Experiment #3

Group 3

Nazanin Sabri

810194346

nazanin.sabrii@gmail.com

Nima Jarrahiyan

810194292

jarrahian.nima76@gmail.com

***Abstract*—working with Verilog, working with FPGAs, Clock, PWM, DAC, Function Generator.**

***Keywords*— clock divider, oscillator, FPGA, PWM, Function Generator.**

I. Introduction

After building a steady clock from our previous experiments now we are able to build a Function Generator.

A Function Generator consists of 4 parts, which are considered important to build a signal:

1. Bandwidth
2. Frequency
3. Waveform
4. DAC

We build each part in a different session and attach the parts in Quartus and test our simulation on an FPGA.

II. Methodology and Procedures

Procedures we have taken is consisted of building each parts mentioned above separately and then connecting them via wires using a block diagram. So from each part a symbol must be made and then used as a component.

We started with the PWM module one important issue we faced in this part was not using at least one of the signals used in the always sensitivity list in the always body. The logic of this part was to have a counter count up to a threshold value and adjust the output of the module accordingly.

When then moved on to the Frequency Selector, which initial had a 8 bit parallel load and a counter that started counting from the parallel load value up until an overflow occurred and we adjusted the output of 1 or 0 accordingly but when we got to the final parts of the experiment we had to change to structure of this unit due to the small number of available switches that would control the input. (because we needed the switches for function selector and amplitude selector too).

We then moved on to the wave generator. The module it self only has instances of the wave creators and a selection mechanism based on the corresponding FPGA switches. How the codes were written is explained in the results part.

When all modules were completed we connected them in a block diagram.

Our main problem was not getting a decent output from different signals and were not able to solve it in two sessions. Nature of data being digital, we could not use information along side each part to test it.

We however, were able to get a decent square wave.

III. Results

A. PWM

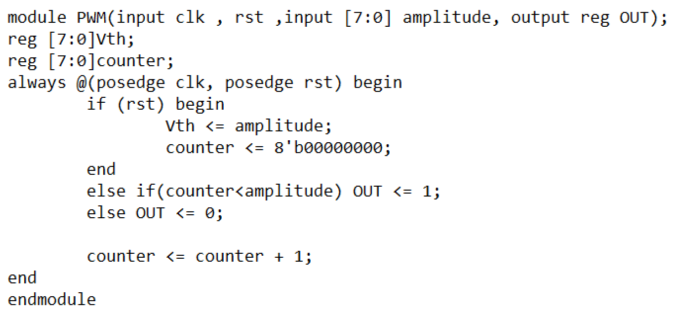
What PWM does is convert digital signal to analog signal using time of 1's and 0's and changing them into duration of Duty Cycle with applying a DC of desired size.

Another way to build a PWM is to use an on board DAC chip but we did not use this method.

Output of PWM is connected to a 1KΩ resistor and a small Capacitor as a low pass filter.

To test our PWM, we made a test bench using data signals and examined output, which worked as implanted.

Here is how the code works: on the posedge of reset we make the value of the counter register zero and then with every positive edge of clock we add one to it. As long as the counter is smaller that a threshold value the one-bit output is one and once it is bigger up until when an overflow occurs the output will be zero.



```
module PWM(input clk , rst ,input [7:0] amplitude, output reg OUT);
reg [7:0]Vth;
reg [7:0]counter;
always @(posedge clk, posedge rst) begin
        if (rst) begin
                Vth <= amplitude;
                counter <= 8'b00000000;
        end
        else if(counter<amplitude) OUT <= 1;
        else OUT <= 0;

        counter <= counter + 1;
end
endmodule
```

Figure 1 Building PWM

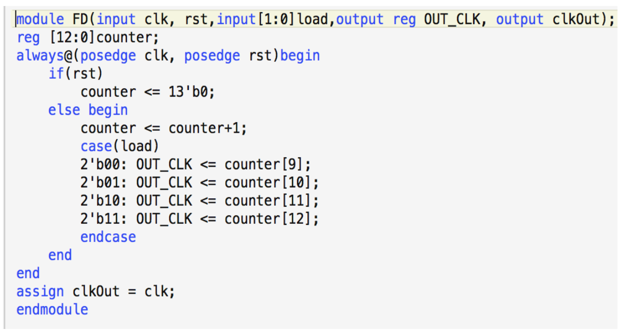
B. Frequency Selector

As done in previous experiment we divided entry signal but this time using codes which was easier and more efficient

We initially used a counter for the divider an make an exact implementation of the counters used in the previous experiments but then changed the code because after the next parts were added we were no longer able to use all 8 bits for parallel load as we once did.

