Department of Electrical Engineering
University of Cape Town

EEE4120F
High-Performance Embedded Systems
YODA Project
Audio Min-Maxing with FPGA

Team Leader	Christal Sima	SMXCHR002
	Motivates, organizes, and ensures that the team is meeting expectations regarding the project.	
Co-ordinator	Joyce Ejumu	EJMJOY001
	Puts together and proofreads the final documents before submission.	
Secretary	Phalo Mathungana	MTHPHA010
	Takes minutes and reads them before following meeting.	
Archivist	Tyran Naidoo	NDXTYR006
	Archives minutes, simulation files and all other important documents used throughout this project.	

Introduction

A crucial factor in all FPGA implementation is simulation and this YODA project is no different the purpose of this status report is to detail the progress made by this team in terms of data sampling, data analysis and tracking output. At this stage this has all been done using Vivado an FPGA simulator where you can use Verilog to program your device and see your input, see the effect of each module on that input and the output itself. In the hopes finding the Min-Max amplitude and potentially the interval-based min-max amplitude.

Overview of progress

Following on from Blog proposal where it concluded the board to be used would be a Nexys A7. Given that Vivado does not have a Nexys A7 to simulate with a compromise was made to use a Nexys 4 DDR board. It is similarity to the FPGA used was why it was chosen.

3 x Things Done

Written algorithm

The algorithm to be used for the purpose of the project has been written. This includes a method of calculating the minimum, maximum, mean and standard deviation of an input signal. This forms the basis for implementation on the FPGA as well as the development of a golden measure/benchmark.

The following modules have been made to use for the FPGA board.

Min_max: This module takes in an array of data, and a clock signal, and then outputs the minimum and maximum values of the data array. The algorithm it follows is to initially set their values to the first element of the data array. Then, the module iterates through the data array at each clock cycle. While doing this, it compares the current data value to the minimum and maximum values. Should the current array element be greater than the maximum value, or less than the minimum value, the minimum or maximum values are replaced. This is done using an if statement.

Mean: This module takes in an array of data, and a clock signal, and will calculate the mean of the data at each clock cycle. For the purposes of calculating the mean, three integer variables are used, mean, sum, and count. Mean will keep track of the mean value of the data. Sum continuously sums the data from the array during each clock cycle. Finally, counter keeps track of how many data points have been added to the sum. The mean, sum and counter variables are initially set to zero, and change at each clock cycle. The mean variable is set to the sum divided by the counter.

Std: This module uses the mean value calculated in the mean module and the clk as input to calculate the standard deviation of each of the points of this sine audio wave. How is does this is by breaking up the formula bit by bit. It finds the x_value by using the reading a value of the sine wave off the memory then reducing it by the mean value. From that point it continues the rest of the formula as shown in the next section. The standard deviation here is calculated at every positive clock edge. This is to make sure that it is coordinated with the other modules (the mean module in particular).

Made the outline of the testbench using a sine wave input

The current focus has been to create working modules that are able to process data. To do this, instead of a sound wave, a simulated sine wave has been used. The sine wave is generated using a lookup able to sine_gen module. The testbench generates this sinewave and then uses the min_max, mean and std modules to calculate the minimum, maximum, mean and standard deviation of data.

Research

Research into the different FPGAs and which would be appropriate

Nexys4

This is a development board based on the Atrix-7 FPGA. It has various USB, Ethernet, and other ports. The Nexys board can also host many unique designs of combinational circuits, powerful embedded processors, and various other applications. In addition to all of this, this board has built in peripherals such as an accelerometer, temperature sensor, and digital microphone a speaker amplifier amongst others.

Nexys A7

This development platform is based on the latest Artix-7 FPGA. It is a large high-capacity FPGA with external memory, and various ports. It also features an accelerometer, temperature sensor, digital microphone, speaker amplifier and other input out devices. Like the Nexys4, it can be used for a wide range of applications without the need for other components.

Deciding on which FPGA board to use.

Research was also done into the Nexys2 and Nexys3 boards. While they may be the more cost-effective option, they are the more difficult boards to find or simulate. Therefore, we limited the decision to be between the Nexys4 and Nexys A7. While each board presents their own merits. These include usability, resources available or price of the FPGA. The Nexys A7 has more available features than the Nexys4. This does come at an excessive cost. But because both board options were presented to use, we have decided to with the Nexys A7 due to its range of features.

