Advanced DSP & FPGA Implementation (EENGM4120)

Coursework Assignment for FPGA Implementation

This document outlines the coursework assignment for the FPGA part of the unit. 50% of this coursework mark will contribute towards the final grade. The other 50% will be taken from the DSP part.

Note: This work can be carried out in groups of not more than three students. There are two parts in this coursework and you are required to write a report which will be assessed. The main body of the report should be no more than 10 pages long (include SystemVerilog codes in an appendix). What to include in the report for Part 1 and Part 2, and the marking details are clearly outlined at the end of each part.

Part 1: Simple FIR Filter (RTL): Simulation only [Max. 40 Marks]

In this lab, you are going to implement a Finite Impulse Response (FIR) filter using SystemVerilog. The schematic of the filter is given in Figure 1 and it can be mathematically expressed as:

$$y[n] = \sum_{k=0}^{N} b_k x[n-k]$$

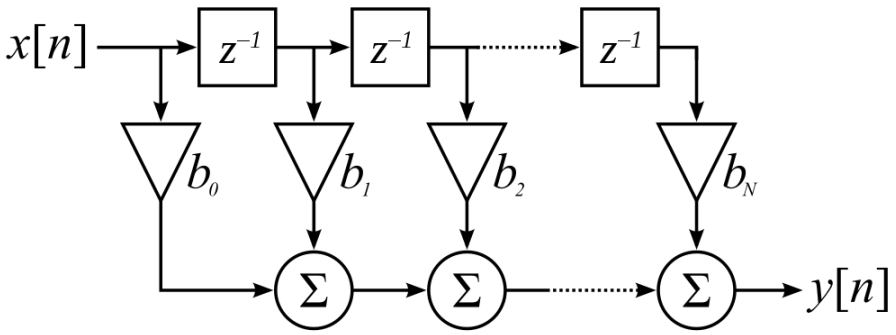

Figure 1: A discrete-time FIR Filter with order N. It has an N-stage delay line and N+1 taps.

The minimum time required for this N-tap FIR filter to generate a new output depends on the delay of one multiplier and $(N-1)$ adder delays, which is $T_{mult} + (N-1)T_{add}$, where T_{mult} is the delay of a multiplier and T_{add} the delay of an adder. Therefore the sample frequency should be decided such that $F_s \leq \frac{1}{T_{mult} + (N-1)T_{add}}$.

A 5-tap averaging FIR filter with coefficients (b_i for $i = 0, 1, \ldots, 4$) equal to $\frac{1}{5}$ is required to implement in this exercise. The input x is a 10-bit unsigned bit vector, and the output (y) could be of the same size or larger in bit-width to ensure that there is no overflow. The purpose of this class, and this lab in particular, is not to focus on filter design, but on the modelling of filter circuit modelling using HDLs. You will also gain an understanding on fixed-point number arithmetic operations.

Task 1: MATLAB simulation

Your are required to implement this filter in MATLAB. We use a noisy sinusoidal waveform (x) as the input and it is shown in Code 1. Here we generate 512 sample points of a full-cycle for a 200 Hz sinusoidal signal. Plot the input and output waveforms on the same graph to compare the filtering. In Task 3, these results will be compared against SystemVerilog simulation results.

roshan.weerasekera@bristol.ac.uk

```
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

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));

%% YOUR FILTER CODE GOES HERE
```

Code 1: MATLAB code for filter simulation

Task 2: Implementation in SystemVerilog

FIRFilter.sv is the SystemVerilog module that you will be writing in this task and a incomplete template containing the port definitions is given in Code 2. This design requires a delay line (containing registers), multipliers and adders. You may use SystemVerilog **behavioural style** to implement this FIR filter.

