Low Cost Direct Digital Synthesis Using an FPGA

Presented by:
Myrin Naidoo

Prepared for:
Dr F. Schonken
Dept. of Electrical and Electronics Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town
in partial fulfilment of the academic requirements for a Bachelor of Science degree in
Electrical and Computer Engineering
Myrin Naidoo (Electrical and Computer Engineering)

Code Available at: https://github.com/MyrinNaidoo12/FPGA-DDS-Thesis
The code can be downloaded as a zip file from Google Drive
The code can also be found in Appendix A.

July 5, 2020

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

Signature:...........................
M.Naidoo

Date:...............................

Acknowledgements

To my parents, Shan and Thiloshini. Thank you for encouraging me and listening to me worry. Thank you for allowing me to become a hermit in my room, during the corona virus lock down, to complete this. Your never ending support is the reason I made it this far.

To my sister, Yekisha. Thank you for always making me strive to be better. You've never let me be complacent with my work and have kept my ego in check. Your unbiased opinions have kept me on the right path throughout my university career.

To my aunt, Margaret. Thank you for letting a 10 year old me use your very expensive laptop. That ancient laptop is what got me interested in technology in the first place. Without it, and you, I may never have studied engineering.

Thank you to Walter, Thomas, Thanusha, Kira and Yusuf for keeping me sane while completing this. Thank you to Sylvan, Alex, Walter and Luka for helping me study. Thank you to all my friends for keeping my life balanced. I couldn't have asked for better friends.

Abstract

Direct digital synthesis(DDS) in this context is the creation of a sine wave at tunable frequencies. This is commonly done using an integrated circuit. These integrated circuits are generally expensive. This project aims to create an alternative to this by using FPGAs that are already owned by the university. An FPGA is a Field Programmable Gate Array. It has a configurable processor made up of a large number of logic gates. The interconnections of these gates can be changed using hardware description languages. Two such languages are Verilog and VHDL. For the purposes of this study, verilog will be used primarily. A DDS integrated circuit accepts a frequency as an input and produces a digitally sampled sine and cosine wave as an output. The circuit would typically have a range of frequencies it can operate within. The device may also produce a linear frequency sweep, which is a sine wave that increases or decreases in frequency after each wavelength. Another feature that these devices can be capable of is quadrature modulation. Quadrature modulation takes two signal and modulates them with either a sine wave or cosine wave. The modulated waves are then added together and sent to a receiver.

Contents

1 Introduction **1**

1.1 Background to the study 1

1.2 Problem Statement 1

1.2.1 Purpose of the study 2

1.3 Plan of development 2

1.3.1 Project Requirements 3

1.4 Scope and Limitations 3

1.5 Problems Encountered 4

2 Literature Review **6**

2.1 Direct Digital Synthesis(DDS) 6

2.2 Hardware 8

2.3 Linear Frequency Sweep 8

2.4 Quadrature Modulation 8

3 Design and Implementation **10**

3.1 Hardware 10

3.1.1 FPGA 10

3.1.2 Digital to Analog Converter 10

3.2 Software and Hardware Description Languages 11

3.3 Block Diagram 12

3.4 Components 13

3.4.1 Sine Generator 13

3.4.2 Frequency Tuner 14

3.4.3 Multiplier 15

3.4.4 Adder 15

3.4.5 Signal Generator 15

3.4.6 Electrical DAC Circuitry 16

3.4.7 Components Not Used in Final Version 17

3.5 Assembly and Test Benches 18

3.5.1 Analog Version 19

3.5.2 Main Final Version 20

3.5.3 High Frequency Version 21

3.5.4 Ideal Version 21

4 Version Progression **22**

4.1 Initial Orientation 22

4.2 Version One 22
4.3 Version Two 23
4.4 Version Three 23
4.5 Version Four 24
4.6 Version Five 24
4.7 Version Six 24
4.8 Final Versions 25
4.8.1 Main Version 25
4.8.2 Alternate Version - High Frequency 26
4.8.3 Alternate Version - Analog Version 26

5 Results and Discussion 27

5.1 Unit Testing 27
5.1.1 Sine Wave Generator 27
5.1.2 PWM Generator 28
5.1.3 Digital to Analog Converter 28
5.1.4 Multiplier 29
5.1.5 Adder 29
5.2 Simulation Results 30
5.2.1 Sine Wave 30
5.2.2 Frequency Sweep 31

5.2.3 Quadrature Modulation 32

5.3 Simulation Results - High Frequency Model 36

5.3.1 Sine Wave 36

5.3.2 Linear Frequency Sweep 36

5.3.3 Quadrature Modulation 36

5.4 Simulation Results - Ideal Model 38

5.5 Experimental Results 39

5.5.1 Period and Frequency 39

5.5.2 Accuracy of Quadrature Modulation 40

6 Conclusions **43**

7 Recommendations **45**

7.1 Implementation 45

7.1.1 Hardware 45

7.1.2 Analog Signals 46

7.1.3 Frequency Tuning and Sample Sets 46

7.2 Testing 46

7.2.1 Simulation Software 46

7.2.2 Lab Equipment 46

7.2.3 Demodulation 47

A Code Files **50**

A.1 Sine Module 50

A.2 Multiply Module 51

A.3 Test Bench 51

A.4 Signal Generator 52

A.5 Adder 53

A.6 Frequency Translater 53

B Addenda **54**

B.1 Ethics Forms 54

Chapter 1

Introduction

1.1 Background to the study

Direct digital synthesis(DDS) in this context is the creation of a sine wave at tunable frequencies. This is commonly done using an integrated circuit. These integrated circuits are generally expensive. This project aims to create an alternative to this by using FPGAs that are already owned by the university. An FPGA is a Field Programmable Gate Array. It has a configurable processor made up of a large number of logic gates. The interconnections of these gates can be changed using hardware description languages. Two such languages are Verilog and VHDL. For the purposes of this study, verilog was used primarily. A DDS integrated circuit accepts a frequency as an input and produces a digitally sampled sine and cosine wave as an output. The circuit would typically have a range of frequencies it can operate within. The device may also produce a linear frequency sweep, which is a sine wave that increases or decreases in frequency after each wavelength. Another feature that these devices can be capable of is quadrature modulation. Quadrature modulation takes two signals and modulates them with either a sine wave or cosine wave. The modulated waves are then added together and sent to a receiver.

1.2 Problem Statement

The main objective was to configure an FPGA to output a sine and cos wave at tunable frequencies as a replacement for Direct Digital Synthesis done by an integrated circuit.

It should be able to generate a linear frequency sweep and convert an input bit stream into a quadrature modulated output. The product must be cost effective for the use in low budget studies.

1.2.1 Purpose of the study

This study aims to give future studies an alternative to expensive DDS integrated circuits. This will allow students and researchers to develop projects using cheap and reusable equipment. The device should also produce a linear frequency sweep and be capable of applying quadrature modulation to an input wave. These are commonly used functions of DDS integrated circuits. Previously expensive projects may become feasible to those with limited budgets by using these tools.

1.3 Plan of development

The first step is to set up the IDE and run basic tutorials. This will establish familiarity with the IDE and remove the possibility of extraneous errors from the device or from installation. The project will start with a basic version that has a single output and iterates through a lookup table in memory. More features will be added once the previous function works. Next, the linear sweep will be implemented. Lastly the input bit stream will be converted into quadrature modulated output. The features will be developed as separate modules. Each module will be tested separately to ensure it is functional. The device will then be assembled and tested further to ensure the components work in conjunction and produce the correct output.

The project methodology will follow the literature review and develop a prototype using methods that have previously been implemented. The success or failure of this will dictate how the approach needs to be altered. Simulations shall be used to test the code before implemented on hardware. The testing shall be done by graphically confirming that the outputs are correct and then through manual computation and comparison of data.

After this, the design and implementation elements will be explained and thorough explanations of all of the components will be detailed. The chronological progression of the project and named versions of the code will be documented.

The results section will consist of unit testing of each component, simulation results from the device and experimental results in which the outputs of the devices are checked for correctness. In a following discussions chapter the overall results are discussed in detail, along with an evaluation of what the project accomplished. The conclusion will summarise the results of the study and the recommendations will highlight improvements that could be made if the project was extended.

1.3.1 Project Requirements

The following requirements for the project are based on the project brief, related literature and the available equipment. The requirements will also help to narrow the scope of the project. The project must:

1. Use low cost hardware
    - It must be done using an FPGA
    - The FPGA used must be cheap and/or currently available.
    - Any other hardware used must be cost effective and attainable
2. Store a digital waveform
    - Read only sine wave stored in memory to be accessed when generating a wave.
3. Output a sine and cosine wave
4. Have an adjustable frequency
5. Manipulate the frequency of a digital sine and cosine wave
6. Produce a linear frequency sweep
7. Accept an input bit stream and apply quadrature modulation to it
8. Provide testing to prove the outputs are correct

1.4 Scope and Limitations

The scope of this study will be to generate a sine signal and cosine signal. The study will determine the range of frequencies that can be generated. The project must allow for a linear sweep of frequencies and quadrature modulation of a input bit stream.