Research into inputting an audio signal into an FPGA

Research was done into looking how the chosen board from above, the Nexys A7, can take it audio data, as well as the appropriate bitrate and whether any alternatives to the regular audio in port can be found. Looking at the concept of receiving audio data on an FPGA, it is usually done using a board that already sports a serial audio in port. Our chosen board does not have this port however, alternatively it has a microphone directly on the board. We decided this would not be an effective way to receive and process audio mostly because getting specific data into the board would be difficult. Instead, we opted to look into alternative ways of getting data onto the board. The options we had at our disposal were the other two data ports, the ethernet and USB port. Due to the utility and dynamic nature of the USB port, we decided to use this method. The restrictions we have however involves getting the data to mimic the bitrate of a regular audio port, in this case a sampling rate of 48kHz. It would be better however if we could go beyond this bitrate, which would improve be the overall speed that we can process data, since we can avoid any bottlenecks that would occur on a regular audio input port. Further research is being done into how to mimic this on the testbench and how to utilize the FPGA board to do this using Vivado and my extension Verilog.

3x Things to be Done

Write code for the FPGA

Further development of the code within Verilog to process data faster and streamline our code for easier deployment to the physical board. Also going to work on how to interface with the board via Vivado.

Sample an audio on Verilog

Attempt to use the above method to allow us to stream audio data into the FPGA, via the USB port. Configuring the ports and bitrate will be a part of this task as well.

Process an audio signal on Verilog

Utilizing code to process this incoming audio to make it easier to manipulate and apply the min max algorithm detailed below to.

**Construct Golden Standard / Benchmark

Subtask within processing, to create a golden measure to compare our method of processing to gauge the effective efficiency of our system over a conventional serial system.

Code and explanation for design thus far

(Refer to inline comments for explanations)

Modules

Minimum and maximum

The module to calculate the minimum and maximum values

```
module min_max(clk, samples, max, min);

input clk;                    // clock signal
output reg [15:0] samples;      //audio wave
output reg [15:0] min, max;     //minumum and maximum values of the audio

parameter SIZE = 1024;
reg [15:0] rom_memory [SIZE-1:0];
integer i;

initial begin                 //initial block, run once
   $readmemh("sine_LUT_values.mem", rom_memory); //Use IP of BRAM instead of this
command
   i = 0;
   max=rom_memory[0];
   min=rom_memory[0];
end

//At every positive edge of the clock, output a sine wave sample.
always@(posedge clk) begin      //repeats at each clock cycle on positive rise
   samples = rom_memory[i];
   if(i == SIZE)              //end of loop
      i = 0;                  //reset counter to 0
```

```
    if (min>rom_memory[i])      //checking the minimum value against the current element
        min=rom_memory[i];      //if the current minimum values is greater than the current element,
the element will now replace the min value
    if (max<=rom_memory[i])     //checking the maximum value againt the current element
        max=rom_memory[i];      //if the current maximum values is greater than the current element,
the element will now replace the max value
    i = i+ 1;                   //increment the counter for the next cycle
end
endmodule
```

Mean

The module to calculate the mean of an array of data

```
module mean(                    //module responsible for calculating the mean of a set of data
    input clk,                  // clock signal
    output reg [15:0] mean      //mean output
    );

parameter SIZE = 1024;          //the size of the array of sine signals
reg [15:0] rom_memory [SIZE-1:0];   //the array where the sine signals are stored. This is the data
acessed in the code
integer i, sum,x_value, count;      //i is used to access array elements, x_values is the current array
element, count is the number of elements that have been acessed

initial begin                   //initial block, run once
    $readmemh("sine_LUT_values.mem", rom_memory);   //could use IP of BRAM instead of this
command
    i = 0;                              //initialise to 0 to access first element in array
    mean=0;                             //set mean to zero, will change during simulation
    sum=0;                              //set sum to zero, will increase using cumulative addition
    x_value=0;                          //set to zero in begining, will take on array element values
during each clock cycle
    count=1;                //number of elements accessed, used as N in mean and std formula,
increased during each clock cycle
end

always@(posedge clk) begin              //repeats at each clock cycle on positive rise
    x_value = rom_memory[i];            //set x_value of array element
    sum = sum + x_value;                //cumulative addition of x_value

    if(i == SIZE)

    i = i+ 1;                           //increment the counter for the next cycle
    count=count+1;                      //increment no of accessed elements during each clock cycle

mean=sum/count;                         //finally, calculate the mean of a set of data
end
endmodule
```