You are required to represent the filter coefficient fractions in fixed-point (Q0.8) format [1]. Here Q0.8 means zero bits for the integer part and 8-bits for the fraction part. Within the FIR filter, you need to carry out the multiplication operation of x and b, and their delayed components need to be added together without loosing precision. Here you need a good knowledge in fixed-point number arithmetic operations otherwise the results would be meaningless.

```
module FIRFilter(clock, reset, x, y);
input clock, reset;
input logic [...:0] x;
output logic [...:0] y;

/*
Rest of the code goes here
*/

endmodule
```

Code 2: FIRFilter.sv port definitions

One approach to implement this filter is to design a basic multiply-and-add unit and replicate as many as you want in the way demonstrated in Figure 2. This design is scalable to any stages (taps).

Typical multiplication in hardware is repeated addition, for example $M \times N$ is adding N M times. This process is computationally inefficient in terms of delay as well as area. In this design you are multiplying x[n] with a constant (same for all stages), therefore you can reduce it down to some shifting and adding [2]. For example,

$$71 \times x = 1000111_2 \times x = x \ll 6 + x \ll 2 + x \ll 1 + x,$$

reduces down to three shift operations and additions. You may use any suitable technique to improve the performance of this filter and those techniques will gain you more marks.

Task 3: Testing and Verification

In order to test the FIR filter, we are going to create a add few more blocks which generates stimulus for the FIR filter. Shown in Figure 3 is block diagram for the test setup. The FIRFilter block in the diagram is the SystemVerilog module designed in Task 2. In this task, you are required to design the address generator and the ROM blocks as explained below.

