# Experiment #4

Group 3

Nazanin Sabri
810194346
nazanin.sabrii@gmail.com

Nima Jarrahiyan
810194292
jarrahian.nima76@gmail.com

***Abstract*— working with Verilog, Audio processing, working with FPGAs, DAC, ADC, Echo Synthesizing Principles, Playing a Sound, FIFO Structure**

***Keywords*— DAC, ADC, FIFO, FPGA, Sound, Echo.**

## I. Introduction

In this experiment, we got familiar with the audio processing concepts using FPGAs. In order to do this we first learned about DAC and ADC converters in order to be able to process a signal, which was not digital inside the FPGA, and to see a digital signal outside the FPGA.

We then moved on to echoing sound signals using delays, which we were to be provided using a FIFO.

## II. Methodology and Procedures

Procedures we took to make this experiment work were as follows:

We first coded a comparator, we tried to pay attention to the fact that the number of bits the numbers which were meant to be compared had would change in time because in order to test it we were going to use 8 bit inputs but then it was to be used as an one bit comparator so we in order to support this functionality we made the unit parameterized.

After the comparator came the binary search unit. Our code as explained later is shown in figure 2, this was not our initial design however, because we first designed it so it would search for one number and then would not do much again unless it was reset, we soon learned that this implementation would not work because sound signals change very fast and manually resetting would not be humanly possible. The second thing we had initially done was making the binary search work with two signals, a greater than and an equal signal to make sure we reached the exact value, this did not work when connected to a physical comparator (part 2) though, because the comparator chip only had one output signal (greater than) so this was also something we had to change later on in the experiment.

In part two we were meant to input a sound signal and hear the exact input signal from a speaker connected to the FPGA output. We faced some issues in connecting the parts on the breadboard, connecting or not connecting the grounds; some voltage drops in different parts of the circuit and so we had to connect and reconnect the part till we were able to hear the sound signal. We still had some noise on the output signal but after asking we were told that noise was normal and to be expected.

In part 3 we were able to code and test the FIFO without any problems using the help of the code snippets provided in the laboratory description.

In the last two parts (single and multiple echo) we wrote the code and using a test bench were able to prove that the logic of our coding was right and without any problems but the test bench also showed why we couldn't hear the echo; the reason was that the delay between the sound signal and the echo signal (the delayed signal) was only 10 ns, meaning there was only 10 ns of delay and we all know nanosecond of delay is impossible for human ear to recognize and so there was a delayed signal it was just so close to the original signal that we were not able to differentiate the two.

To be able to hear it we needed to make this delay more (something in the lines of a couple of seconds instead of a couple of nanosecond) and to do this we had to make the size of out FIFO memory bigger so writing and reading from it would take more time but the FPGA did not have enough memory for us to be able to do that.

Some issues we faced and solved to our best ability were: the noise, which by electrostatic discharging capacitor, we noticed that in some point of charging capacitor the noise would be cancelled so we increased the size of capacitor. Another problem was not having steady 5 volt on the other side of level converter. Increasing the output resistance would cause steady current building higher voltage (5v).

## III. Results

### A. Part One

In this part we learned about different implementation methods of an ADC converter. The method we were to implement was the one using a binary search unit. We first designed a comparator shown in figure 1.

```
module comparator(
  A, B,
  A_ls_B,
  A_gt_B,
  A_eq_B
);
  parameter SIZE=16;
  input [SIZE-1:0] A, B;
  output A_ls_B, A_gt_B, A_eq_B;

  assign A_ls_B = (A<B)? 1:0;
  assign A_gt_B = (A>B)? 1: 0;
  assign A_eq_B = (A==B)? 1: 0;

endmodule
```

**Figure 1: comparing unit**

This comparing unit returns three less than, greater than and equal signals indicating the state of greatness between the two inputs, it is also parameterised so it can be used for any two inputs with any number of bits as long as they have equal number of bits.

We then coded the binary search unit as shown in figure 2.

```
always@(posedge clk)begin
counter <= counter+1;
  if(counter==4'd9) begin
    first <= 8'd0;
    last <= 8'b11111111;
  counter <= 4'd0;
  end
 /*if((temp+1 == mid) || (temp -1 == mid))begin
 first = 8'd0;
 last = 8'b11111111;
 end*/
  else begin
    if(~comp_eq)begin
      if(comp_gt)begin
        first <= mid;
      end else begin
        last <= mid;
      end
    end
  end

end

always@(negedge clk) begin
  sum <= ((first+last)>>1);
  mid <= sum[7:0];
end

assign out_val =  mid;
```

**Figure 2 : comparing unit**

This is how this code works: with every clock pulse a counter is counted (counter <= counter +1) and for 8 counts we search to find the number we were looking for and then the starting and the ending point of our search go back to the least and most possible values. (Meaning first becomes 0 and last becomes 255). In the 8 counts that we are searching if the greater than signal (comp_gt) is 0 it means that the number we are looking for is less than the value we are on now (this value is called mid in our code) so we search in the lower half the next time and so on and so forth.