The project will not establish a price point or a cost difference between the chosen FPGA and the integrated circuit. FPGAs that are currently owned by the university will be used. The aim is to use components that are already in stock as a cheaper alternative to buying IC components for single use.

Due to COVID-19 pandemic, the practical testing was limited. The components were tested using simulations generated by vivado, and graphs created using values from log files. Without lab equipment analog outputs can not be tested so the project will use digital waves and values. The bit stream input can not be tested using an external bit stream but accommodations will be made so it can be used at a later stage. The modulation will instead be tested by using bit streams generated in the simulation.

1.5 Problems Encountered

The following are the issues that arose with the project. These include lab constraints and time constraints and COVID-19.

Before the goal was attempted, the first thing done was to execute basic interactions with the FPGA. This was to ensure the version of the IDE was correct and to ensure there were no errors that would effect later programs. The program was a basic set of instructions to switch on the LEDs and switches so that the LEDs are controlled by the switches.This set up a base line for the subsequent code. This was following a tutorial for the Nexys 4 DDR. It was not discovered till much later but this tutorial did not make use of any output pins. The Nexys 4 DDR does not possess any output pins, or at least not any that an IDE would recognise as output pins (the device does contain output ports with fixed configurations that function with a XADC chip on segmented data between the pins in the port). This became important during Implementation when the sine wave was assigned to a pin in verilog that did not exist. Many weeks of failed experimentation followed, during which the IDE attempted to automatically place the pins and then failed. This showed as an error in the placement by the program and seemed to imply the constraints on the program were incorrect. The next attempt was to manually assign the ports. This also failed as the ports were not compatible with the output pins that they were mapped with. At some point during this laborious, time consuming and futile process the physical FPGA board was examined and the absent pins became apparent.

During this progression of time the world had changed in a significant way. Due to a culinary incident, in the Wuhan district of China which resulted in the likely xeno-transmission of a coronavirus strain, the world had gone into global quarantine. This meant that the remaining research and development would be confined to a bedroom in Durban. The resources were now limited to a FPGA and a laptop. The practical elements of this study would have needed lab equipment to test. As a result the project was confined to simulations.

Chapter 2

Literature Review

2.1 Direct Digital Synthesis(DDS)

Direct digital synthesis has been accomplished using other FPGAs in multiple studies. The resultant devices have had numerous applications. One such project uses a Heron FPGA. The project creates a wave that has positive, up-scaled values which are represented as integers. The idea behind this is to reduce the processing power needed to represent the numbers. Real numbers require the use of more gates than integers. The Heron study makes mention of a digital oscillator which generates a sine wave that can be adjusted in amplitude, phase and frequency. A single channel DDS is made up of just a single digital oscillator. A multi-carrier DDS has one or more digital oscillators and a sum function to combine them into a single output. This project will focus on a single channel model, at least initially. [1]

The methods used in different studies are fairly similar. They involve a look up table or array of some kind which contains sine wave values. The table is iterated at different rates for different frequencies. The values can be manipulated to increase or decrease the amplitude. The eventual wave needs to be shifted down the y axes to include negative values.

A more complex version can be seen in the study by P. Chandraman. It does still use a look up table but involves more complex methods of tracking phase. This device can synthesize sine, cosine, triangular, square and arbitrary waves. This version uses an accumulator to track the phase of the wave by allocating a number between 0 and 256 to represent the number of radians. 255 would be equal to 2pi radians and 0 is equal to

0 radians.[2]

Some studies were tangentially related as they used FPGA DDS as a basis for larger projects. FPGA DDS was used in a radar study as a signal generator as they found it had low jitter in terms of frequency and phase.[3] It has also been used as a stepping stone to create an analog signal. By passing the digitally generated signal through a filter it becomes an acceptable analog sine wave. [4]

All implementations seem to use the simple method of iterating a table of values that are predetermined and hard coded into the program. There is one main factor that can be varied within this context: number of samples. The number of samples dictates the frequencies that you can output later. The maximum frequency, when using the rising edge, would be iterating through the samples at the same rate as the clock (ie. one sample value per clock pulse). If values were changed on the rising and falling edge then a higher frequency can be obtained. Outside of increasing the number of changes per clock pulse, there is no way to increase speed without sacrificing resolution. For the purpose of this study it would be prudent to use the full list of values every time instead of reducing the number of samples for higher frequencies. By reducing the number of samples it may alter the shape of the desired wave and no longer represent a sinusoidal wave. Most studies seem to use frequency in radians as an input. This may end up being best but this project will attempt a version using Hertz as the input.

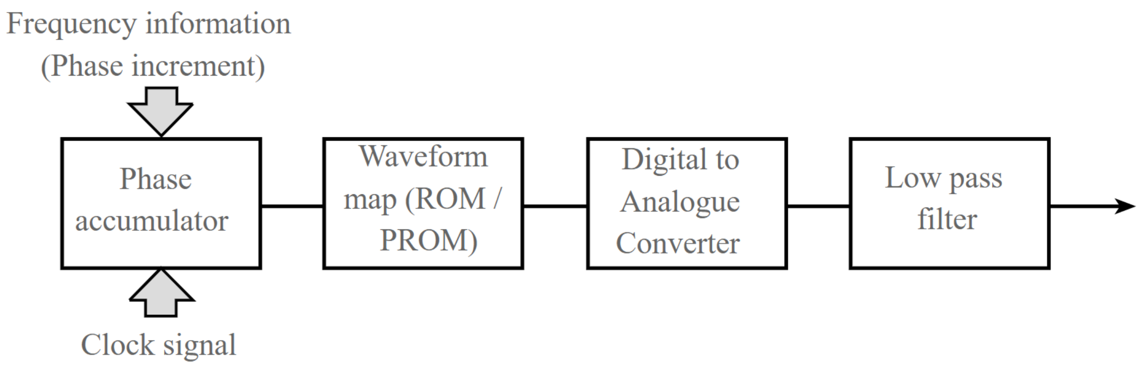


Figure 2.1: DDS pipeline. graphic from www.electronics-notes.com

2.2 Hardware

The device chosen is the Nexys 4 DDR. It is built to be affordable for students to use. It doesn't have as powerful a processor as the FPGAs used in other studies. The outputs from this device aren't dedicated to pins as one would expect. There are output ports which can be used but they do not register as output wires in verilog. The screen is also a potential means of representing the output. The benefit of this device is that it is readily available at UCT and it is a relatively cheap FPGA. It has a 100Mhz clock frequency. The XADC chip acts as a analog to digital converter.

2.3 Linear Frequency Sweep

A frequency sweep is a function that produces a wave that increases or decreases in frequency (the change can be linear or exponential). The frequency sweep contains one wavelength of each frequency. This study will focus on linear frequency sweeps (also referred to as a linear chirp). It has sonar and radar applications. It is one of the aims of this project to produce a linear frequency sweep. [5] The end goals of the Malaysian and German studies referenced preciously were both to create a chirp generator. These were used for radar applications.[4] [3]

2.4 Quadrature Modulation

Quadrature modulation is a technique that uses a high frequency sine and cosine wave to transmit two signals to a recipient. Of the two signals, one is multiplied by the sine wave, the other is multiplied by the cosine wave. The sine and cosine waves must be at the same frequency. The waves are then superimposed over one another and transmitted. [8] To demodulate, and separate the combined wave, the wave is multiplied by a sine wave and a copy of the wave is multiplied by a cosine wave. These must be at the same frequency as the waves they were modulated with. The resultant waves are then separately passed through low pass filters. The low pass filter removes the high frequency sine or cosine waves which have separated from the intended signal (in the frequency domain) due to a trigonometric identity. [9]

The literature primarily consists of patents for methods and circuitry to modulate and

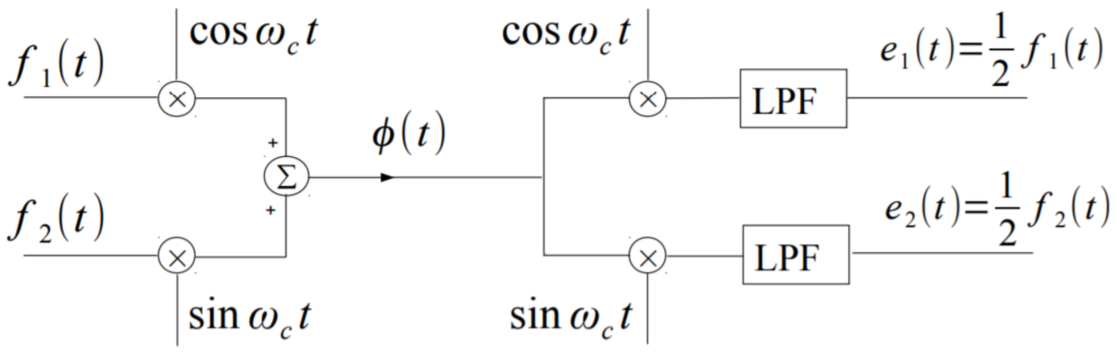


Figure 2.2: Quadrature Multiplexing Block Diagram
Taken from lecture slide created by A.J.Wilkinson, UCT

demodulate signals. The devices seem to rely on the same basic principles, to modulate and demodulate, that will be discussed in this project. [11] [10]

Chapter 3

Design and Implementation

Code Available at: https://github.com/MyrinNaidoo12/FPGA-DDS-Thesis
The code can be downloaded as a zip file from Google Drive
The code can also be found in Appendix A.