roshan.weerasekera@bristol.ac.uk

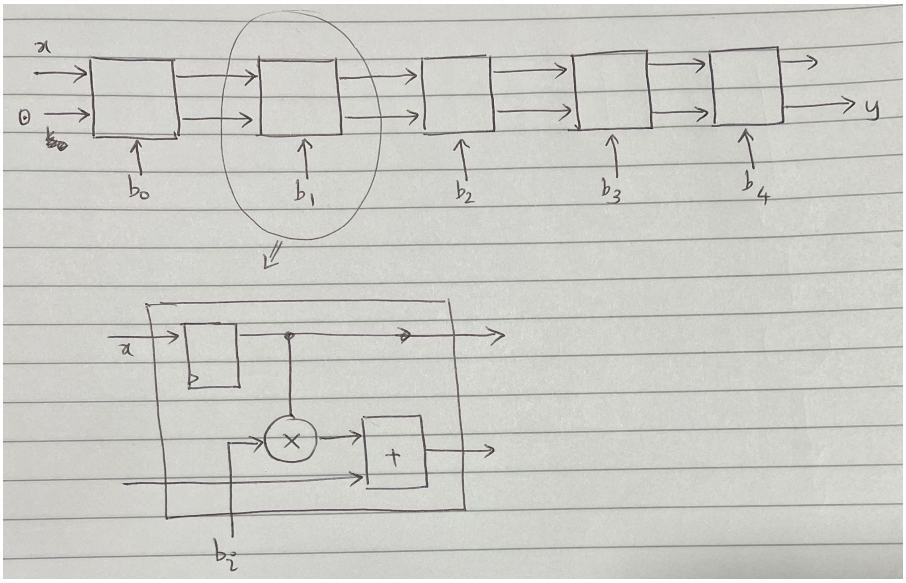

Figure 2: Scalable FIR Filter

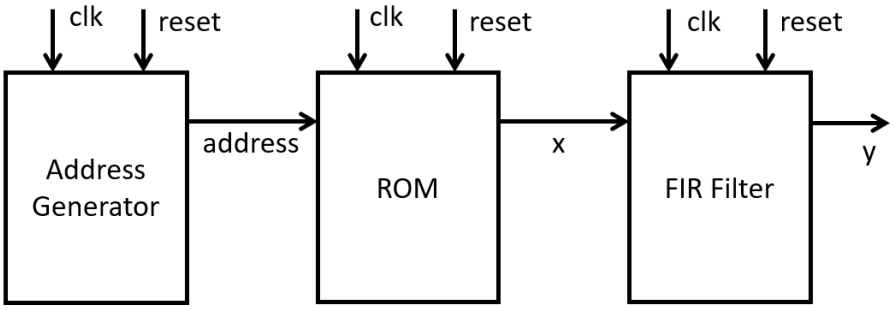

Figure 3: Block diagram for the FIR Filter Test setup

Since we are going to compare MATLAB simulation results with HDL simulations, it is better to use the same input waveform as in Task 1. To generate a sine wave, we only need data for a full-cycle and then call values repeatedly starting from the beginning. We could save the input waveform (x) data from Task 1 as a file (*ROMData.mem*). The MATLAB command lines in Code 3 will save the x variable generated in Task 1 as a file. Note that the waveform is pushed upward to make all the values positive to make it easy to process in HDL.

```
% normalize the amplitude of x so that it can be represented by 10bits
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);
```

Code 3: MATLAB code for saving waveform data

The content of *ROMData.mem* will be similar to what is shown below and it will have 512 lines representing memory locations 0 (first line) to 511 (last line). Note that data has been converted to hexadecimal values and the numbers you get might be different from what you see below because of random number addition to a pure sinewave signal.

```
267
277
218
```

roshan.weerasekera@bristol.ac.uk

```
281
261
221
23D
264
29C
2A2
23F
...
252
```

The ROM module can be designed as shwon in Code 7.

```
module dataROM(clock,address,dataOut);
input clock;
input [0:8] address;
output logic [9:0] dataOut;

logic [0:9] ROMData [0:511]; // data width is 10bits; depth is 256

initial begin
    $readmemh("ROMData.mem", ROMData);
end

always @(posedge clock)
begin
  dataOut <= ROMData[address];
end
endmodule
```

Code 4: ROM Module

The address generator module is a simple counter which count up in every clock cycle until it reaches the maximum memory location of the ROM. Then the counter should re-start. If this process continues, a multicycle sine wave can be generated. It is recommended for you to test the ROM output before it fed in to the FIR filter.

The ROM data file should be added to the project as a design source file.

Viewing the output of the FIR filter in the time domain provides a more intuitive picture of the filtering. Figure 4 shows the outputs of the filter.

Once you connect the test vector inputs to the FIR filter carry out Vivado/Modelsim simulation and view the waveforms. Here you may need a testbench file in order to simulate the design (update the details accordingly).

```

`timescale 1ns / 1ps
module Testbench();

logic clock = 0;
logic reset = 1;
logic [9:0] y;

Lab2 U0(reset, clock, y);

initial
forever #10 clock = ~clock;

initial begin
#15 reset = 1;
```

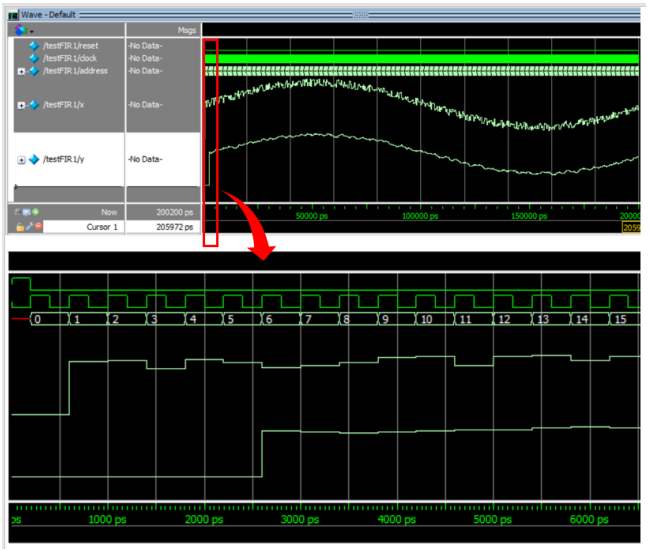

Figure 4: Modelsim Simulation waveforms - in order to display waves in Analogue form, select the waveform and right-click. Now select format and Analog (automatic).

```
#26 reset = 0;
#10000 $finish;
end
```

