EENGM4120_2022_TB-2

Advanced DSP & FPGA Implementation 2022

Coursework

University of Bristol

Department of Electrical and Electronic Engineering

Teacher:	Dr. Roshan Weerasekera
Group:	05
Student 1:	Qingyu Zhang (vn22984)
Student 2:	Shuran Yang (rw22242)
Student 3:	Ruilong Liu (hx22195)
Date:	May 2023

The project link:

https://github.com/alfredzhang98/Bristol_FPGA_lab.git.

Catalogue

1 Simple FIR Filter (RTL)

1.1 MATLAB simulation

The purpose of this MATLAB simulation is to realize a FIR filter, which can filter a sinusoidal signal with additional noise. This report will explain how the sinusoidal signal with additional noise is generated and the role of the “linspace” function in generating the signal. In addition, the report will explain the algorithm code for implementing a 7th order FIR filter using a rectangular window. Finally, the report will give the filter signal diagram, by comparing the original signal, FPGA output signal and MATLAB simulation output signal to evaluate the filtering effect of the fir filter. Also, pole-zero graphs, and frequency response graph are generated to evaluate the performance of the fir filter.

First, define the signal frequency, sampling number and bit width. Then, by generating a time vector using the “linspace” function, the sine wave signal containing noise is generated and normalized to a 10-bit binary value. The “linspace” function is used to create a time series whose input parameters are the start time, end time, and number of sampling points. According to these parameters, a set of sampling points are generated at equal intervals within the specified time range. Finally, by combining this time series with the functions of "sin" and "rand", the sinusoidal signal "x" is calculated to generate the signal. Eventually, in this MATLAB code, plus a piece of code given by the teacher, a ROM file, named "ROMData.mem", is generated that contains the hexadecimal representation of the input signal for the FPGA. This ROM file will be used for subsequent actual FPGA development. When talking about the algorithm code, this simulation uses a rectangular window to design a 7-order fir filter to filter the input signal. First, the standard difference equation is used to express the FIR filter algorithm, and then the "my_conv ()" function in MATLAB is used to realize the convolution operation, and the coefficient of the filter and the input signal are convolved to get the output signal.

As shown in Figure 1.1.(a), the pole-zero diagram shows that six zeros of the 7th-order FIR filter are on the unit circle, and one pole is at the origin. This FIR filter structure is an "all-pass" filter, simple structure, easy to implement. Because of having linear phase characteristics, no group delay and frequency selectivity, it is very suitable for FPGA implementation.

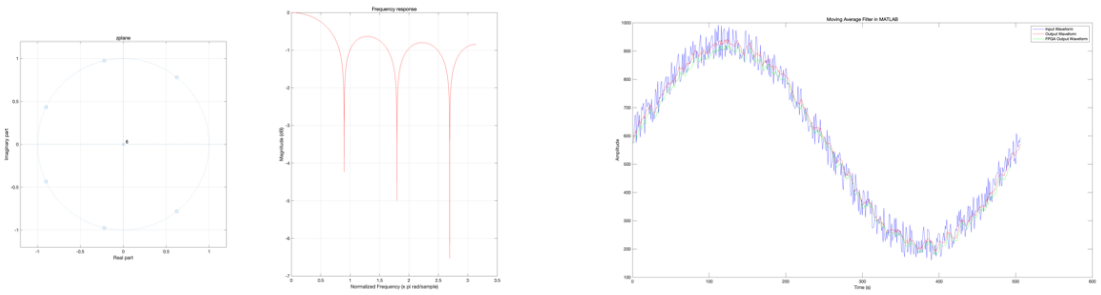

Figure 1.1 (a) Pole zero diagram and frequency response diagram, (b)Final filtering result.

As shown in Figure 1.1.(b), by observing the original signal, FPGA output signal and MATLAB simulation output signal, it can be concluded that the filtering performance of the filter is good.

1.2 Implementation in System Verilog

To implement FIR filter in System Verilog, this report provides the corresponding design scheme of the delay line, multiplier, and adder.

In the code, six delay lines, “z_1”, “z_2”, “z_3”, “z_4”, “z_5”, and “z_6” are used to store the history value of the input signal. The input signal “x” is delayed by 6 sampling cycles respectively, totaling 7 delayed signals. The weighted sum of each delay signal gives the output signal “y”.