3.1 Hardware

3.1.1 FPGA

This study was designed with the Nexys 4 DDR in mind as it is available at UCT and is currently used for coursework. This FPGA is designed and priced for students so it is a cost effective unit. Further testing would be necessary but it is likely that this project could be easily adapted for similar FPGAs. The practical component of this project has been performed through simulations (see Problems Encountered).

3.1.2 Digital to Analog Converter

The FPGA used does not have a DAC so it will be simulated using verilog. Some FPGAs have a built in DAC which would have been preferable. While this did make the code more complex it allows the project to be used on more FPGAs. This device does have a XADC chip that is classified as a mixed signal chip. It is documented as being able to convert analog to digital signals but there is no evidence that DAC operations are possible with this device. The conversion to an analog signal was not used in the main version of the final code.

3.2 Software and Hardware Description Languages

Verilog

There are two hardware description languages that were considered for this study; Verilog and VHDL. Verilog is an easier language to use as it is more simplistic and follows conventions of most programming languages. VHDL is a more complex language that is an entity unto itself and follows a unique format. Verilog was chosen due to its simplicity. The syntax is similar to C coding. These languages are not considered programming languages as they operate on a more basic level. They are called hardware description languages. Hardware description languages describe the behavior of circuitry. In the case of the FPGA the language dictates how logic gates are connected in order to configure a custom processor for the intended purpose. [6]

Vivado

The software used to write the Verilog code is called Vivado. It has the appearance of a normal IDE. The program can be used to simulate an FPGA, track the outputs and use an artificial system clock. The program allows different FPGAs to be simulated and respond accordingly for their various hardware specifications and constraints. The software also allows code to be loaded to the FPGA without the need for command line operations.

ModelSim

ModelSim is simulation software that is compatible with Vivado. This software was acquired as it was likely that Vivado simulations would not be sufficient. ModelSim is supposed to integrate with Vivado to formulate the simulations with minimal manual intervention. Vivado needs the libraries to be compiled before a simulation is executed. The libraries did not correctly interface with Vivado and as a result the software would not simulate the system. Due to this, the simulations were done using vivado and any graphs that weren't generated there were created manually using Mircosoft Excel.

3.3 Block Diagram

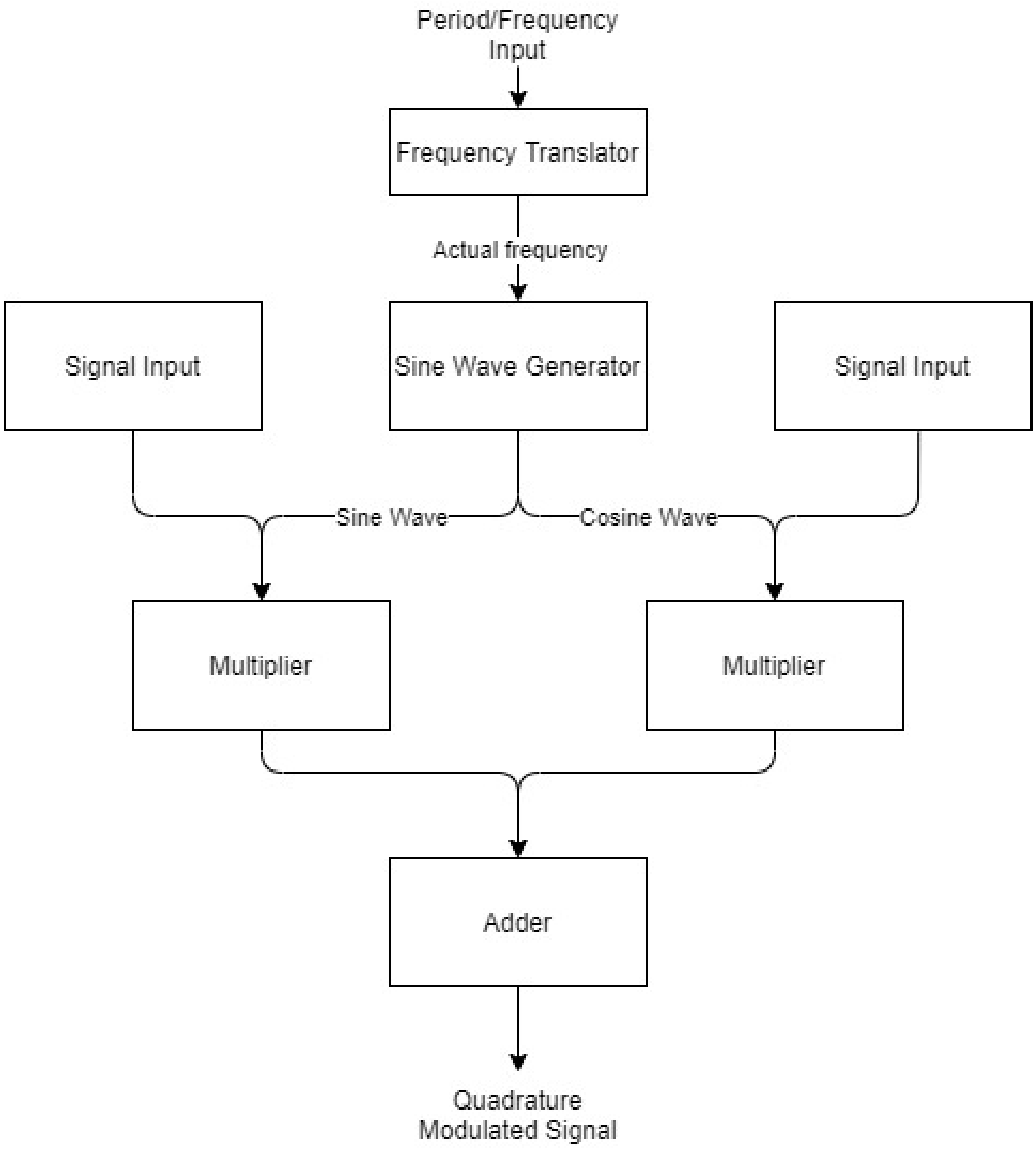


Figure 3.1: Block Diagram of System

In Figure 3.1 the block diagram of the system shows the flow of the program without the user input values. This is how a quadrature modulated signal is produced. The output waves are labeled to allow sine waves or cosine waves to be used separately in the system.

3.4 Components

3.4.1 Sine Generator

module sinemodule(Clk, freq, mode, data_out, co_out);

This produces a digital sine wave. It can be varied in frequency but not amplitude. The values range from -77 to 77. The module contains values for a sine wave that are stored in an array. The are 38 values stored but 29 values make up a wavelength. The table is iterated at intervals of the clock pulse. The generator has three inputs and two outputs. The first input is the Clk, this accepts a clock input from the FPGA or test bench. The second input is freq which is a multiplier of clock impulses per value change. By increasing the freq input, the frequency of the output wave is decreased. The value changes every 2ns when the frequency is set to 1. The frequency input times 2ns is the time step and the frequency input times 2ns times 29 outputs the period, the formula can be found below.