Code 5: ROM Module

The output waveform (y) can be saved in a .txt file which can be plotted in MATLAB with task 1 simulation data. To save the output y to a .txt file, you can use the following code segment in the SystemVerilog file.

```
initial begin
  integer fd;
  fd= $fopen("y.txt", "w");

  forever begin
    @(posedge clock);
    $fwrite(fd,"%d \n", y);
  end
end
```

Code 6: ROM Module

Task 4: Higher Order Filter Design and Synthesis

MATLAB provides a tool to design a basic quantized discrete-time FIR filter, generate HDL code for the filter, and verify the HDL code with a generated test bench. There is a guided tutorial which can be accessed from: `https://uk.mathworks.com/help/hdlfilter/basic-fir-filter.html` Follow the tutorial and understand the process. Input the same sinusoidal input in this task and see the filter output.

roshan.weerasekera@bristol.ac.uk

What to submit/include in the report

(a) An explanation about the design approach (include the techniques used in the design with required diagrams etc) [05 marks]

(b) SystemVerilog codes for the entire design [10 marks]

(c) Test plan for individual blocks and a summary of results [05 marks]

(d) Plots/figures/screenshots of functional simulation in MATLAB and/or Vivado/Modelsim (demonstrate your results to a TA or the Lecturer to obtain marks for this section, otherwise this part will be marked out of 7). [15 marks]

(e) A discussion explaining your results [05 marks]

roshan.weerasekera@bristol.ac.uk

Part 2: Digital Sinusoidal Oscillator Design [Max. 60 Marks]

Background

There is a widespread requirement in sinusoidal signal generation in engineering such as communications, control, music synthesis, and instrumentation because it is the basis for reproducing any periodic signal using Fourier components. Traditionally a sine wave oscillator can be built using analogue circuits, but the control of the signal parameters is difficult in comparison to digital implementation.

In the previous lab, we have used a Look-up table (LUT) method for the same purpose but it takes up the memory space in the hardware to store sample amplitudes and when the sampling frequency is higher, the higher the memory space. And therefore it is not a viable option for resource constraint applications. Alternatively CORDIC (Coordinate Rotational Computer) algorithm, and the polynomial approximation of the sine or cosine function are some of the other methods. The approach we are going to use in this lab is based on second-order Infinite Impulse Response (IIR) filter design [3–5].

As explained in [6] (Section 4.5.7), the second-order system given by:

$$H(z) = \frac{b_0}{1 + a_1 z^{-1} + a_2 z^{-1}}$$

with $a_1 = 2r\cos\omega_0$ and $a_2 = r^2$ generates a response of:

$$h(n) = \frac{b_0 r^n}{\sin\omega_0} \sin(n+1)\omega_0 u(n)$$

If the poles are placed on the unit circle ($r = 1$) and b_0 is set to $A\sin\omega_0$, the impulse response of this second order system is:

$$h(n) = A\sin(n+1)\omega_0 u(n)$$

a digital sinusoidal oscillator is a limiting form of a two-pole resonator with the complex-poles lie on the unit circle ($r = 1$).

It can be expressed in mathematically as shown below:

$$y[n] = -a_1 y[n-1] - y[n-2] + b_0\delta[n]$$

where the parameters are $a_1 = -2\cos\omega_0$, $b_0 = A\sin\omega_0$ and ω_0 the output signal frequency, and the initial conditions are $y[-1] = y[-2] = 0$. The application of the impulse at $n = 0$ instigates the sinusoidal oscillation and thereafter, the oscillation is self-sustaining as the system has no damping (i.e. r=1).

Interestingly, the difference equation can be simplified as below:

$$y[n] = -a_1 y[n-1] - y[n-2]$$

by setting the input to zero and setting initial conditions to $y[-1] = 0$, $y[-2] = A\sin\omega_0$.

This generator is rather simple and does not consume much memory space and takes lot of computational time as every sample is calculated only when it is requires, but it may cause instability because positive feedback is employed [7].