The design of the multiplier is realized by shifting and adding. In the code, using a shift operation to reduce the number of bits in the input signal and then adding them together to get the product. For this filter with a coefficient of 0.149 (fixed-point (Q0.8) format is 0 00100100), using the following Equation 1.1 to design multiplier:

$$\text{assign } mal_0 = (x \gg 3) + (x \gg 6) + (x \gg 7) \tag{1.1}$$

In the code, moving the input signal "x" 3, 6, and 7 bits to the right to get the three shifted results respectively (Because the binary representation of 0.1429 is 0 00100100), and then adding them up to get the product. Unlike the way it's calculated in the code, typical multiplication in hardware is repeated addition. So, in the process of designing adder, by adding the product of fixed points in each delay line and following Equation 1.2 to calculate the output signal:

$$\text{assign } y = mal_0 + mal_1 + mal_2 + mal_3 + mal_4 + mal_5 + mal_6 \tag{1.2}$$

In summary, a method like the shift adder is used to implement multiplication and addition elements. This design can reduce delay and area consumption and is suitable for scalable FIR filter design.

1.3 Testing and Verification

1.3.1 Verification Plan

The objective is to test and verify the implemented FIR filter. To meet the requirement, some modules need to be added around the FIR filter to generate simulation test data. These modules include the address generator and ROM module, as shown in Figure 1.2. Among those, the FIR Filter module is the System Verilog module designed in Task 2, which function is filtering the signal. And the ROM module is generated by the MATLAB code in Task 1, which function is to store the signal generated from MATLAB simulation.

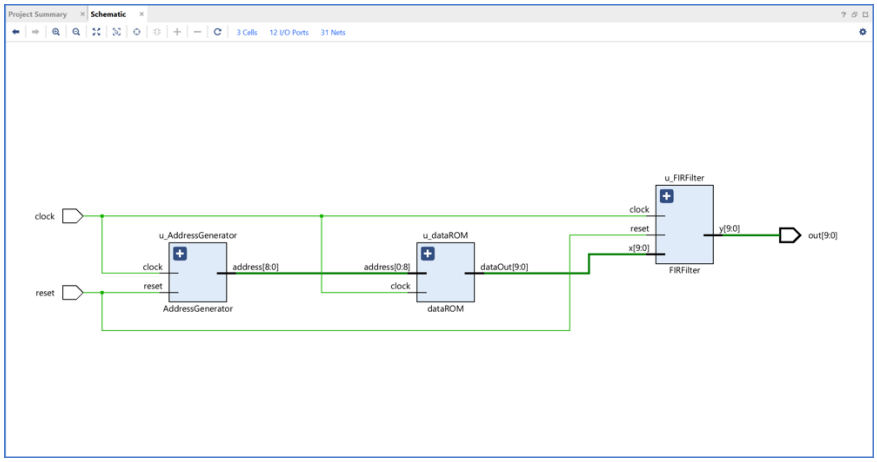

Figure 1.2. Block diagram for the FIR Filter Test setup.

The above hardware block diagram clearly indicates the input and output mode of each signal: The whole design process needs the control of clock signal and reset signal. More specifically, the clock signal is used to drive the operation of the entire system, and the reset signal initializes the address counter. To ensure that the MATLAB simulation results are consistent with the HDL simulation results, this testing uses the same input waveform as in Task 1. When the input signal is loaded from the dataROM module, the address register points to the first storage unit. The dataROM module reads data from this storage unit and sends it to the FIRFilter module. The FIRFilter module filters the input data and sends the result to the output port. The output of the FIRFilter module will be read and displayed in the simulation waveform using Vivado. At the same time, the test module also updates the address register when the clock pulse arrives.

1.3.2 Verification Results

Once the test vector input is connected to the FIR filter, the waveform of the filter can be viewed by using Vivado simulation function. During this process, the code needs a testbench file to simulate the design. In the testbench file, after instantiating the corresponding modules for the FIR filter and ROM blocks, they are wired together.

The principle of the initial block of the testbench file is first initializing the clock and reset signal, and pulling the reset signal down after a certain amount of time to ensure that the design is running from the correct state. Each time the clock signal rises or resets the rise edge of the signal, the address counter will be increased by 1 until the ROM's maximum memory location is reached. When the address counter reaches its maximum position, the address counter will be reset to 0. In this way, it can generate a sinusoidal waveform over multiple periods.

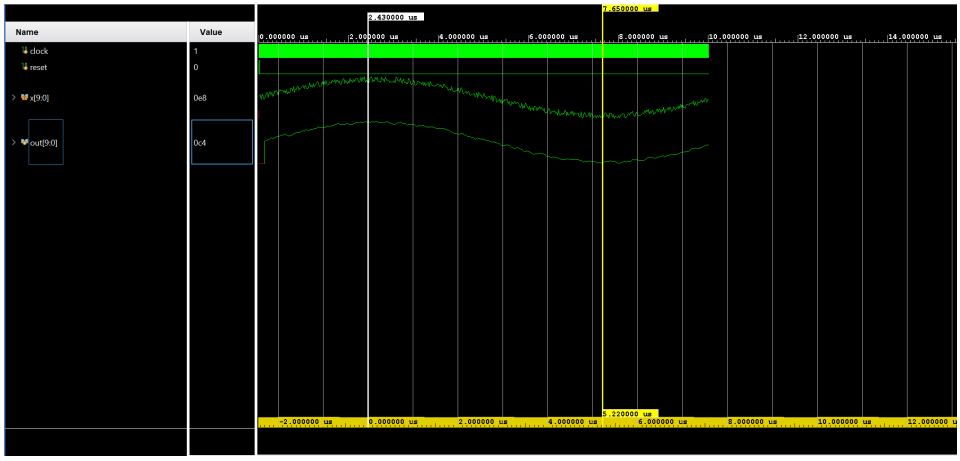

Figure 1.3. simulation result.