We then put them together to be able to test the ADC as shown in figure 3.

```
module part1(clk, rst, Vin, data_out);

  input clk, rst;
  input [7:0] Vin;
  output [7:0] data_out;

  wire V_ls_bi, V_gt_bi, V_eq_bi;
  wire [7:0] binary_out;

  comparator #(8) cp(Vin, binary_out, V_ls_bi, V_gt_bi, V_eq_bi);
  binarySearchADC bs(V_gt_bi, 1'b0, clk, rst, data_out, binary_out);

endmodule
```

**Figure 3 : part one code**

## B. Part Two

In this part we were to connect the FPGA to an external comparator and a sound source (such as a mobile device) and hear the sound signal as an output of the FPGA.

In order to do this we had to design a number of components:

A PWM unit: this unit is used to convert an eight bit value to a one bit value, the logic of a PWM was explained to us in experiment 3. The code is shown in figure 4.

```
module pwm (
  clk,
  pwm_in,
  pwm_out
);

  parameter SIZE = 8;
  input clk;
  input [SIZE-1:0] pwm_in;
  output pwm_out;

  reg [SIZE-1:0] counter = {(SIZE){1'b0}};

  always@(posedge clk) counter <= counter +1'b1;


  assign pwm_out = (pwm_in > counter);

endmodule
```

**Figure 4 : PWM unit**

A display unit: this unit is used to show values on the 4 available 7-segments on the FPGA. The unit was tested and works fine with a low clock frequency it however fails to work with the FPGA's 50 MHz clock because the seven segments can't keep up.

```
module display(
  clk,
  data_in,
  segA, segB, segC, segD, segE, segF, segG, segDP,
  segA1, segB1, segC1, segD1, segE1, segF1, segG1, segDP1,
    segA2, segB2, segC2, segD2, segE2, segF2, segG2, segDP2,
  segA3, segB3, segC3, segD3, segE3, segF3, segG3, segDP3

);
  input clk;
  input [15:0] data_in;

  output segA, segB, segC, segD, segE, segF, segG, segDP;
  output  segA1, segB1, segC1, segD1, segE1, segF1, segG1, segDP1;
  output  segA2, segB2, segC2, segD2, segE2, segF2, segG2, segDP2;
  output  segA3, segB3, segC3, segD3, segE3, segF3, segG3, segDP3;
  sevenSegment ss1(data_in[3:0], clk, segA, segB, segC, segD, segE, segF, segG, segDP);
  sevenSegment ss2(data_in[7:4], clk, segA1, segB1, segC1, segD1, segE1, segF1, segG1, segDP1);
  sevenSegment ss3(data_in[11:8], clk, segA2, segB2, segC2, segD2, segE2, segF2, segG2, segDP2);
  sevenSegment ss4(data_in[15:12], clk, segA3, segB3, segC3, segD3, segE3, segF3, segG3, segDP3);


endmodule
```

**Figure 5 : display unit**

```
module sevenSegment(input[3:0] data_in, input clk, output segA, segB, segC, segD, segE, segF, segG, segDP);

  reg [7:0] segValues=8'b0;
  always@(data_in)begin               //is clock needed? would @data_in work?
    case(data_in)
      4'b0000: segValues <= 8'b11000000;
      4'b0001: segValues <= 8'b11111001;
      4'b0010: segValues <= 8'b10100100;
      4'b0011: segValues <= 8'b10110000;
      4'b0100: segValues <= 8'b10011001;
      4'b0101: segValues <= 8'b10010010;
      4'b0110: segValues <= 8'b10000010;
      4'b0111: segValues <= 8'b11111000;
      4'b1000: segValues <= 8'b10000000;
      4'b1001: segValues <= 8'b10011000;
      4'b1010: segValues <= 8'b10001000;
      4'b1011: segValues <= 8'b10000011;
      4'b1100: segValues <= 8'b11000110;
      4'b1101: segValues <= 8'b10100001;
      4'b1110: segValues <= 8'b10000110;
      4'b1111: segValues <= 8'b10001110;
    endcase
  end
  assign segA = segValues[0];
  assign segB = segValues[1];
  assign segC = segValues[2];
  assign segD = segValues[3];
  assign segE = segValues[4];
  assign segF = segValues[5];
  assign segG = segValues[6];
  assign segDP = segValues[7];

endmodule
```