The next input is the mode. If the device is in mode 0 it produces a sine wave to the frequency value. The program uses two integers as counters; i and j. Integer i is a counter used to track which sample in the array is being output. Integer j is a counter used to track how many rising edges of the clock pass before the i value is changed. As j is increased it reaches the value from the freq input. When this value is reached i is incremented and j is set back to 0. The value output in the co_out output uses the value in the array of i+7. If the device is in mode 1 then it performs a frequency sweep. For the frequency sweep, after each full period the frequency input is increased by one so after each wavelength the time step is increased by 2ns (increasing the period by 58ns). The frequency sweep will continue as long as the device is run. For the frequency sweep the freq input is ignored. Instead it uses a moving upper limit for the j value. An integer FreqCount is used as the limit for the i value to change. It is initially set to 1 and then increased by 1 every period. When i reaches 29 it is reset to 0. For simulation purposes 600 samples were used. The shortest period was 58ns as one period is 29 samples at 2ns per sample. The periods that can be generated are multiples of 58ns.

$$ActualFrequency = \frac{1}{(FrequencyInput)(2x10^{-9})(29)}$$

This is only accurate when using the 500MHz processor that the simulator uses. The Nexys 4 DDR had a 100Mhz clock, which means that the actual values of the frequency follows this formula:

$$ActualFrequency = \frac{1}{(FrequencyInput)(10^{-8})(29)}$$

With a higher processor speed the period of a single wavelength will be shorter as more values are being output per second. To calculate the actual frequency at any clock speed this formula applies:

$$ActualFrequency = \frac{1}{(FrequencyInput)(\frac{1}{ClockFrequency})(29)}$$

3.4.2 Frequency Tuner

<u>module freqTrans(Clk, inA, outA, outRFreq);</u>
This component turns a frequency below 17MHz into a period that can be read by the sine wave generator. Instead of calculating the frequency based on a formula this can translate the period for the sine wave module input. This device works with a 500MHz clock. The device takes a clock input, integer inA of the frequency and has an integer output outA that gives a value readable by the sine wave generator. The last output, outRFreq, gives the real frequency that the device will output. The component works by first converting the frequency to nanoseconds. 1 000 000 000 is divided by inA, the resultant period correlates to the frequency. That period in nanoseconds is then divided by 58 to determine the how many simulator clock ticks must pass before the value on the sine wave generator changes. The value in then output to outA. The frequency will be the closest possible value within this configuration. As a result the frequency that the overall device produces will be close to, but not perfectly, in line with the output of this device. The true frequency can be seen in the outRFreq output so the actual frequency input can be found. outRFreq will only reflect the real frequency output if the value is changed by this component. This will not give an accurate frequency for the frequency sweep.

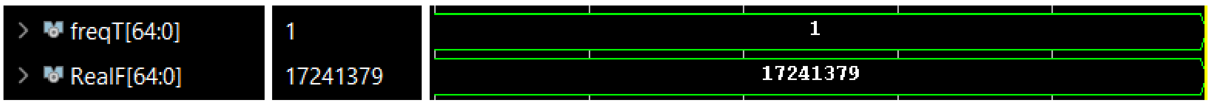


Figure 3.2: Frequency translator input and output

3.4.3 Multiplier

module multiVer(Clk, inA, inB, outAB);
The multiplier is a simple component that was initially coded to multiply the PWM output with the sine wave generator output. This produces a PWM waveform with the magnitude of the sine wave, which can easily be turned into an analog sine wave. The multiplier simply takes in one value from each of the two input signals, at every rising clock edge, multiplies them together and then assigns the resultant value to the output. In the digital schematic a modified version of this component was used to modulate the input signal by multiplying it by a sine or cosine wave. The component is synchronous and accepts a clock input. inA and inB are both 64 bit signed number inputs. outAB is the multiplied signal. It is a 128 bit signed number output.

3.4.4 Adder

module add(Clk, inA, inB, outAB);
The adder was made using a modified version of the multiplier. The operation and the size of the inputs and outputs were changed. This component accepts two 128 bit numbers and outputs a 256 bit number. This is used to add the two modulated waves together to produce the quadrature modulated output. This is a synchronous component. inA and inB are both inputs that are 128 bit signed numbers. outAB is the sum of the signals. It is a 256 signed number output. Similarly to the multiplier, the adder takes in one value from inA and inB and adds them together to generate outAB.

3.4.5 Signal Generator

module SignalGen(Clk, sig_out);
This component creates a saw tooth wave. This was used in the place of a bit stream input. To change this input the values in the array need to be changed. The array accepts up to 38 values, each of 65 bits. The component is set to repeat a 29 number sequence and output this as a wave. By changing the if statement in line 87 of the component it can accept up to a 38 number sequence. This component can be replaced with any digital signal with values of 65 bits or less. Without lab equipment it was difficult to test random bit streams. The component accepts a clock input as the output is synchronous. The output, sig_out, is a 64 bit signed number. This is simply an iterator and an array.

It works in the same way as the sine generator but without the j integer counter. At every rising edge clock an integer, used as the location in the array, is incremented and a correlating value is assigned to sig_out. When it reaches the end of the array the integer is reset to 0.

```
module SigIn(Clk, sig_out, bitIn);
```

A second version of this component, called SigIn.v was developed but untested. It accepted a 1949 bit number as an input which correlated to a 30 value waveform represented in one number. The component would then separate it into portions of 64 bits. This proved impossible to test as it would mean calculating the value of a 1949 bit number. 1949 bits would be a 588 digit decimal number if unsigned. A signed number would be significantly smaller but it was extremely difficult to calculate the value. To input a bit stream the best method would be to input a signal in place of the signal generator component. It accepts a clock input. The output, sig_out, is a 64 bit signed number. The input bitIn is the 1 949 bit signed number that contains the bit stream. This works the same as the signal generator but the array values are set by slicing the bitIn input into 64 bit pieces. Each 64 bit piece becomes a value in the table that is iterated.

3.4.6 Electrical DAC Circuitry

It is worthwhile to mention that the sine wave generator could be directly applied to a digital to analog converter circuitry and produce an analog wave. The circuitry is shown in Figure 3.3. It is an arrangement of a summing amplifier. Each input is a binary digit from most significant to least significant. This configuration would not be suitable for large numbers. The components for this would be inexpensive and the design is simplistic so it would an acceptable replacement for an on-board DAC for small projects. By using external circuitry for this, one could use a cheaper FPGA with fewer gates as the DAC does not need to be approximated in the verilog code. A DAC device can be bought cheaply and would work better than a built circuit or an FPGA approximation of a DAC.

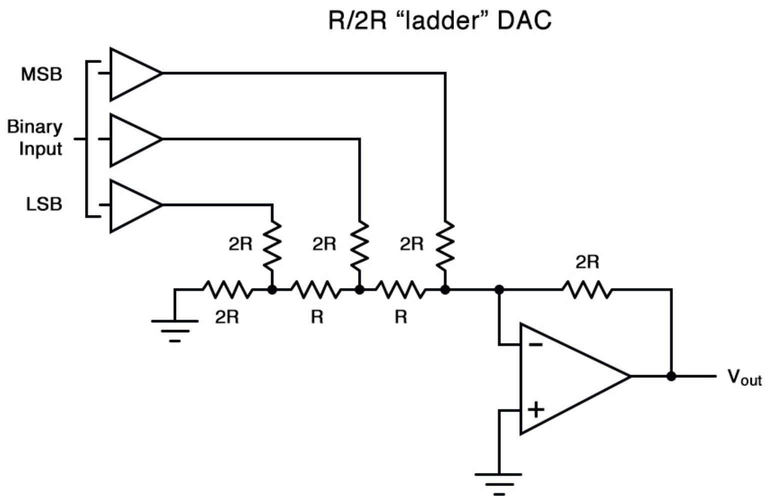


Figure 3.3: Analog circuitry to convert Digital signals to an Analog signal

3.4.7 Components Not Used in Final Version

Digital to Analog Converter(Low pass filter)

module DAC(signal_in, signal_out, clock_in, reset, enable);

The component named digital to analog converter (DAC) in this project, strictly speaking, isn't a DAC. When a PWM wave is passed through a low pass filter the resultant wave is an analog wave. This component accepts a 67 bit input signal, clock input, reset input and enable input. The reset and enable were set to 0 and ignored for the project. This module is a moving average filter developed from code that can be found at www.allaboutcircuits.com. The transfer function for this filter can be seen in Figure 3.4.

$$\begin{aligned} H(\omega) &= \frac{1}{N}\frac{e^{-j\omega N/2}}{e^{-j\omega/2}}\frac{j2\sin\left(\frac{\omega N}{2}\right)}{j2\sin\left(\frac{\omega}{2}\right)} \\ &= \frac{1}{N}\frac{e^{-j\omega N/2}}{e^{-j\omega/2}}\frac{\sin\left(\frac{\omega N}{2}\right)}{\sin\left(\frac{\omega}{2}\right)} \end{aligned}$$

Figure 3.4: The transfer function of the filter in the frequency domain [7]

The digital to analog conversion could be done with circuitry, external to the FPGA, which would be easier to accomplish. A summing amplifier with an input for each bit would perform this function without the moving average filter and the PWM. This low pass filter could be replaced with a simple RC filter, which would be a passive first order low pass filter. Depending on constraints of the particular FPGA this may be preferable. The cut off frequency of such a filter would be $\frac{1}{2\pi RC}$. The circuit diagram can be seen in Figure 3.5. [7]

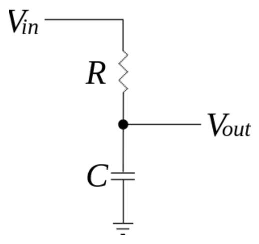


Figure 3.5: RC circuit low pass filter

PWM

module PWNGen(clk, pwm);

The PWM generator creates a series of ”steps” of 1 and 0. The PWM used was a 50% duty cycle. This signal is created to convert a digital waveform into a pulse width modulation waveform.The module accepts a clock input. pwm is the output of the module. It is a one bit output of the PWM signal. A PWM waveform can easily be converted to an analog waveform with a low pass filter. The PWM has a magnitude of 1 so the waveform will maintain the values of the sine wave when multiplied. To create the 50% duty cycle the output is alternated between 1 and 0 at every rising clock pulse.