The divided signal would be propagated to generation area which would develop our desired shape.

Here is how the code works: we dedicated two FPGA switches to this module. They would be the load input. Based on the value of load we would decide on 4 different divisions of our signal. The code is shown bellow:



```
module FD(input clk, rst,input[1:0]load,output reg OUT_CLK, output clkOut);
reg [12:0]counter;
always@(posedge clk, posedge rst)begin
    if(rst)
        counter <= 13'b0;
    else begin
        counter <= counter+1;
        case(load)
        2'b00: OUT_CLK <= counter[9];
        2'b01: OUT_CLK <= counter[10];
        2'b10: OUT_CLK <= counter[11];
        2'b11: OUT_CLK <= counter[12];
        endcase
    end
end
assign clkOut = clk;
endmodule
```

Figure 2 Frequency Selector

C. Waveform Generator

Building each signal requires different approach as in Sine we have Tailor series but in Rhomboid we need to develop it by dot by dot implementation.

But for all signals we use one counter which counts up and then we use it differently in each module.

In Sawtooth the counter would directly be connected to the output resulting in an up count until overflow in the output signal.

In Square counter needs to pass a number to develop and signal meaning for half the max value the counter could reach, we outputted 0 and for the other half 1.

In Triangle when counter passes half of its value, the output signal starts decrementing until it reaches zero.

In Rhomboid we use the same strategy as triangle for the two sides of the rhomboid (the right and left side if we look at it from the very middle) and for each side when in odd numbers of Counter we mirror the output and so the signal is made.

In sine we follow tailor principle and using previous numbers for future ones. Sine starts from 0 and Cos starts from 1 (16'd30000) in this formula:

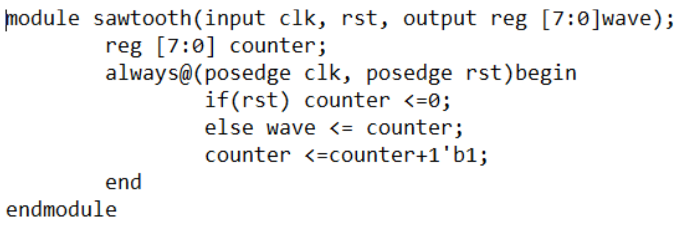
$$\text{Sine}(n) = \sin(n-1) + 1/64*\cos(n-1)$$
$$\text{Cos}(n) = \cos(n-1) - 1/64*\sin(n)$$

In arbitrary unit we simply get a signal from a ROM and display it.

For this part we used 3 keys from FPGA.

Only 2 of our signals worked perfectly (Square and Triangle) but others didn't give us a clean output.

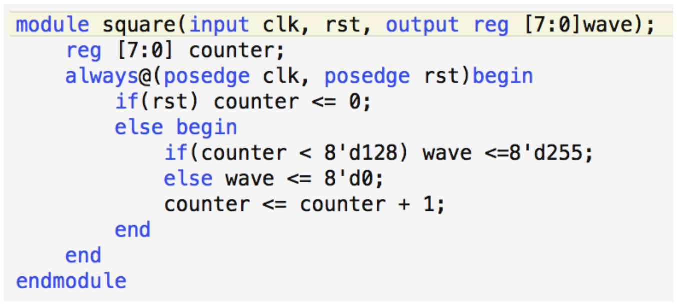
The sawtooth generation code: we want the output wave to start from 0 go up to its maximum value in a line like manner and then suddenly become zero, well this is exactly what an up counting counter does so we simply connected the counter to the output wave.



```
module sawtooth(input clk, rst, output reg [7:0]wave);
        reg [7:0] counter;
        always@(posedge clk, posedge rst)begin
                if(rst) counter <=0;
                else wave <= counter;
                counter <=counter+1'b1;
        end
endmodule
```