**Figure 6: seven segment unit**

We also used the code from part one to be able to search for the received sound signal. The parts put together are shown below.

```
pwm #(SIZE) P(clk, data_out, pwm_out);
part1_analog #(SIZE) p1a(clk, 1'b0, Vin, data_out, binary_out);
display d1(clk, data_out, segA, segB, segC, segD, segE, segF, segG,
segDP,
segA1, segB1, segC1, segD1, segE1, segF1, segG1, segDP1,
segA2, segB2, segC2, segD2, segE2, segF2, segG2, segDP2,
segA3, segB3, segC3, segD3, segE3, segF3, segG3, segDP3);
```

**Figure 7: part 2 code**

Here is how the code works: Vin is the output wave of the physical comparator unit, this is given to the part1 code which then gives it to the binary search unit and the data_out is an 8 bit value that is the output of the binary search unit this value is given to display and PWM to be able to be printed out on the seven segments and to be given as an output of the FPGA.

# C. Part 3

In this part we have built a FIFO memory. In FIFO each data that is entered a pointer known as a write pointer is increased making the illusion of doing down in memory and sort of pushing to one memory space and if the data is read, another pointer called a read pointer is increased sort of like it's popped out of memory. This type of memory is not suitable for keeping data for long term but only to use in short periods.

Two control signals called "FULL" and "EMPTY" keep us notified when FIFO pointer have reached the beginning or the end of memory. Simply put if the FIFO is full or empty.

FIFO and other types of megafunctions can be created using Quartus wizard project but as laboratory objective we built one.

Beside these control signals, we have implemented CS control signal to almost implement a life like use of memories by selecting them. CS is short for Chip select which is used when more than one memory is connected or a when only parts of the CPU address are to be referenced in the memory.

Inside FIFO we have two parts. One is the main body of FIFO, which is dedicated to control signals. Another part is a RAM module which keeps the data and has the control signals as inputs and takes the data to be written or the data that has been read.

When Read or Write and CS signals are activate, RAM would use the address data to write to or read from memory.

```
38  //write pointer
39  always@(posedge clk or posedge rst)begin
40    if(rst)begin
41      wr_pointer <= 0;
42    end else if(wr_cs && wr_en)begin
43      wr_pointer <= wr_pointer +1;
44    end
45  end
46
47  //read pointer
48  always@(posedge clk or posedge rst)begin
49    if(rst)begin
50      rd_pointer <= 0;
51    end else if(rd_cs && rd_en)begin
52      rd_pointer <= rd_pointer +1;
53    end
54  end
55
56  //read data
57  always@(posedge clk or posedge rst)begin
58    if(rst)begin
59      data_out <= 0;
60    end else if(rd_cs && rd_en)begin
61      data_out <= data_ram;
62    end
63  end
64
65  //status counter
66  always@(posedge clk or posedge rst)begin
67    if(rst)begin
68      status_cnt <= 0;
69    end else if((rd_cs && rd_en) && !(wr_cs && wr_en) && (status_cnt!=0))begin
70      status_cnt <= status_cnt -1;
71    end  else if( !(rd_cs && rd_en) && (wr_cs && wr_en) && (status_cnt!= RAM_SIZE))begin
72      status_cnt <= status_cnt +1;
73    end
74  end
```

**Figure 8: FIFO unit**

```
//memory write
always@(address_0 or cs_0 or data_0 or we_0 or we_1 or address_1 or cs_1 or data_1)begin
  if(cs_0 && we_0)begin
    mem[address_0] <= data_0;
  end else if(cs_1 && we_1)begin
    mem[address_1] <= data_1;
  end
end

//memory read 0
always@(address_0 or cs_0 or we_0 or oe_0)begin
  if(cs_0 && !we_0 && oe_0)begin
    data_0_out <= mem[address_0];
  end else begin
    data_0_out <= 0;
  end
end

//memory read 1
always@(address_1 or cs_1 or we_1 or oe_1)begin
  if(cs_1 && !we_1 && oe_1)begin
    data_1_out <= mem[address_1];
  end else begin
    data_1_out <= 0;
  end
end

//tri state output
assign data_0 = (cs_0 && oe_0 && !we_0)? data_0_out:8'bz;
assign data_1 = (cs_1 && oe_1 && !we_1)? data_1_out:8'bz;

endmodule
```

**Figure 9: RAM unit**

Writing and reading entire memory of our FIFO would take 10ns, which is pretty fast, but we were unable to increase the delay due to FPGA not allowing memories having more than 13 address bits.