In discrete form, we can write ω_0 as $\frac{2\pi F_0}{F_s}$ where F_0 is the signal frequency and F_s the sampling frequency such that $F_s > 2F_0$. In summary, we can write the difference equation for the digital oscillator as:

$$y[n] = 2\cos\left(\frac{2\pi F_0}{F_s}\right) y[n-1] - y[n-2] \tag{1}$$

with $y[-1] = 0$, $y[-2] = A\sin\left(\frac{2\pi F_0}{F_s}\right)$.

A python implementation of the algorithm produces the sinusoidal waveform in Figure 5.

roshan.weerasekera@bristol.ac.uk

```python
import math
import numpy as np
import matplotlib.pyplot as plt

F0 = 440    # signal frequency (middle c)
Fs = 10e3 # sampling frequency
A = 1 # signal amplitude
r = 1 # damping factor
SAMPLES = int(Fs/F0)

# coefficients
a1 = r*math.cos(2*math.pi*F0/Fs)
a2 = r*r

# initialize ysin
ysin = np.array([A*math.sin(2*math.pi*F0/Fs), 0])

for i in range(2,2*SAMPLES):
    ynew = 2*a1*ysin[i-1] - a2*ysin[i-2]
    ysin = np.append(ysin, ynew)

plt.step(ysin,'r-',where='post', linewidth=0.5)
plt.plot(ysin,'o--',linewidth=0.5, color='grey', alpha=0.3)
plt.xlabel("Sample")
plt.ylabel("Amplitude")
plt.show()
```

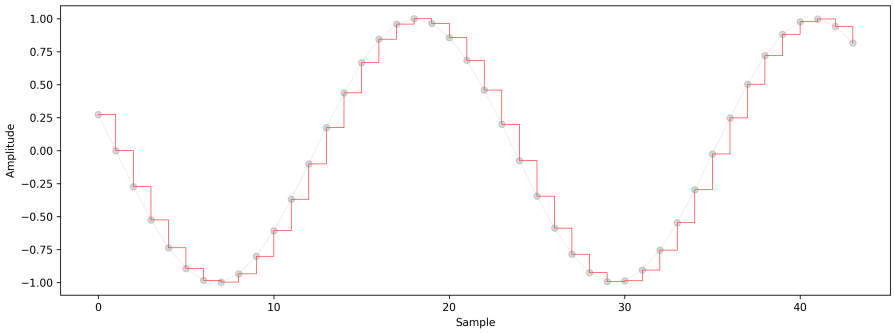

Figure 5: Digital Filter Sinusoidal Waveform output

roshan.weerasekera@bristol.ac.uk

Task 1: Oscillator Implementation in SystemVerilog and simulation

You are required to implement the difference equation explained in the background section in SystemVerilog . The `DigSineGenerator.sv` module will have two 1-bit inputs (clock and reset), and one 10-bit output y. The output y should be a sinusoidal signal with frequency (F_0) equal to 440 Hz. The sampling frequency is not fixed, but should be as high as possible to generate a smooth sine wave output.

```
module DigSineGenerator(clk, reset, y);
input clk, reset;
output logic[9:0] y;

/*

YOUR CODE GOES HERE!

*/
endmodule
```