3.5 Assembly and Test Benches

The assembly was done in Vivado using a file as a ”test bench”. The modules were declared as objects. The inputs and outputs were connected using wire variables, these ”linked” the component objects. Input values were instantiated using register variables. Any wires or registers that needed to be monitored in the log files were added using the $monitor() function. Any comma separated variables in the brackets would have their values stored in the log file. The #600 $finish function dictates how many samples will be used. To set how often the simulated clock ticks this piece of code is used:

```
always begin
#1 clock = clock;
```

end
This sets the clock frequency for the test bench against the simulator clock frequency. The simulator clock frequency was 500MHz. This value can also be set to fractions to allow for test bench clocks faster than the simulator clock.

3.5.1 Analog Version

The first version only produced the sine and cosine waves but was made to generate an analog output. This version could not be fully tested or produced without lab equipment. It contains the DAC and PWM modules to turn the sine wave into a PWM waveform and then low pass filters it to become an analog wave. The schematic for this version can be seen in Figure 3.6.

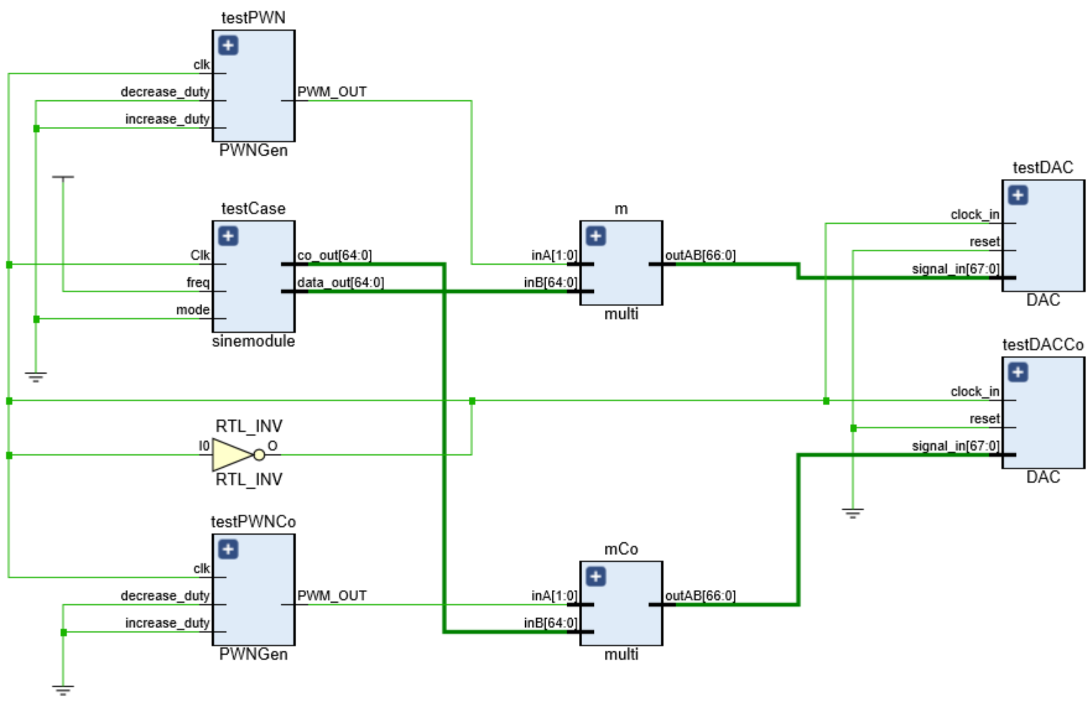


Figure 3.6: Schematic of assembled components for analog output

3.5.2 Main Final Version

Figure 3.7 shows the schematic that was used for the main product of the project. This device is strictly digital and therefore does not involve a DAC of any kind. The sinemodule generates a sine and cosine wave according to the frequency setting when on mode 0. The SignalGen module creates a saw tooth wave. The saw tooth wave is created by iterating saved values in the module. The values or the module can be replaced to allow for an input wave or bit stream. The next modules are the multiVer modules. These multiply the input wave by the sine and cosine wave. They accept one value from the wave and one from the sine/cosine every rising clock pulse. The result is that the modulated waves are added together by the adder. The output from the adder module is the quadrature modulated output. The frequency input to the sine wave generator can be generated using the freqTrans module. This is the frequency translator that finds the closest frequency possible to the freqTrans frequency input.

If the sine module is on mode 0 a sine and cosine wave can be taken from the outputs of this component. The data_out produces a sine wave and the co_out produces a cosine wave. The waves will correspond to the frequency input. For a linear frequency sweep, the mode on the sine module must be set to 1. The output must be taken from the output of the sine module. This creates a linear frequency sweep of both the sine and cosine waves.

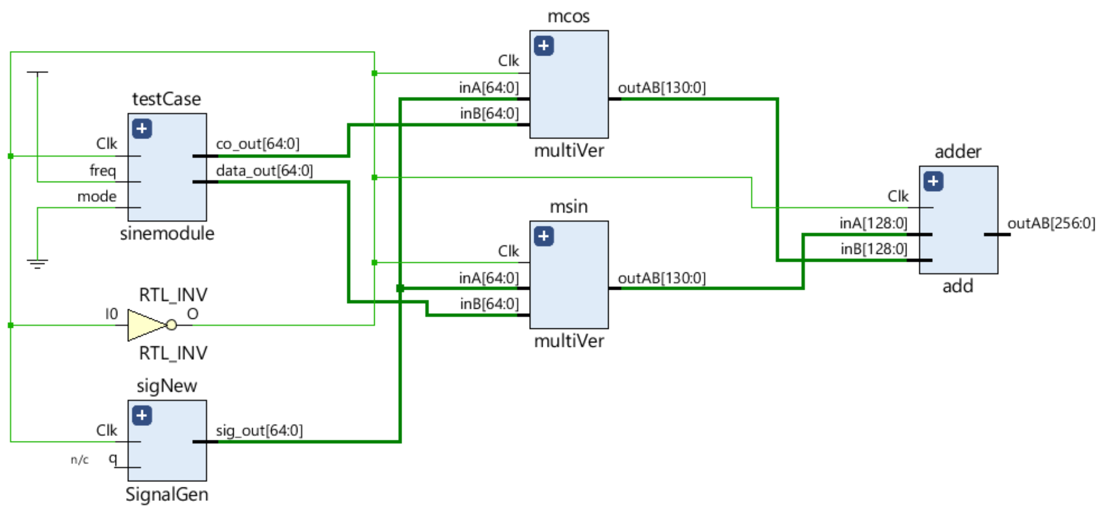


Figure 3.7: Schematic of assembled components - Digital output

3.5.3 High Frequency Version

The high frequency model allows for more gross control over the frequency. In an attempt to allow for a smaller period and higher frequencies, this version uses 8 samples per wavelength. The highest frequency is 125kHz and the period is 8ns. As a result, this version has a significant loss of precision. The values in the array are sine values that have been multiplied by 10 and then rounded to the nearest integer. The data set is show by the set:

$$S = \{0; 7; 10; 7; 0; -7; -10; -7\}$$

As you can see, from the data set, the version is severely lacking in resolution. However, when it comes to modulation, high frequencies are usually preferable. The previous design allowed for a great deal more resolution in the sine wave but the at its highest the frequency was approximately 17kHz. This model is less suited to applications that use low frequency waves.

3.5.4 Ideal Version

This version of the code is possible in a simulation but it would not be possible to replicate it using the hardware available. Within the simulation it is possible to change how often the system sees the rising edge of the clock to a fraction of the clock pulse. This isn't possible in the real world but since the project is purely simulation based a model was developed that gives a wave a period of approximately 1,08ns. The aim was to make the time step as close to 1ns as possible. This was the smallest time step that the simulator would execute. The formula for the frequency is:

$$ActualFrequency = \frac{1}{(FrequencyInput)(1.08 \times 10^{-9})}$$

With this clock value the device would be have an almost fully tune-able frequency. The highest frequency would be approximately 0.925 GHz. The resolution allows periods of multiples of 1.08ns. Ideally the period would be a whole number of nanoseconds so usable frequencies could be attained but 1.08ns was the closest that the simulator could process. This is 0.018 times the normal clock period. Most computer processors exceed 1GHz so it is feasible to achieve this kind of frequency on an FPGA in the future.