In the Vivado simulation, Figure 1.3 shows the sinusoidal waveform of the filter input and the waveform of the filter output. It is obvious that the filter successfully removes the high-frequency noise in the original sinusoidal waveform and smoothens it into a more stable waveform. The result proves that the FIR filter successfully realizes the filtering process of sinusoidal signal with noise with visible validity and reliability.

1.3.3 Optimization and analysis

First, the results in Task 1 show that the fir filter used in this experiment is a 7-order fir filter, not a 5-order FIR filter in the example pdf. The advantages of this move are obvious, as the use of higher order FIR filters can achieve better filtering results, especially for situations requiring higher cutoff frequencies and steeper filtering characteristics. However, higher order FIR filters also have some problems, including higher computing and resource consumption, including more multipliers and adders, resulting in higher power consumption and delay. In addition, too high an order may also lead to overfitting, so that the filter cannot adapt to the noise or nonlinear signal.

In the specific operation process, it is found that: when using Q0.8 format to calculate the data, an 8-bit decimal with signed bits can only represent 512 different values. Therefore, the accuracy may be insufficient when using the 7th-order FIR filter, especially when using the multiplier to convolve the filter coefficient and the input signal. Therefore, in this case, the best way to optimize is to use the Q16 format to provide higher accuracy, because the Q16 format can represent 65536 different values. Correspondingly, the way to adapt to the new data format is that changing the register bit width can increase the maximum number of bits of data processed by FPGA, thus improving the calculation accuracy.

By analyzing the hardware block diagram, due to the substantial number of modules, there may be some problems in the transmission process of internal signals, including timing problems, data bit width mismatch and line delay problems. For different problems, there may be corresponding optimization schemes. For example, to solve the problem of data bit width mismatch, a reasonable optimization scheme is to add a data width conversion module. The module converts the data bit width transmitted between different modules to ensure the consistency of data bit width. For the line delay problem, the reasonable optimization scheme is to add a cache, for example, by adding a FIFO cache before the input signal module. The special function of this cache is to store a certain amount of data in advance, so that the clock frequency of the input signal and the output signal can be uncoordinated when the data is processed in FPGA, to achieve a certain delay optimization.

2.1 Methodology

2.1.1 Matlab Background

The difference equation for the digital oscillator as:

$$\begin{cases} y[n] = 2\cos\left(\frac{2\pi F_0}{F_s}\right) y[n-1] - y[n-2] \\ y[-1] = 0 \\ y[-2] = A\sin\left(\frac{2\pi F_0}{F_s}\right) \end{cases} \tag{2.1}$$

Where F0 is signal frequency and Fs is the sampling frequency. A waveform with ten periods output using MATLAB is shown in Figure 4.1. Where F0 is 440 Hz, and Fs is 1000 Hz [1].

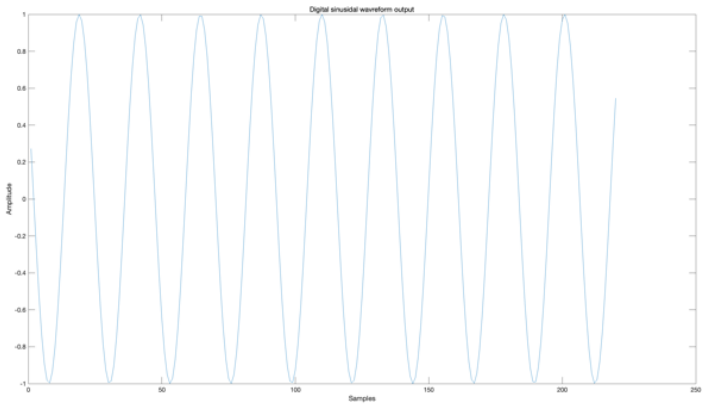

Figure 2.1. Digital filter sine wave output (ten periods).

