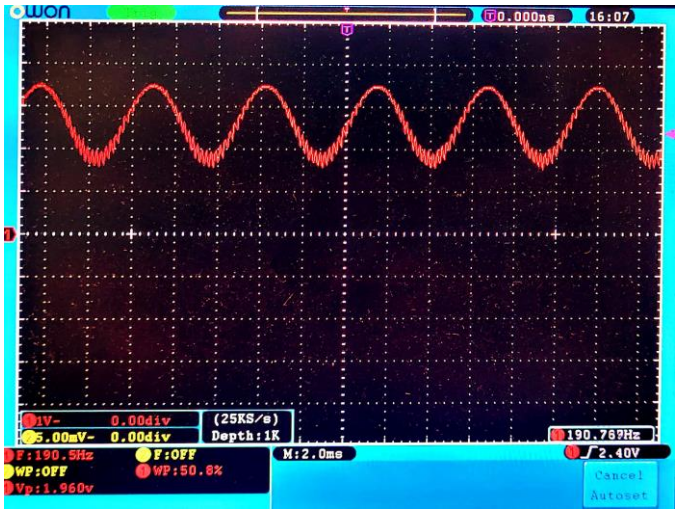


Fig.19 Sine Wave with Amp 1/4

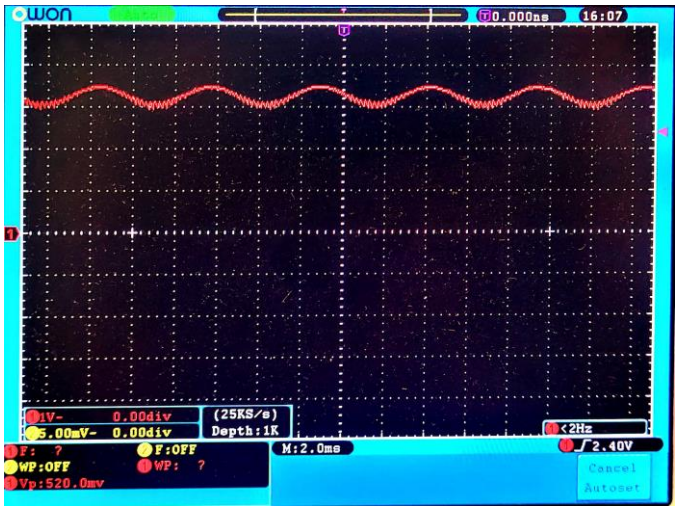


Fig.20 Sine Wave with Amp 1/8

V. DDS AND ROM

Most modern function generators use Direct Digital Synthesis (DDS) for generating their output waveforms. DDS is used for generating arbitary phase tunable output from a single fixed-frequency reference clock (like an oscillator). The output of DDS module is a quantized version of the output files (usually a sinusoid). The period of this signal is controlled by a Phase control value. To generate the DDS signal, you should use a 1-port ROM memory to store the value of a sine wave for several clock cycles.

In FPGA if we get a ROM instantiate from synthesizer(like Quartus) it will use a memory instruction as a ROM but if we want to write a ROM module manually we can use an LUT or a memory instruction as a ROM, using memory is more efficient so we must declare in our code that we want to use a memory instead of LUT because LUT is chosen default.

```
module ROM (
    input clock,
    input [7:0] address,
    output reg [7:0] out
);
    // "  (* romstyle = "M9K" *)    " is written to use memory instead of LUT
    (* romstyle = "M9K" *)(* ram_init_file = "sine.mif" *) reg [7:0] mem [7:0];

    always @(posedge clock) begin
        out = mem[address];
    end

endmodule
```

Fig.21 ROM Verilog Code

```
module DDS (
    input clk, rst,
    output [7:0] DDS
);
reg [7:0] address = 0;
    always @(posedge clk, posedge rst) begin
        if(rst)
            address = 8'b0;
        else begin
            address = address + 1;
        end

    end
    assign DDS = address;
endmodule
```