Chapter 4

Version Progression

This chapter documents the progression of the project as it was built. While the goal of the project remained constant, the method changed fairly significantly as the project developed. This aims to give a clearer picture of those changes and why they were made. It also aims to help future research by showing which methods were effective and which were tried and abandoned.

4.1 Initial Orientation

Before DDS was attempted, the step was to execute basic interactions with the FPGA. This was to ensure that the version of the IDE was correct and to ensure that there were no errors that would affect later programs. The program was a basic set of instructions to control the LEDs with the switches. This set up a base line for the subsequent code. This method was derived from a tutorial for the Nexys 4 DDR.

4.2 Version One

The goal of this version was to output a sine wave of arbitrary frequency. The model used a Look up table. The output value changed on the rising edge of the clock input. The clock input used was the simulator clock value so the frequency is set and could not be changed at the time. The look up table contained 29 samples. An integer was used as the counter that increased on the rising pulse. A test bench program was needed to run testing simulations. The test bench created an object of the wave generator and inserted values to the inputs. The test bench can adjust how many real clock cycles equals one clock cycle in the simulation. For the purposes of this study we used one to one clock pulse. The test bench also specified how long the simulation would run for. This had been set to 10ns.

4.3 Version Two

The second attempt was much like the first but allowed for some control in the frequency. The processor was 100MHz, which translated to a period of 10ns so the period of the clock was set to be 10ns. This version of the program included a frequency pin. This acted as a multiplier for the period. The multiplier was an integer so any multiple of 10ns would be possible. The program used two counter variables, i and j. The i variable incremented the counter that changes the value from the look up table. The j variable counted the multiplier for the frequency. When the j variable reached the frequency input, variable i was incremented.

4.4 Version Three

This version added the second output for the cosine output. This was done by offsetting a second output by 7 samples. This shifted the wave by 90 degrees. The sine array was made bigger by 7 samples to accommodate the new offset. This version acted as expected in the simulation. This was during the national lockdown so physical testing on a FPGA wasn't possible. The frequency could be varied at this point but the input wasn't in Hertz, it was just an arbitrary multiplier that increased the number of clock ticks before the output was changed. The output was still a whole number so the output needed to be scaled. The array wouldn't work with a decimal number. The output was ”XX”. It is likely that the output put needed to be bigger. It's also possible that it was the result of attempting to index a value in the array that did not exist.

4.5 Version Four

This version added the linear frequency sweep function. The upper and lower limits of the frequency were not established. For testing purposes the minimum frequency was set to 10 clock pulses per sample and the maximum was 1 clock pulse per sample. Each frequency will complete one full wavelength. Afterwards the number of clock pulses between the changing of the value was decreased by 1. This progressed until there was 1 pulse between samples. This version had a ”mode” input which specified whether the device would create a normal sine wave, a linear sweep or quadrature modulation (not yet implemented). The time frame in the test bench needed to be adjusted to be the length of the linear sweep. The sweep was done using a counter. The counter took the place of the frequency input in the previous versions. The counter started at a high number and decreased after each wave length. It marked the number of clock pulses until the value changed. Decreasing this value increased the frequency. The size of the output values were changes to be 64 bit so that they were able to support the negative numbers in the lookup table. The outputs were previously 7 bit which was big enough for the largest value but failed to take the negative numbers into account. The registers assigned to the inputs also needed to be signed.

4.6 Version Five

The system was changed to act on the rising edge and falling edge of the clock. This sped up the program and allowed for finer control with the output frequency. By setting the frequency to 1 in this program it produced a 34.5KHz wave. The value changed once every 2ns. The period is 58ns. 1 divided by 29x10^{-}9 equals a frequency of 17KHz. Using the rising and falling edge of the clock is not supported by vivado for implementation and will fail when the code is synthesised. Using both impulses is supported in some code libraries, and is supported by the simulations. It is unclear whether computing on both edges is supported by the Nexys 4 DDR.

4.7 Version Six

This version included a low pass filter and a PWM generator. The digital waveform was multiplied by an PWM waveform and then passed through a low pass filter to convert it to analog. The digital signal is a 'blocky' waveform that is difficult to immediately turn into an analog wave. However, a PWM signal can be turned into an analog wave by passing it through a low pass filter.

The PWM generator created a 50% duty cycle like the one that can be seen in the figure below. By multiplying the sine wave values by the PWM values, the resultant wave would be a PWM representation of a sine wave which could be converted to an analog wave.

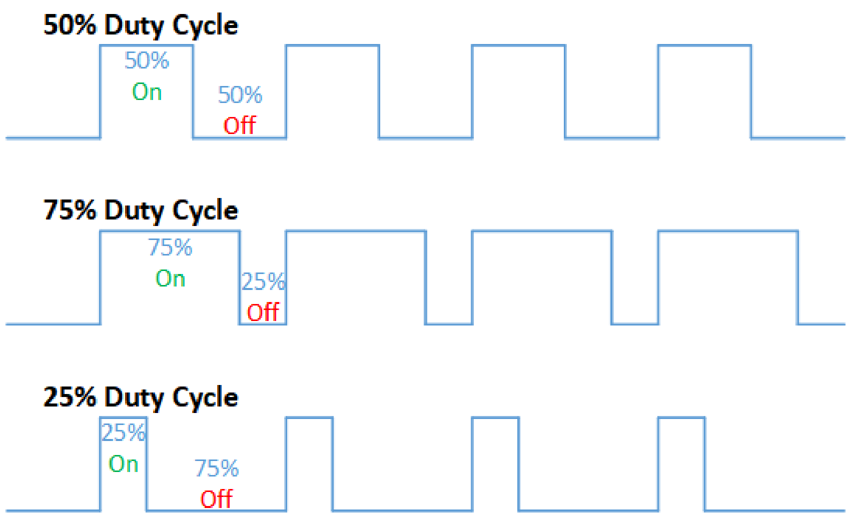


Figure 4.1: PWM example graphic from https://en.wikipedia.org/wiki/Pulse-width$_{modulation}$

The low pass filter was built using the transform function in section 3.4. In practical applications it would make sense to apply an electrical circuit that acts as a low pass filter. For this filter the log_2 of the number of samples was used. [7]

4.8 Final Versions

4.8.1 Main Version

At this point it became clear that without lab equipment testing analog values would not be possible. The PWM generator and low pass filter were removed from the project. The sine wave generator remained the same. A multiplying component was added which multiplies two input waves. A component that adds two waves together was also added. A signal generator that creates a saw tooth wave was added. The saw tooth wave was multiplied separately by the sine and cos wave using the multiplying units. The two resultant waves were added together by the addition component and produced a quadrature modulated wave. The sine wave generator was capable of producing a linear frequency sweep. A frequency translator was added which converted the input frequency to the closest achievable frequency. The frequency input to the model was previously entered as a period.

4.8.2 Alternate Version - High Frequency

This is version used the same components and logic as the main version but used a slightly different sine wave generator. The sine wave generator used 8 values instead of 29 and created a wave form with a much higher frequency to the main version but sacrifices resolution.

4.8.3 Alternate Version - Analog Version

This was the assembled version that included the low pass filter and the PWM generator. This did not perform quadrature modulation but theoretically could produce an analog linear sweep. The functionality of this version was unproven but was included for the sake of completeness.

Chapter 5

Results and Discussion

5.1 Unit Testing

For this section, each component of the device was simulated separately to ensure each component worked before the device was assembled. The components were loaded into separate Vivado projects and then assigned dummy inputs, by a test bench, so it could be simulated and the outputs could be graphed. Unit testing is essential in a complex project like this where it can be difficult to find an error when the components are presented as a system. If a single component fails in unit testing it is certain that the error is with that unit.

5.1.1 Sine Wave Generator

As it can be seen below, the sine wave generator was functional. It changed in value at each rising edge of the clock pulse. The wave changed at regular intervals and can be seen to oscillate between -77 and 77. This value would be adjusted to a fraction of a voltage when output in analog form. The result can't be graphed as a sine graph using Vivado but the results were extracted from the log files and graphed in section 5.2.1. From the Vivado diagram we can calculate the period and frequency of the wave. This test should produce the maximum frequency as the freq input is set to one in the test code.