2.1.2 Oscillator Implementation in SystemVerilog and Simulation

In this task, the amplitude (0.2730) and coefficient (0.9620) of a sine wave are calculated by Matlab. Since it is necessary to use the fixed-point (Q0.8) format, the amplitude of the sine wave is 00001000101 and the coefficient is 001111 0110, where the highest bit is the sign bit. For signed numbers, the use of shifting not only makes the logic of the code simpler but also ensures the reliability of the signal. The method of shift splicing is reflected in the code as follows, where 'y_temp_2[10]' is the sign bit.

$$\begin{aligned} \text{mul} = (1(y_2[10]), y_2[10:1]) + (2(y_2[10]), y_2[10:2]) + (3(y_2[10]), y_2[10:3]) \\ (4(y_2[10]), y_2[10:4]) + (6(y_2[10]), y_2[10:6]) + (7(y_2[10]), y_2[10:7]) \end{aligned} \tag{2.2}$$

In order to sample the DAC, the output sine wave value needs to be scaled by adding 10'd512 to make it fit within the input range of the DAC. The output waveform of the DigSineGenerator.sv module is shown in

Figure 2.2. The period is 2.24 ms and the frequency is 1/2.24 ms = 446.429 Hz, which meet the requirements of the question.

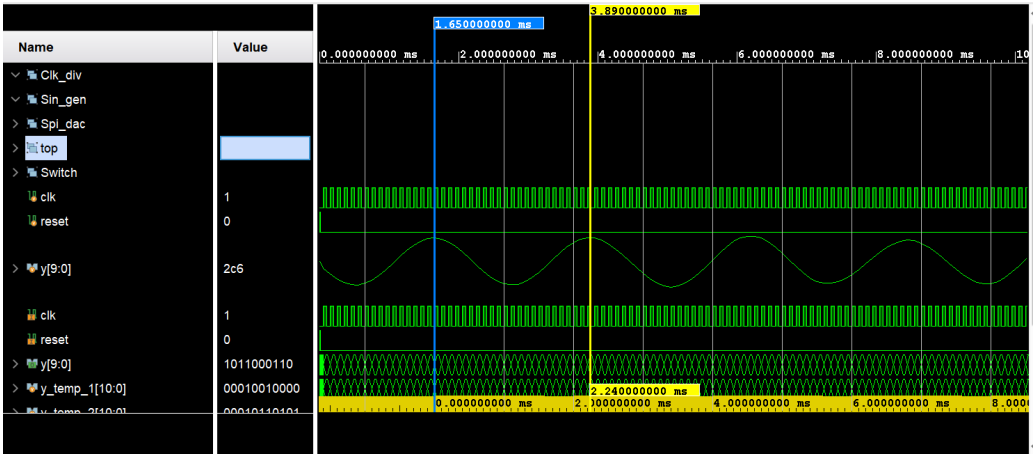

Figure 2.2. The waveform of DigSineGenerator.sv module output.

2.1.3 Implementing a Frequency-amplitude Adjustable Oscillator on FPGA

In this task, a total of seven modules are used, and their schematic is shown in Figure 2.3. The functions and relationships of each module are as follows.

1. U_SwitchInterface: A key with an anti-jitter function for adjusting the frequency and amplitude of the waveform. The output signal flag_out is passed to the next module (KeyFlagSettingValue).
2. KeyFlagSettingValue: This module accumulates the received flag_out signals and obtains the relative change in frequency and amplitude (valid_clk, valid_mag), which is passed to the next module (BottonChangeMgnitude, BottonChangeClk), so that the waveform can be adjusted in real time.
3. BottonChangeMgnitude: This module converts the signal received from valid_mag into a change in the amplitude of the waveform and outputs it to the DigSineGenerator module for real-time adjustment of the waveform amplitude. There is no overflow in the amplitude here.
4. BottonChangeClk: This module converts the signal received from valid_clk into a change in waveform frequency and outputs it to the DigSineGeneratorUpdate module for real-time adjustment of the waveform frequency or period. There is also no overflow of frequency here.
5. DigSineGeneratorUpdate: This module generates an initial sinusoidal signal waveform with 440 Hz and 3.3 volts and is controlled by the previous amplitude adjustment signal and frequency adjustment signal. Here, the signal is accepted by means of a handshake.
6. clkDivider: used to divide the 100 MHz clock and drive the spi2dac module. 7. spi2dac: used to divide the 100 MHz clock and drive the spi2dac module.
7. spi2dac: converts the spi signal into a dac signal for output.

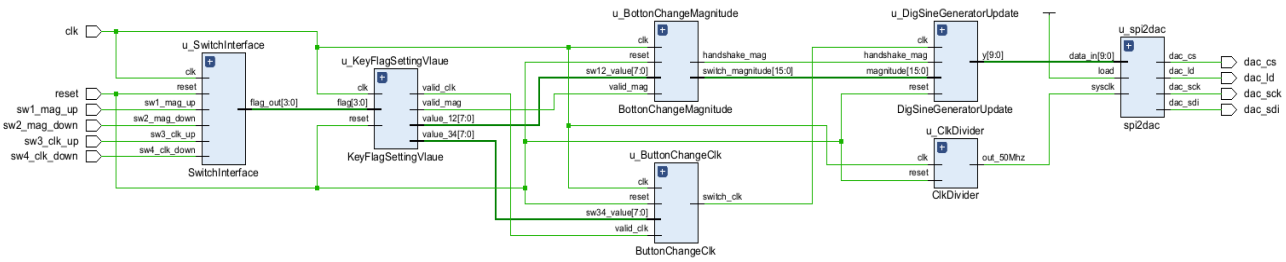

Figure 2.3 Digital Oscillator Schematic.