Figure 3 SawTooth

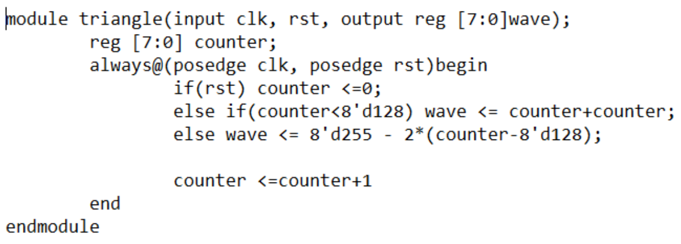
The square generation code: since the maximum value an 8 bit counter can have is 256 so half the value would be 128 so for the first half the output was the max value if could be and for the other half 0, creating a perfect square.



```
module square(input clk, rst, output reg [7:0]wave);
    reg [7:0] counter;
    always@(posedge clk, posedge rst)begin
        if(rst) counter <= 0;
        else begin
            if(counter < 8'd128) wave <=8'd255;
            else wave <= 8'd0;
            counter <= counter + 1;
        end
    end
endmodule
```

Figure 4 Square

The triangle generation code: we want the signal to start from 0 go to the max and then half way through the duty cycle come back down to 0 both in line like manners. so we wrote the following code that follows that logic:



```
module triangle(input clk, rst, output reg [7:0]wave);
        reg [7:0] counter;
        always@(posedge clk, posedge rst)begin
                if(rst) counter <=0;
                else if(counter<8'd128) wave <= counter+counter;
                else wave <= 8'd255 - 2*(counter-8'd128);

                counter <=counter+1
        end
endmodule
```

Figure 5 Triangle

The rhomboid generation code: the code was written using the explanation we said before in mind, the code can be seen bellow:

```
module rhomboid(input clk, rst, output reg [7:0]wave);
        reg [7:0] counter;
        always@(posedge clk, posedge rst)begin
                if(rst) counter<=0;
                else begin
                        if(counter < 8'd128) begin
                                if(counter[0]==1) wave <= 8'd127+counter;
                                else wave <= 8'd127-counter;
                        end
                        else begin
                                if(counter[0]) wave <= 8'd255 -(counter-8'd127);
                                else wave <=(counter - 8'd127);
                        end
                end
                counter <= counter+1;
        end
endmodule
```

Figure 6 Building Rhomboid

The sin generation code: as explained before we used tailors principle, the only thing worth mentioning here is that for the dividing part of the principle we used shift and sign extend.

```
module sinWave(input clk, rst, output [7:0] sinout);
        reg [15:0] cosprev, sinprev;
        reg[15:0] sinwave, coswave;
        always@(posedge rst, posedge clk)begin
                if(rst)begin
                        sinprev <= 0;
                        cosprev <= 16'd30000;
                end
                else begin
                        sinprev <= sinwave;
                        if(~(coswave==0)) cosprev <= coswave;
                end
        end
        always@(posedge clk, posedge rst)begin
                if(rst) begin
                        sinwave <=0;
                        coswave <=0;
                end
                else begin
                        sinwave <= sinprev + {{(6){cosprev[15]}}, cosprev[15:0]};
                        coswave <= cosprev - {{(6){sinwave[15]}}, sinwave[15:0]};
                end
        end
        assign sinout = sinwave[15:8];
endmodule
```

Figure 7 Buiding Sine

D. Amplitude Selector

In this part we divide the input taken from wave generator and divide it by 4 desired values changing from using 2 keys of FPGA.

```
module amplitudeSelector(input [1:0] func, input clk, input [7:0] inWave, output reg [7:0] outWave);
        always@(clk, func)begin
                if(clk) begin
                        case(func)
                                2'b00: outWave <= inWave;
                                2'b01: outWave <= (inWave>>1);
                                2'b10: outWave <= (inWave>>2);
                                2'b11: outWave <= (inWave>>3);
                        endcase
                end
        endendmodule
```

Figure 8 Amplitude Selector

E. Total Design

After compiling each part in Quartus, we make symbols of each parts and connect them using wires or buses.

We make inputs signals of clock and reset and entry keys in our block diagram. We have to make sure our block diagram is high level in Quartus.

After pin assignment and uploading code in FPGA the project is done.

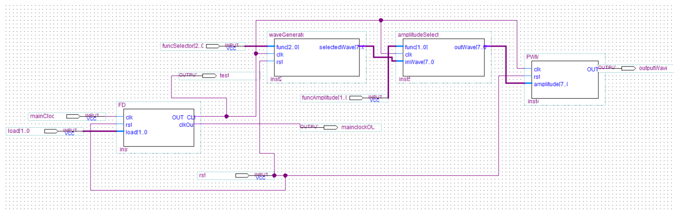


Figure 9 Block Diagram

IV. Conclusion

In this we learned about digital to analog converters, wave creation and frequency dividers.