Standard deviation

```
//the following module calculated the standard deviation of a set of data
module std(
```

```
    // takes an input of the clock signal and mean calculated using the mean module
    input clk, mean,
     output reg [15:0] std                  // outputs the calculated standard deviation
    );

parameter SIZE = 1024;                      //the size of the array of sine signals
reg [15:0] rom_memory [SIZE-1:0];           //the array where the sine signals are stored. This is the
data acessed in the code
integer i, std_sum,x_value, count,top,top_sqrd,inside_brackets; //i is used to access array elements,
x_values is the current array element, count is the number of elements that have been acessed

//squareroot function
    reg [15:0] sqr;

    //Verilog function to find square root of a 32 bit number.
    //The output is 16 bit.
    function [15:0] sqrt;
        input [31:0] num;  //declare input
        //intermediate signals.
        reg [31:0] a;
        reg [15:0] q;
        reg [17:0] left,right,r;
        integer i;

    begin
        //initialize all the variables.
        a = num;
        q = 0;
        i = 0;
        left = 0;   //input to adder/sub
        right = 0;  //input to adder/sub
        r = 0;  //remainder
        //run the calculations for 16 iterations.
        for(i=0;i<16;i=i+1) begin
            right = {q,r[17],1'b1};
            left = {r[15:0],a[31:30]};
            a = {a[29:0],2'b00};    //left shift by 2 bits.
            if (r[17] == 1) //add if r is negative
                r = left + right;
            else    //subtract if r is positive
                r = left - right;
            q = {q[14:0],!r[17]};
        end
        sqrt = q;   //final assignment of output.
    end
    endfunction //end of Function

initial begin                   //initial block, run once
    $readmemh("sine_LUT_values.mem", rom_memory); //could use IP of BRAM instead of this
command
    i = 0;              //initialise to 0 to access first element in array
    std=0;              //set std to zero, will change during simulation
    count=1;            //number of elements accessed, used as N in mean and std formula, increased
during each clock cycle
```

```
    //NOTE: The following variables are for the purposes of calculating the std of the data. Their
names are more descriptive of their place in the formula of std calculation
    x_value=0;          //set to zero in begining, will take on array element values during each clock
cycle
    top=0;              //describes x_value-mean
    top_sqrd=0;         //describes (x_value-mean)^2
    std_sum=0;          //describes sum((x_values-mean)^2)
    inside_brackets=0;  //describes sum((x_value-mean)^2)/N
end

always@(posedge clk) begin              //repeats at each clock cycle on positive rise
    x_value = rom_memory[i];            //set x_value of array element
    top=x_value-mean;                   //x_value-mean
    top_sqrd=top*top;                   //(x_value-mean)^2
    std_sum=std_sum+top_sqrd;           //sum((x_values-mean)^2)
    i =  i+ 1;                          //increment the counter for the next cycle
    count=count+1;                      //increment no of accessed elements during each clock cycle
    inside_brackets=std_sum/count;      // sum((x_value-mean)^2)/clk
    std = sqrt(inside_brackets);        //finally, calculate the std of a set of data

    if(i == SIZE)
        i = 0;
end
endmodule
```

Test Bench

The test bench thus far

```
`timescale 1ns / 1ps

module testBench;
    reg clk;
    wire [15:0] sine;
    wire [15:0] Input_Data; //may be a reg
    wire [15:0] Min_Output;
    wire [15:0] Max_Output;
    wire [15:0] Mean_Output;
    wire [15:0] Std_Output;

    //initates and connects the sine generator to the testBench
    sine_gen baseSineGen(.clk (clk), .sineOutput (sine));
    min_max samp(.clk (clk),.samples (Input_Data), .min (Min_Output), .max(Max_Output));
    mean mean1(.clk (clk),.mean(Mean_Output));
    //std std1(.clk (clk),.mean(Mean_Output), .std(Std_Output));


    //frequency control
    parameter freq = 100000000; //100 MHz
    parameter SIZE = 1024;
    parameter clockRate = 0.2; //clock time (make this an output from the sine modules)

    //Generate a clock with the above frequency control
    initial
    begin
```

```
    clk = 1'b0;
    end

    always #clockRate clk = ~clk; //#1 is one nano second delay (#x controlls the speed)

endmodule
```