BottonChangeMgnitude and BottonChangeClk control the DigSineGeneratorUpdate module by using the handshake method as in Figure 2.4. The data from the BottonChangeMgnitude module and the clock period of the BottonChangeClk module together determine the state of the handshake_save1 signal. module and the clock period of the BottonChangeClk module together determine the state of the handshake_save1 signal. It is worth noting that in order to obtain a signal for controlling the data transfer, handshake_save1 and handshake_save2 need to be XOR, where the handshake_save2 signal is the inverse of the handshake_save1 signal.

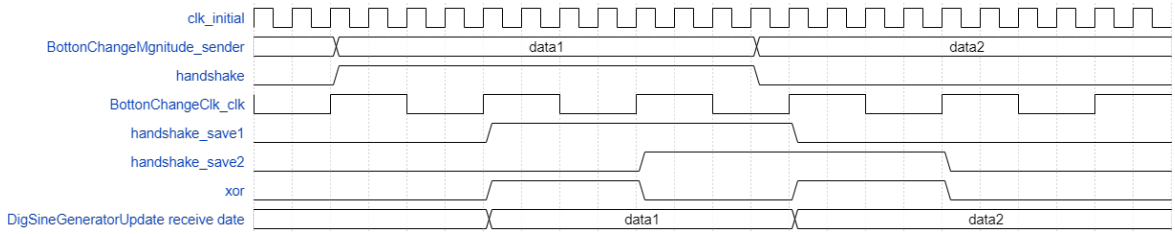

Figure 2.4. Illustration of the timing relationship between the control methods for the amplitude and frequency of the signal.

From Figure 2.3, there will have four output signals from ZedBoard to MCP4911 chip. The Package Type of MCP4911 as shown in Figure 2.5. Table 1 shows not only the ZedBoard pins and the Pmod connections of the MCP4911 chip but also the reset button of the system and the switches for signal frequency and amplitude adjustment [2].

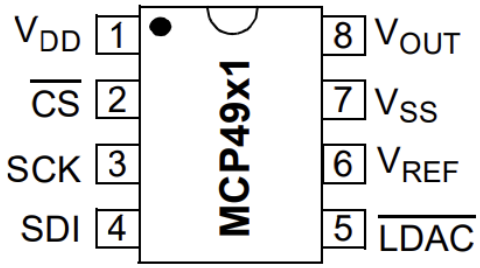

Figure 2.5. Package Type of MCP4911 [2].

Table 1. Task Connections

ZedBoard Pin [3]	Signal Name	MCP4911 Chip
VDD	VDD	1
JA2	$\overline{CS}$	2
JA3	SCK	3
JA1	SDI	4
JA4	$\overline{LDAC}$	5
VDD	VREF	6
GND	VSS	7
	VOUT	8
BTNC	reset	
BTND	sw2_mag_down	
BTNL	sw3_clk_up	
BTNR	sw4_clk_down	
BTNU	sw1_mag_up	

2.2 Simulation Waveforms in Vivado

Table 2 and Figure 2.6-7 describe the adjustment range of the phase and magnitude of the signal. The design of this section enables the amplitude and frequency of the signal to be adjusted according to the times of button presses.

Table 2. Adjustment range for the value of analogue waveforms.

Function	Figure	Range	Simulation Value
Frequency	2.7 (a)	Max	44444.44 Hz
	2.6	Initial	444.44 Hz
	2.7 (d)	Min	232.56 Hz
Amplitude	2.7 (c)	Max	912 (3.3 V)
	2.6	Initial	704 (1.58 V)
	2.7 (d)	Min	512 (0 V)

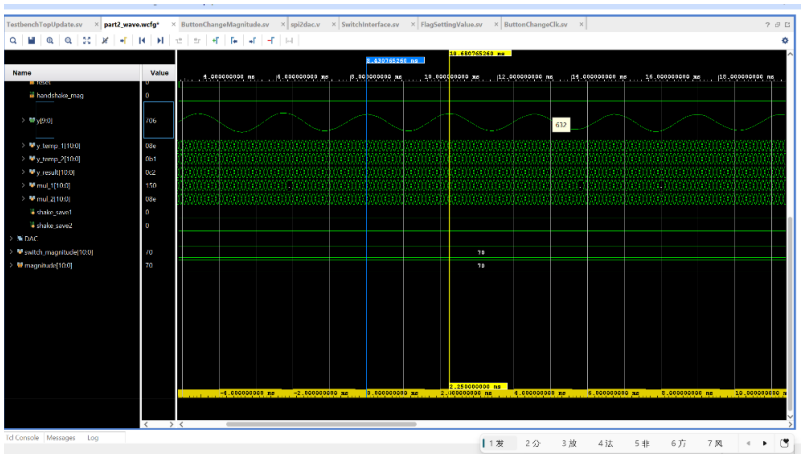

Figure 2.6. Output waveform of the initial frequency and amplitude signal.