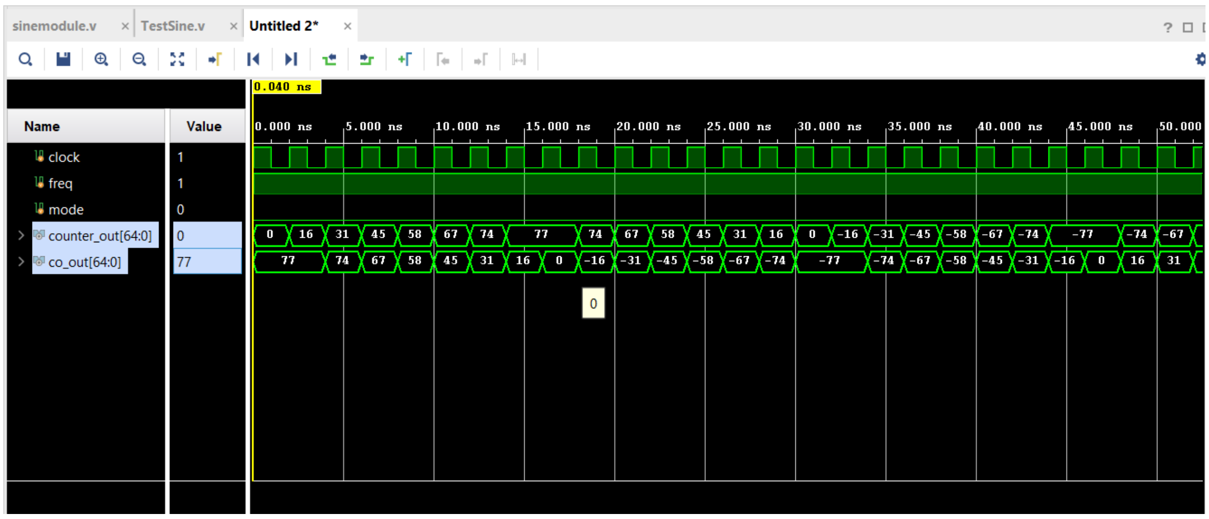


Figure 5.1: Simulated Sine output from Vivado

5.1.2 PWM Generator

This version tested two different PWM generators. One was a verilog PWM generator and the other was a VHDL PWM generator. While both served the same function it was uncertain which method would be best as the VHDL model appeared in research and the verilog model was created without external code. As it can be seen by Figure 5.2, the VHDL model failed to produce the desired wave pattern. The verilog model produced the correct wave which has a 50% duty cycle.

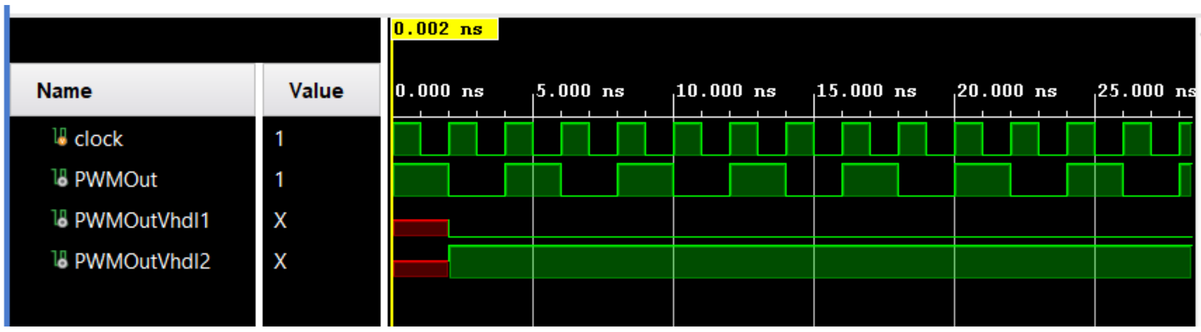


Figure 5.2: Simulated PWM output from Vivado

5.1.3 Digital to Analog Converter

The DAC was tested to ensure the low pass filter would turn the PWM signal into an analog signal. As it can be seen in figure 5.3 below, the test failed. It seemed that it might not be possible to convert digital data to analog data within Vivado. It was also possible that this step would have needed to be done with a DAC or external circuitry. The FPGA used was relatively cheap hence the lack of a DAC. It is possible to use the XADC chip on the device to convert analog signals to digital but not vice versa. As discussed earlier, this was not tested further as unforeseen circumstances moved the project from practical to purely digital testing.

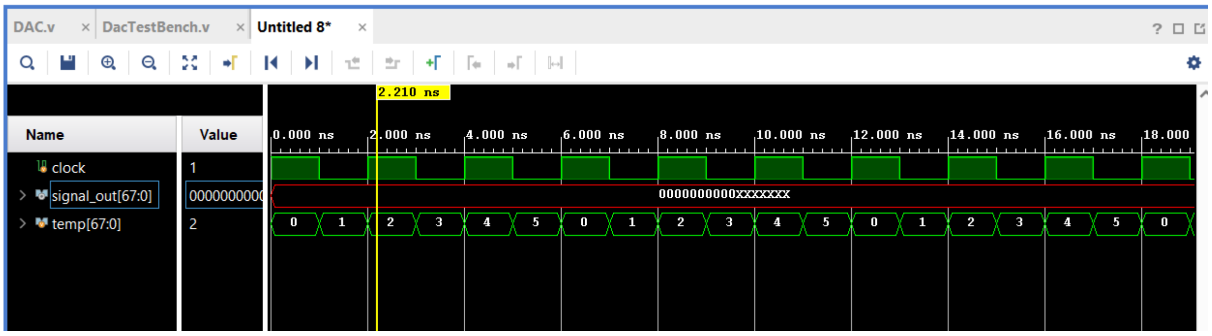


Figure 5.3: Simulated DAC output from Vivado

5.1.4 Multiplier

This was a simple unit to multiply two inputs, one 2 bit input and one 64 bit input, being the PWM wave and the sine wave respectively. The PWM wave would be 1 or 0. The sine input ranged from - 77 to 77. The expected output was seen in the simulation when the PWM wave was set to 1 and was multiplied by 4D (which is 77 is hexadecimal). The simulation generated an output of 4D which can be seen in figure 5.4. The device can easily be made asynchronous for an analog input. Seeing as the analog outputs could not be tested, a synchronous version was used for testing. An earlier version used a VHDL model for this but it was abandoned in favour of verilog. This test was performed to ensure the mechanics of the program were correct. The input and output sizes were changed for some uses but unit testing was not performed again as the logic of the code was proven to be correct.

5.1.5 Adder

As the adder was a modified version of the multiplier it was not tested individually. Like with the other multipliers (that had altered inputs) this didn't necessarily need separate unit testing as the logic was already proven to be correct when testing the multiplier.

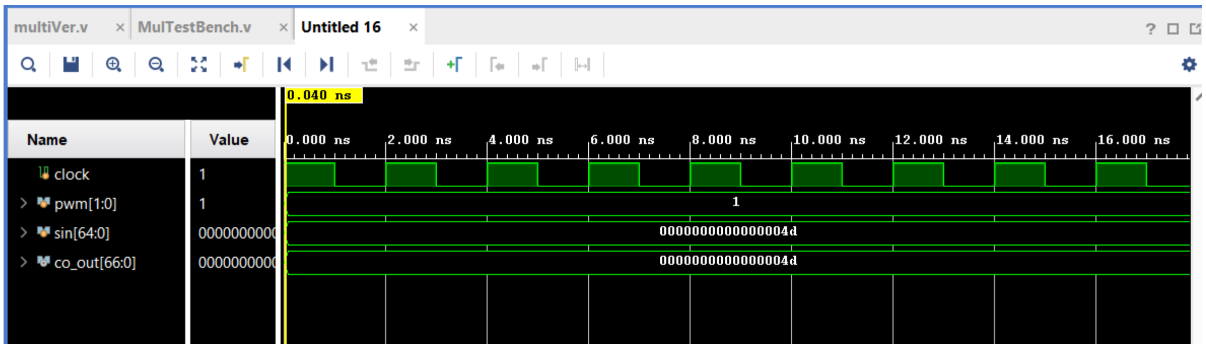


Figure 5.4: Simulated Multiplier output from Vivado

5.2 Simulation Results

Each graph was made using log file data after running the simulation. 600 samples were taken.The x axis is time in nanoseconds.

5.2.1 Sine Wave

As seen in the unit testing, the simulation could produce the values of the sine wave from the saved data in the array. In Figure 5.5 the sine wave output can be seen. It oscillated between 77 and -77. This could be adjusted by multiplying or dividing the output. The output would naturally be scaled to a fraction of the voltage if the wave was converted to analog.

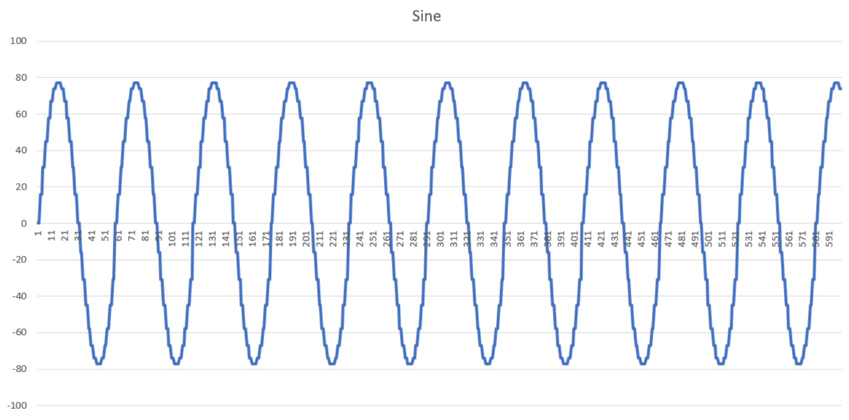


Figure 5.5: Sine Output vs Time(ns) - 600 samples

In Figure 5.6 the sine and cosine wave are overlaid. This figure illustrates that the produced sine and cosine values are offset by 90 degrees. They are visibly of the same frequency and amplitude. The tabulated results underlying the graph confirmed this.