Evidence of work done

Simulation

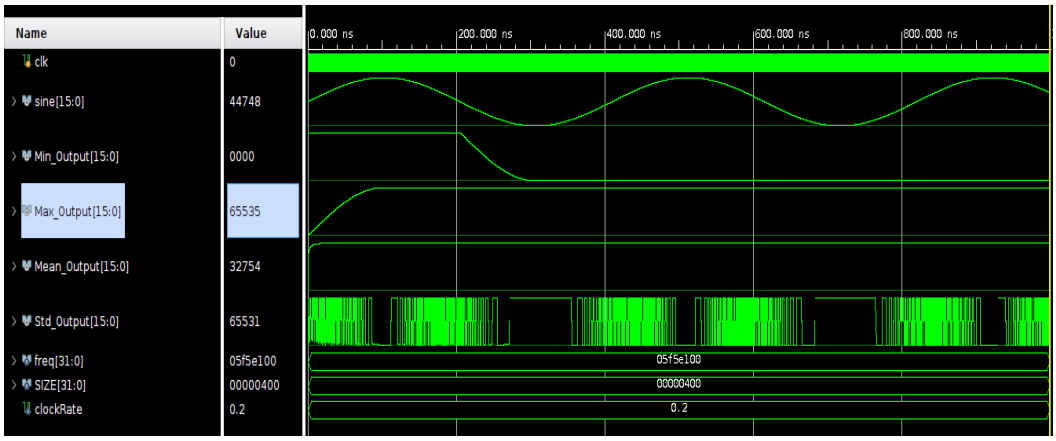
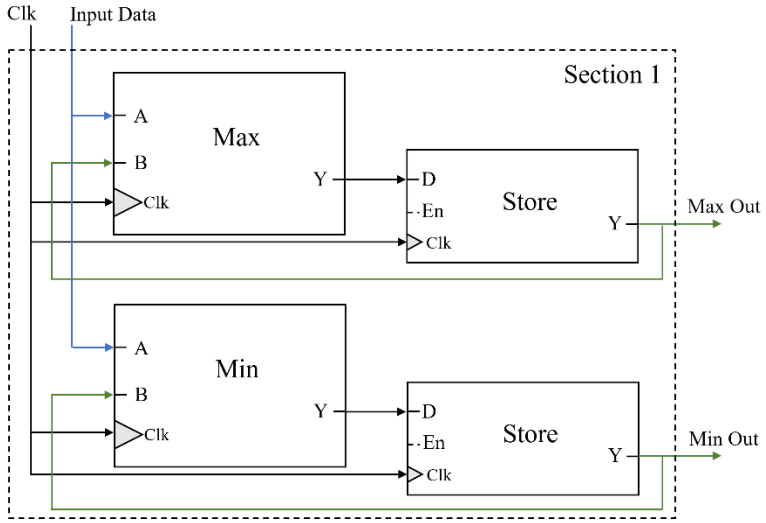


Figure 1: Simulation of above Testbench

Block Diagrams

Higher Behaviour of Section 1



2: Block Diagrams of inner block behaviour for minimum and maximum calculation

Lower Behaviour of Section 1

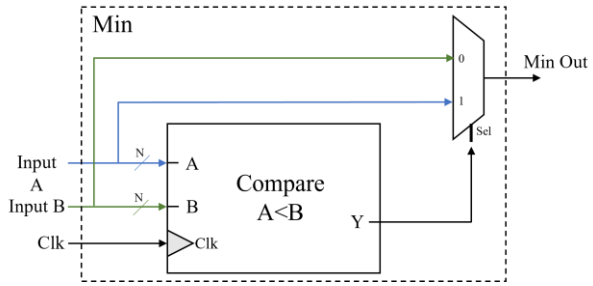


Figure 2: Block Diagrams of inner block behaviour for minimum calculation

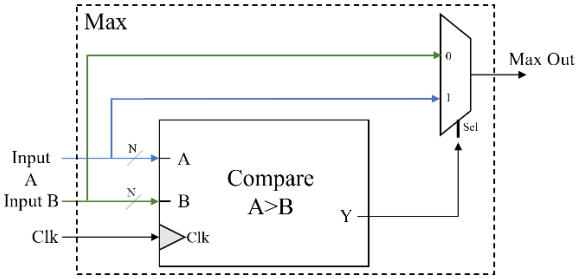


Figure 3: Block Diagrams of inner block behaviour for maximum calculation