Figure 2.7. Output waveform of the (a) maximum frequency signal, (b) minimum frequency signal, (c) maximum magnitude and (f) minimum magnitude.

2.3 The Outputs Waveforms of DAC Signals with Adjustable Magnitude and Frequency

The testing of the output waveforms for this project was done on the laboratory RTB2004 Digital Oscilloscope. As explained earlier, the parameters in this section are passed in a handshake fashion; frequency and magnitude can be adjusted by pressing buttons (Refer Appendix 3.2 and 3.3 for a detailed code implementation). Table 3 and Figure 2.8 depict the waveform schematics of the DAC signal's phase and amplitude adjusted for frequency and amplitude on an oscilloscope. For a clearer presentation of the waveform, the waveform given in Figure 9 is not the limit of adjustment, and in fact, the frequency and period of the waveform can be changed by pressing buttons larger or smaller.

Table 3. Adjustment range for the value of oscilloscope outputs waveforms.

Function	Figure	Range	Test Value
Frequency	2.8 (b)	Max	4276.44 Hz
	2.8 (a)	Initial	443.21 Hz
	2.8 (c)	Min	227.70 Hz
Amplitude	2.8 (e)	Max	3.236 V
	2.8 (d)	Initial	2.169 V
	2.8 (f)	Min	0.731 V

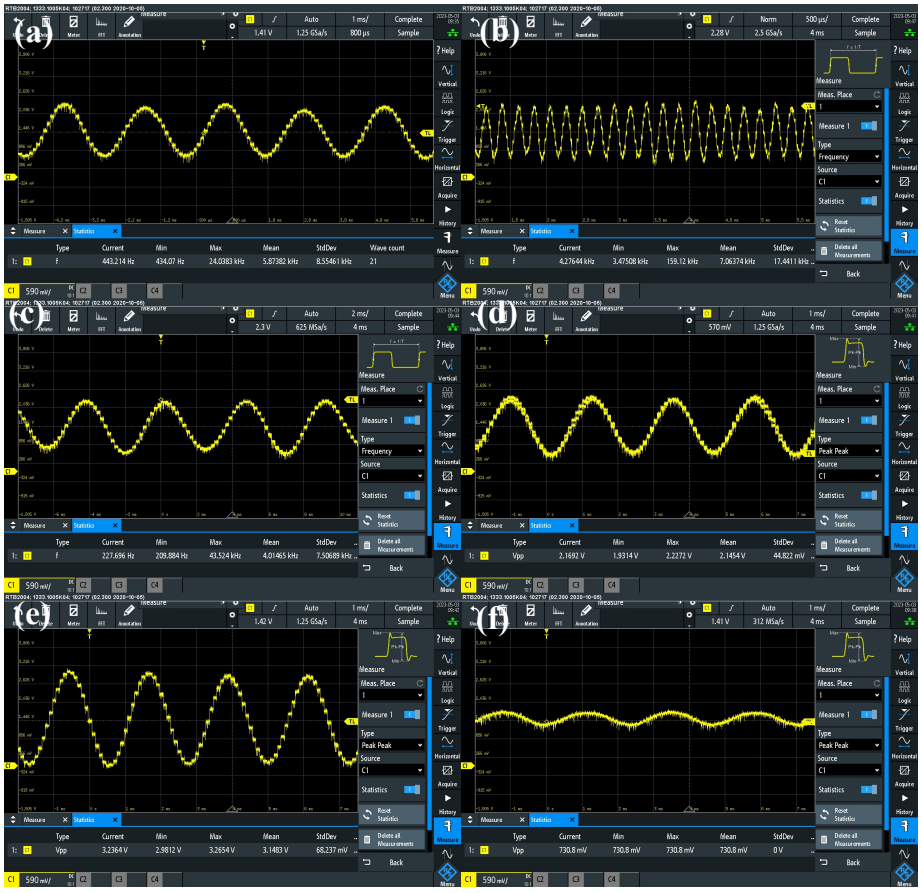

Figure 2.8. Waveforms on the oscilloscopes (a) initial frequency, (b) maximum frequency, (c) minimum frequency, (d) initial magnitude, (e) maximum magnitude and (f) minimum magnitude.

Table 3 shows the contributions of the group members.

Table 3. Contribution of teams.