Code 7: template for DigSineGenerator.sv

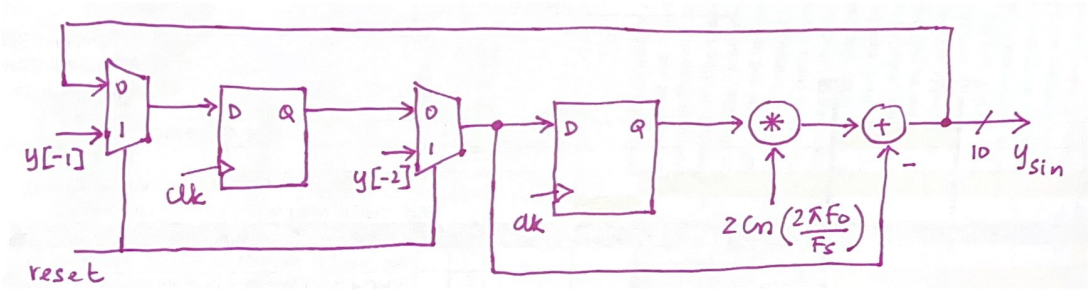

Figure 6: Digital Oscillator Schematic

Using the equation (1), a block diagram of the architecture for the oscillator is shown in Figure 8. At the start *reset* should be logic one for at least 1 clock cycle to store the initial values in the registers and calculate y[0]. After that *reset* should be set to logc zero so that updated values to flow through the multiplexors and store in the registers to aid in estimating y[1] and keep that continue in every clock cycle.

How would you plan this design? The main challenge is to represent the initial condition y[-2] and the multiplication constant because both values are less than 1. The accuracy of the final sine value will depend on this initial value.

Carry out the design and demonstrate the functionality using Modelsim/Vivado simulations (see the example in Figure 7).

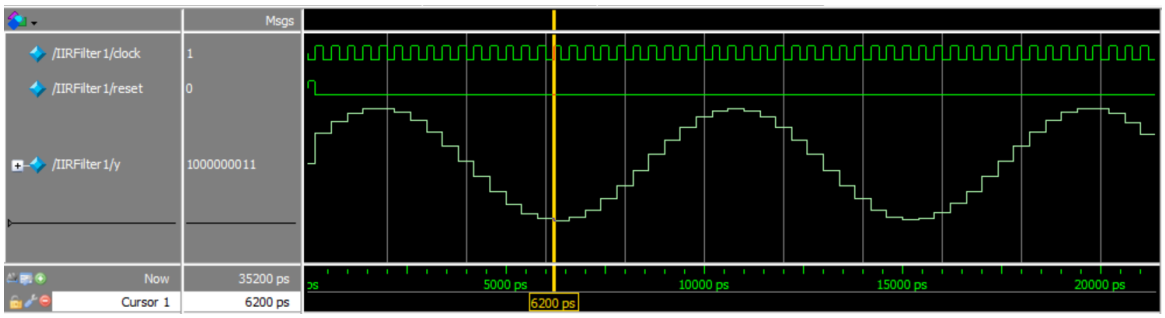

Figure 7: Digital Filter Sinusoidal Waveform output

roshan.weerasekera@bristol.ac.uk

Task 2: Oscillator Implementation on FPGA

Once the model has been verified, the design should be implemented on a FPGA. In order to test the digital output we use a Digital-to-Analogue Converter (DAC) and its output can be monitored on a oscilloscope.

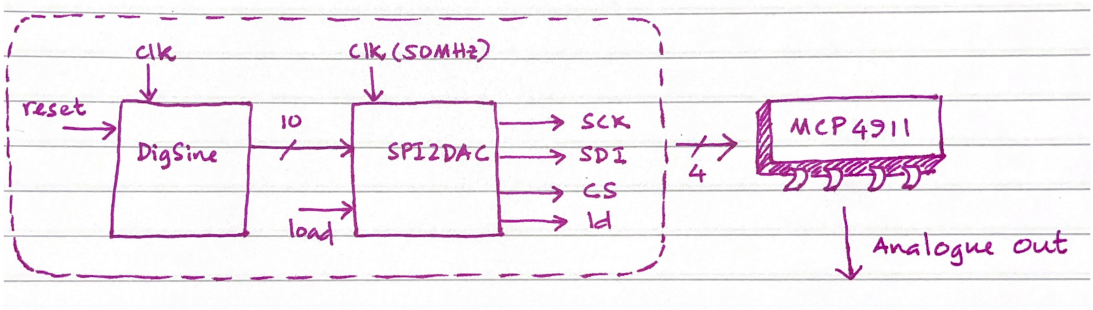

Figure 8: Digital Oscillator Schematic

The DAC which is available for this purposes is MCP4911 [8] has a single channel and uses 10-bit SPI interface to load data into it. In order to communicate with the DAC you need a controller to send data to the DAC using its SPI interface. Read the datasheet for MCP4911 to understand the operating mechanism of the DAC and the timing details of the SPI interface. The `spi2dac.v` module available in [9] which is a controller for the DAC. Note that the input clock speed for the spi2dac module is 50MHz and you may need some modifications to match the sine wave generator.