Fig.22 DDS Verilog Code

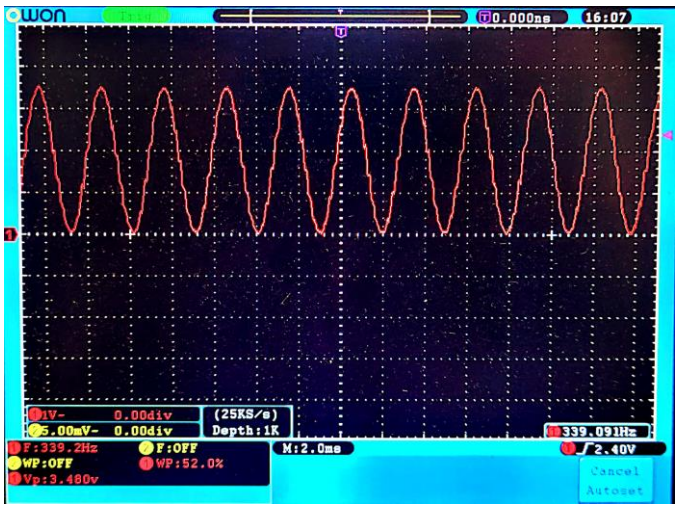


Fig.23 Sine Wave Using DDS

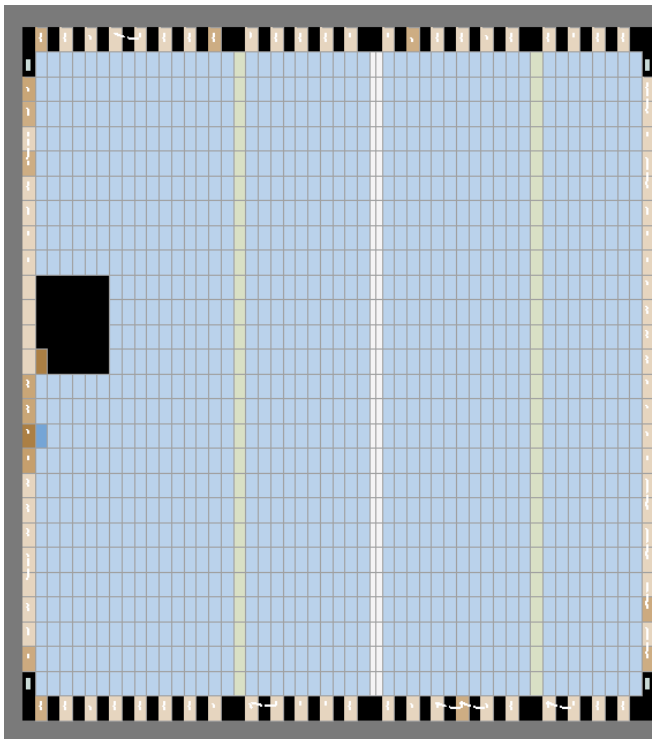


Fig.24 Using a LUT as a ROM in FPGA

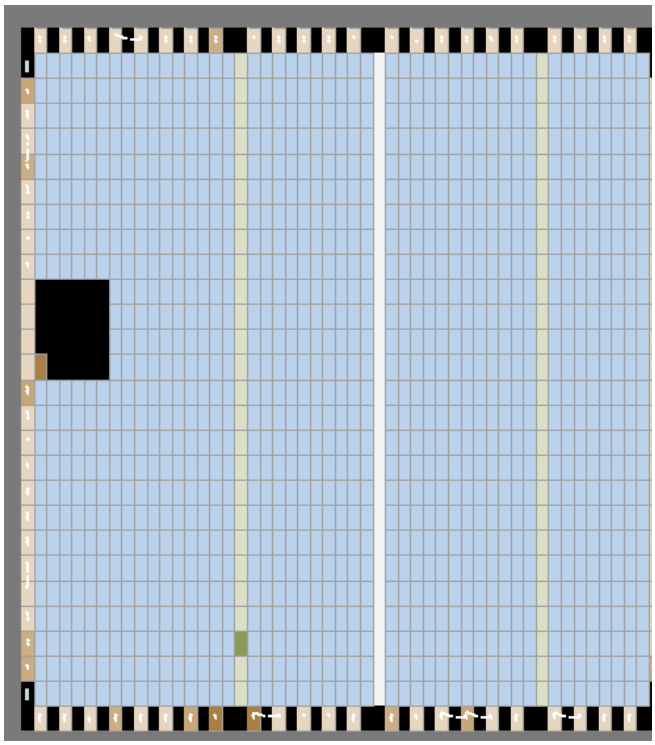


Fig.25 Using a Memory as a ROM in FPGA

VI. Effect of Capacitor on Smoothing

As we increase capacitance of RC circuit, the sampling will be better and the output becomes more smoother.

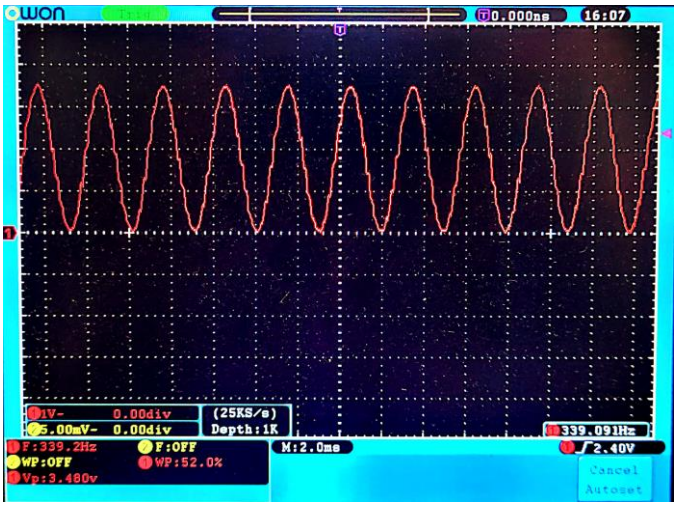


Fig.26 Sine Wave With 10 nF Capacitor

Fig.27 Sine Wave With 200 nF Capacitor