Team Member	Contribution for Part 1	Contribution for Part 2
Qingyu Zhang (vn22984)	Design plan Implementation of Matlab plotting code Implementation of SystemVerilog code	Design plan Oscillator and clock divider Implementation in SystemVerilog and simulation Changing the systems' amplitude/frequency in SystemVerilog Testbench
Shuran Yang (rw22242)	Implementation of Matlab filter code Testing MATLAB code Testing SystemVerilog code	Oscillator communication and constrain file implementation on FPGA Oscilloscopes measurement Testbench Report Compilation
Ruilong Liu (hx22195)	Testing MATLAB code Testbench Report Compilation	Implementation of Matlab and Python code Testing SystemVerilog code Oscilloscopes measurement

Reference:

[1] "Advanced DSP & FPGA Implementation (EENGM4120)."

[2] O. Amp and V. Dd, "Block Diagram."

[3] "ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide," 2014.

Appendix 1

All the code is available on Github, here is the link of this project:

https://github.com/alfredzhang98/Bristol_FPGA_lab.git.

If you want to clone this project, try this command in your terminal:

git clone --recursive https://github.com/alfredzhang98/Bristol_FPGA_lab.git

In this Github project, you could access the "readme.md" file on the initial page to get the basic results from the whole coursework. Meanwhile, you could also know what the source files path is for this coursework part1 and part2 separately. It is worth mentioning that the coursework creates a ".gitignore" file to ignore the simulation files and other cache files to minimise this project. Additionally, the ".gitmodules" refer to the "spi2dac.sv" module project to avoid plagiarism. All the source codes are listed in the Github project. Therefore, in this report, only some critical codes are appended.

Appendix 2

Appendix 2.1: Matlab code for Part 1

```
clc,
clear,
close all
freq = 200; % signal frequency in Hz
SAMPLES = 512;            % Samples per full-cycle
WIDTH = 9;                % Size of data in bits
OUTMAX = 2^WIDTH - 1;     % max Amplitude of sinewave
% FIR filter parameters
% sample rate
Fs = SAMPLES * freq;
% passband cut-off frequency
Fc = freq;
% windows length
L = 7;
% windows order
M = L-1;
%omega
omiga = 2 * pi * Fc/Fs;
t = linspace(0,1/freq,SAMPLES);
rng default    %initialize random number generator
% x = 0.7*sin(2*pi*freq*t) + 0.25*rand(size(t));
x = 0.7 * sin(2 * pi * freq * t) + 0.25 * rand(size(t));
x = (OUTMAX*(1+x));
filename = 'ROMData.mem';
fid = fopen(filename,'w');
for i = 0:SAMPLES-1
    fprintf(fid,'%4X \n',int16(x(i+1)));
end
fclose(fid);
fid = fopen('./../part1_task2_4/out.txt','r');
data = textscan(fid,'%s');
fclose(fid);
num_array = [];
for i = 1:length(data{1})
    line_str = data{1}{i};
    line_str = strrep(line_str, 'x', '');
    num = str2double(line_str);
    num_array = [num_array, num];
end
input_signal = x;
a = 0.54;
```

```
n = -M/2:1:M/2;
zero_normal = find(n==0);
n(zero_normal) = 0.0000001;
W_hamming = a - (1-a) * cos(2*pi*n./(L-1)); % Hanming
W_rect = 1;
H_result = omiga ./ pi * my_sinc(omiga * n);
b = W_rect .* H_result;
b = b/sum(b);
w = 0:0.0001:pi;
H_response = zeros(1,1);
z = exp(1j*w);
for i = 1 : L
    H_response = H_response + z.^(n(1,i)) .* b(1,i);
end
figure(1)
subplot(1,2,2)
plot(w,log10(abs(H_response)),'r')
title('Frequency response');
xlabel('Normalized Frequency (x pi rad/sample)')
ylabel('Magnitude (dB)')
grid on
subplot(1,2,1)
% FIR pole
zplane(b);
xlim([-1.2 1.2])
ylim([-1.2 1.2])
title('zplane');
xlabel('Real part')
ylabel('Imaginary part')
grid on
% Filtered signal
output_signal = my_conv(input_signal, b);
figure;
plot(input_signal(L:SAMPLES), 'b-');
hold on;
plot(output_signal(L:SAMPLES), 'r-');
hold on;
plot(num_array(L:length(num_array)), 'g-');
xlabel('Time (s)');
ylabel('Amplitude');
title('Moving Average Filter in MATLAB');
legend('Input Waveform', 'Output Waveform', 'FPGA Output Waveform');
```