Task 3: Measurement and Analysis

You are expected to measure the frequency of the output signal and compare it against your design value (440 Hz). The amplitude can be in the range of 1V - 5V. Carry out this comparison for frequencies from 100 Hz to 1000 Hz.

You could add some novelty into this design for example variable amplitude and frequency.

What to submit/include in the report

(a) An explanation about the design approach (include the techniques used in the design with required diagrams etc) [05 marks]

(b) SystemVerilog codes for the entire design [10 marks]

(c) Plots/figures/screenshots of functional simulation in MATLAB and/or Vivado/Modelsim (demonstrate your results to a TA or the Lecturer to obtain marks for this section, otherwise this part will be marked out of 7) [10 marks]

(d) Snapshots of oscilloscope outputs of the DAC demonstrating the output signal parameters (demonstrate your results to a TA or the Lecturer to obtain marks for this section, otherwise you will receive max 10 marks for this part). [25 marks]

(e) Extra design features introduced and results (can you change the amplitude/frequency while in operation?). [10 marks]

Note: Include a seperate sheet showing the specific contribution from your team members to Part 1 and Part 2.

roshan.weerasekera@bristol.ac.uk

References

[1] D. Goldberg. “What every computer scientist should know about floating-point arithmetic”. In: *ACM Computing Surveys* 23.1 (Mar. 1991), pp. 5–48. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163 (visited on 12/24/2022).

[2] Y. Voronenko et al. “Multiplierless multiple constant multiplication”. In: *ACM Transactions on Algorithms* 3.2 (May 2007), 11–es. DOI: 10.1145/1240233.1240234. URL: https://doi.org/10.1145/1240233.1240234 (visited on 12/25/2022).

[3] T. Hodes et al. “A fixed-point recursive digital oscillator for additive synthesis of audio”. In: *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No.99CH36258)*. Vol. 2. ISSN: 1520-6149. Mar. 1999, 993–996 vol.2. DOI: 10.1109/ICASSP.1999.759867.

[4] A. Abu-El-Haija et al. “Recursive digital sine wave oscillators using the TMS32010 DSP”. In: *Conference Proceedings. 10th Anniversary. IMTC/94. Advanced Technologies in I & M. 1994 IEEE Instrumentation and Measurement Technolgy Conference (Cat. No.94CH3424-9)*. May 1994, 792–796 vol.2. DOI: 10.1109/IMTC.1994.351886.

[5] L. Yan et al. “An Implementation Method for Low Frequency Digital Oscillator”. In: *International Journal of Simulation: Systems, Science & Technology* (Jan. 2016). DOI: 10.5013/IJSSST.a.17.25.10. URL: https://ijssst.info/Vol-17/No-25/paper10.pdf (visited on 12/25/2022).

[6] J. G. Proakis et al. *Digital signal processing: principles, algorithms, and applications*. eng. 3. ed. Prentice Hall international editions. London: Prentice-Hall International (UK), 1996.

[7] URL: https://www.ti.com/lit/an/spra819/spra819.pdf?ts=1671966681180&ref_url=https%253A%252F%252Fwww.google.com%252F (visited on 12/25/2022).

[8] *MCP4911 | Microchip Technology*. URL: https://www.microchip.com/en-us/product/MCP4911 (visited on 12/24/2022).

[9] G. Padley. *VERILOG-Lab*. original-date: 2016-12-08T19:14:44Z. Dec. 2020. URL: https://github.com/GPadley/VERILOG-Lab/blob/513932f76546eade221fad1c63749b2880c9d5e6/My%20Lib/spi2dac.v (visited on 12/20/2022).

roshan.weerasekera@bristol.ac.uk