## D. Part 4

In this part we have 2 different methods to build Echo:
1. Simple echo using a simple delay by writing and reading from FIFO and add its lowered version to original audio.
2. Multiple echo-using differentials between in and out audio by rewriting in FIFO the lowered version of audio.

As mentioned, digital version of audio signal is entered single Echo and is changed using MAC by being multiplied and added to itself. FIFO generates the delay needed and USE_FIFO generates control signals required for FIFO.

To do this part the code is shown as below:

```
pwm #(SIZE) P(clk, echo, pwm_out);
singleEcho #(SIZE) processor(data_out, beta, clk, rst, echo);
part1_analog #(SIZE) p1a(clk, 1'b0, Vin, data_out, binary_out);
display d1(clk, data_out, segA, segB, segC, segD, segE, segF, segG,
segDP,
segA1, segB1, segC1, segD1, segE1, segF1, segG1, segDP1,
segA2, segB2, segC2, segD2, segE2, segF2, segG2, segDP2,
segA3, segB3, segC3, segD3, segE3, segF3, segG3, segDP3);
```

**Figure 10: Including Echo**

The logic of this code is: most of the parts are as explained in part 2, with the difference that the output of the binary search unit is the input of the echo unit and the echoed value is put in echo and given to the PWM unit shown above.

```
use_fifo uf(clk, rst, empty, full,wr_en,rd_en);
FIFO #(SIZE, ADDR_SIZE) fifoTest(clk, rst, 1'b1, rd_en,  1'b1, wr_en, pwm_out, fifo_data_out, empty, full);
mac #(SIZE) mac1(pwm_out, beta, mult_data_2, data_out);
```

**Figure 11: inside the singleEcho unit**

Use FIFO generates the write and read enable signals that are basically the FIFO control signals.

Figure 12 shows how the logic of use_fifo works.

The difference between single and multiple echo is in the inputs and function of Mac (multiply and accumulate) unit. Figure 13 shows how multiple echo logic is meant to work. The mac unit for single echo is shown in figure 14 and for the multiple echo addition is turned to subtraction.

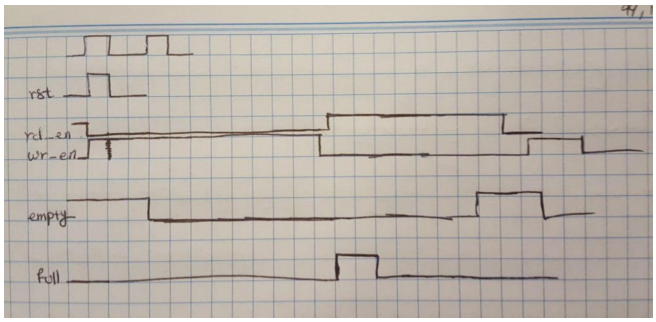


**Figure 12: How FIFO control signals work**

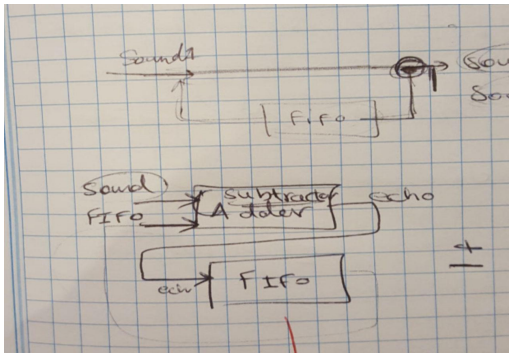


**Figure 12: Single and multiple echo**

```
always@(mult_data_1 or mult_data_2)begin
case(mult_data_1)
  2'b00: mult_result <= 0;
  2'b01: mult_result <= mult_data_2 >>2;
  2'b10: mult_result <= mult_data_2 >>3;
  2'b11: mult_result <= mult_data_2 >>4;
  endcase
end

assign data_out = add_data_1 + mult_result[SIZE-1:0];
```

**Figure 14: Mac of single echo**

## IV. Conclusion

In this experiment we learned about sound signals. How to input them and output them from the FPGA.