Appendix 2.2: SystemVerilog code for Part 1

```
module FIRFilter (clock, reset, x, y);
input     clock;
input     reset;
input     [9:0]     x;
output    [9:0]     y;
logic [9:0] z_1, z_2, z_3, z_4, z_5, z_6;
logic [9:0] mul_0, mul_1, mul_2, mul_3, mul_4, mul_5, mul_6;
//data shift
always @(posedge clock or posedge reset) begin
      if (reset) begin
            z_1 <= 10'b0;
            z_2 <= 10'b0;
            z_3 <= 10'b0;
            z_4 <= 10'b0;
            z_5 <= 10'b0;
            z_6 <= 10'b0;
      end
      else begin
            z_1 <= x;
            z_2 <= z_1;
            z_3 <= z_2;
            z_4 <= z_3;
            z_5 <= z_4;
            z_6 <= z_5;
      end
end
//coef mult
////0 00100100 coef 0.1429
assign mul_0 = (x         >> 3) + (x         >> 6);
assign mul_1 = (z_1       >> 3) + (z_1       >> 6);
assign mul_2 = (z_2       >> 3) + (z_2       >> 6);
assign mul_3 = (z_3       >> 3) + (z_3       >> 6);
assign mul_4 = (z_4       >> 3) + (z_4       >> 6);
assign mul_5 = (z_5       >> 3) + (z_5       >> 6);
assign mul_6 = (z_6       >> 3) + (z_6       >> 6);
//outcome
assign y = mul_0 + mul_1 + mul_2 + mul_3 + mul_4 + mul_5 + mul_6;
endmodule
```

Appendix 3

Appendix 3.1: SystemVerilog code for sinusoidal signal generation module for Part 2

```
module DigSineGeneratorUpdate(clk, reset, handshake_mag, magnitude, y);
input clk, reset;
input handshake_mag;
input logic [10:0] magnitude;
output logic [9:0] y;
logic signed [10:0] y_temp_1, y_temp_2;
logic signed [10:0] y_result, mul_1, mul_2;
logic shake_save1, shake_save2;
always @(posedge clk or posedge reset) begin
	if(reset) begin
		y_temp_1 <= 11'b00001000101;
		y_temp_2 <= 11'b0;
	end
	else if(shake_save1 ^ shake_save2) begin
		y_temp_1 <= magnitude[10:0];
		y_temp_2 <= 11'b0;
	end
	else begin
		y_temp_1 <= y_temp_2;
		y_temp_2 <= y_result;
	end
end
always @(posedge clk or posedge reset) begin
	if(reset) begin
		shake_save1 <= 1'b0;
		shake_save2 <= 1'b0;
	end
	else begin
		shake_save1 <= handshake_mag;
		shake_save2 <= shake_save1;
	end
end
assign mul_1 = {{{1{y_temp_2[10]}}, y_temp_2[10:1]} + {{2{y_temp_2[10]}}, y_temp_2[10:2]} +
{{3{y_temp_2[10]}}, y_temp_2[10:3]} + {{4{y_temp_2[10]}}, y_temp_2[10:4]} + {{6{y_temp_2[10]}},
y_temp_2[10:6]} + {{7{y_temp_2[10]}}, y_temp_2[10:7]}}<<1;
assign mul_2 = y_temp_1;
assign y_result = mul_1 - mul_2;
assign y = y_result + 10'd512;
endmodule
```

Appendix 3.2: SystemVerilog code for setting the value of the sinusoidal signal frequency for Part 2

```
module ButtonChangeClk(clk, reset, valid_clk, sw34_value, switch_clk);
//parameter DIV_FACTOR_10Khz = 10000; // 10KHZ
input clk, reset;
input valid_clk;
input logic [7:0] sw34_value;
output logic switch_clk;

logic [15:0] value_temp;
logic [15:0] count_clk;

//sw34_value 0-50 normal 25
always @(posedge clk or posedge reset) begin
	if (reset) begin
		value_temp <= 16'd5000;
		count_clk  <= 16'b0;
		switch_clk <= 1'b0;
	end
	else if(valid_clk) begin
		value_temp <= sw34_value * 50; // 50-20000
		count_clk  <= 16'b0;
	end
	else begin
		if (count_clk == value_temp - 1) begin
			switch_clk <= ~switch_clk;
			count_clk  <= 0;
		end
		else begin
			count_clk <= count_clk + 1;
		end
	end
end

endmodule
```

Appendix 3.3: SystemVerilog code for setting the value of the sinusoidal signal magnitude for Part 2

```
module ButtonChangeMagnitude(clk, reset, valid_mag, sw12_value, switch_magnitude, handshake_mag);
//parameter NORAML = 69;
input clk, reset;
input valid_mag;
input logic [7:0] sw12_value;
output logic [10:0] switch_magnitude;
output logic handshake_mag;

always @(posedge clk or posedge reset) begin
	if (reset) begin
		switch_magnitude <= 11'd70;
		handshake_mag <= 1'b0;
	end
	else if (valid_mag) begin
		switch_magnitude <= sw12_value; //0-126
		handshake_mag <= ~ handshake_mag;
	end
end
endmodule
```