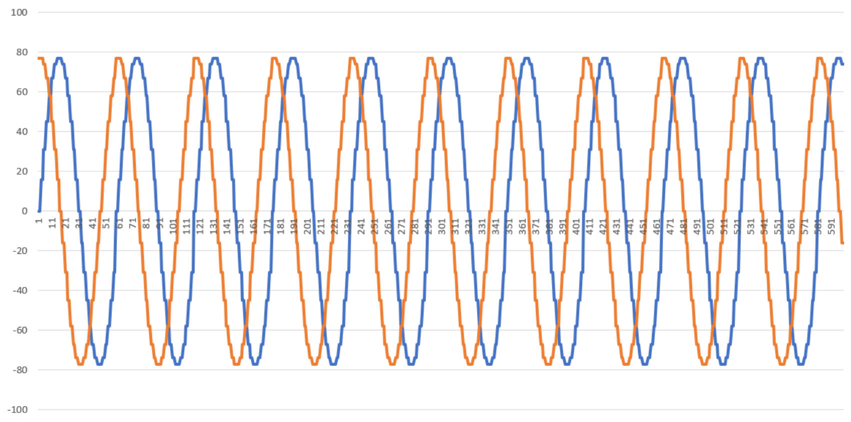


Figure 5.6: Sine and Cos Output vs Time(ns) - 600 samples

5.2.2 Frequency Sweep

The frequency sweep of the sine wave is illustrated in figure 5.7 below. As time progresses, in the graph, the period of the wave increases. The graph shows longer wavelengths towards the right. Each subsequent wavelength is 58ns longer than the last. The output depicted in figure 5.7 is derived from 600 samples. If more samples are used a greater progression will apparent as the period becomes longer. Theoretically the period could extend infinitely till the frequency reaches (approximately) 0. After each wavelength the period increases by 58ns. The table in figure 5.8 shows the period of each wavelength, in

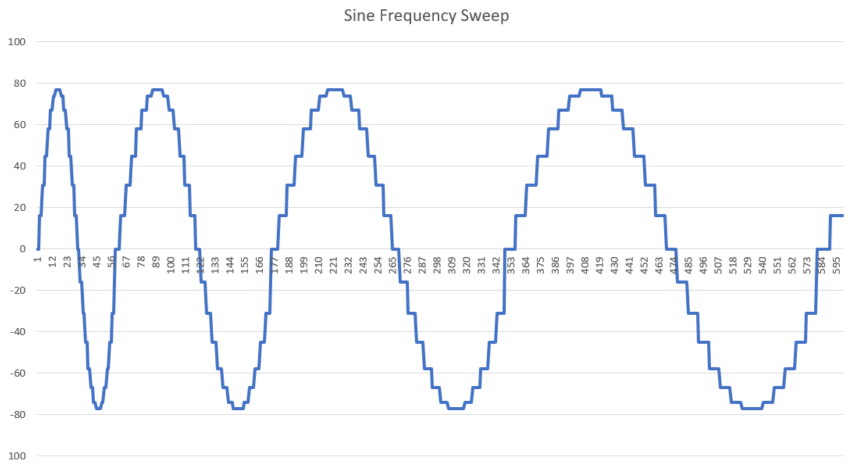
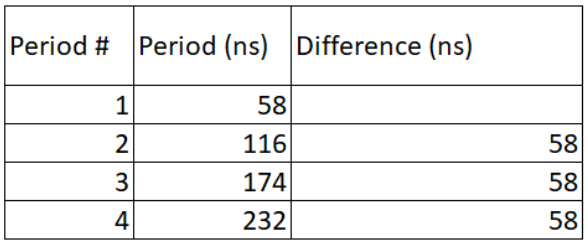


Figure 5.7: Simulated Frequency Sweep vs Time(ns) - Sine Output - 600 samples

nanoseconds. The difference between the periods is constant at 58ns.



Period #	Period (ns)	Difference (ns)
1	58	
2	116	58
3	174	58
4	232	58

Figure 5.8: Table showing the differences in the periods of the linear frequency sweep

5.2.3 Quadrature Modulation

A saw tooth wave was an input to the quadrature modulation to be multiplied by a sine wave and a cosine wave. The resultant modulated signals are illustrated in Figure 5.9. The frequency of the sine wave was too low to adequately modulate saw tooth wave. A lower frequency saw tooth wave could have been used to compensate. However, this test showed the limitations of the device to modulate signals close to of higher than it's highest frequency. With a low pass filter the wave could be tested to see if it is still possible to demodulate the signal successfully. Demodulation is beyond the scope of the project. Without demodulation the output had to be examined graphically. The similarity between the frequency of the carrier waves and the saw tooth wave can be seen from the low number of oscillations in the graph. The saw tooth wave is purely positive so it is possible to determine the period of the sine wave relative to the saw tooth wave. The first peak seen in the graph is followed by a drop that is the result of the saw tooth wave. The graph continues to be positive which means that less than a period of the sine wave has passed. After it becomes negative and reaches 0 for the forth time a single period of the sine wave has passed. This means that the sine wave frequency was lower than that of the saw tooth wave. Because of this it was highly unlikely that the signal would be demodulated successfully.

Figure 5.10 is graph of the unmodulated saw tooth wave. When compared with the frequency of the sine wave in figure 5.5 the frequencies are fairly similar which is why the modulated wave would be difficult to demodulate. Modulating a lower frequency wave would be more suitable for the sine wave generated by this version of the quadrature modulation.

Figure 5.11 is the graph of the fully modulated output. This is the sum of the sine and cosine waves after being multiplied by the saw tooth wave. This is the form in which a quadrature modulated signal would be sent to the receiver. At the receiving end this wave form would be multiplied by a sine or cosine wave of the same frequency of the

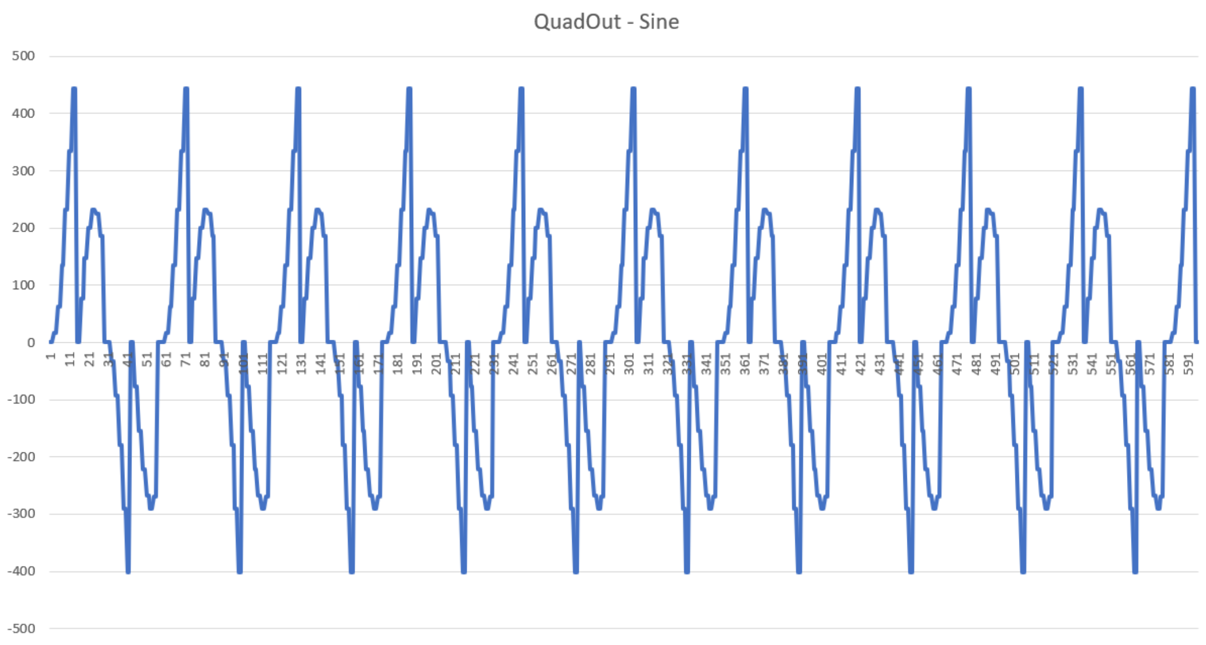


Figure 5.9: Sine Modulated Output vs Time(ns) - 600 samples

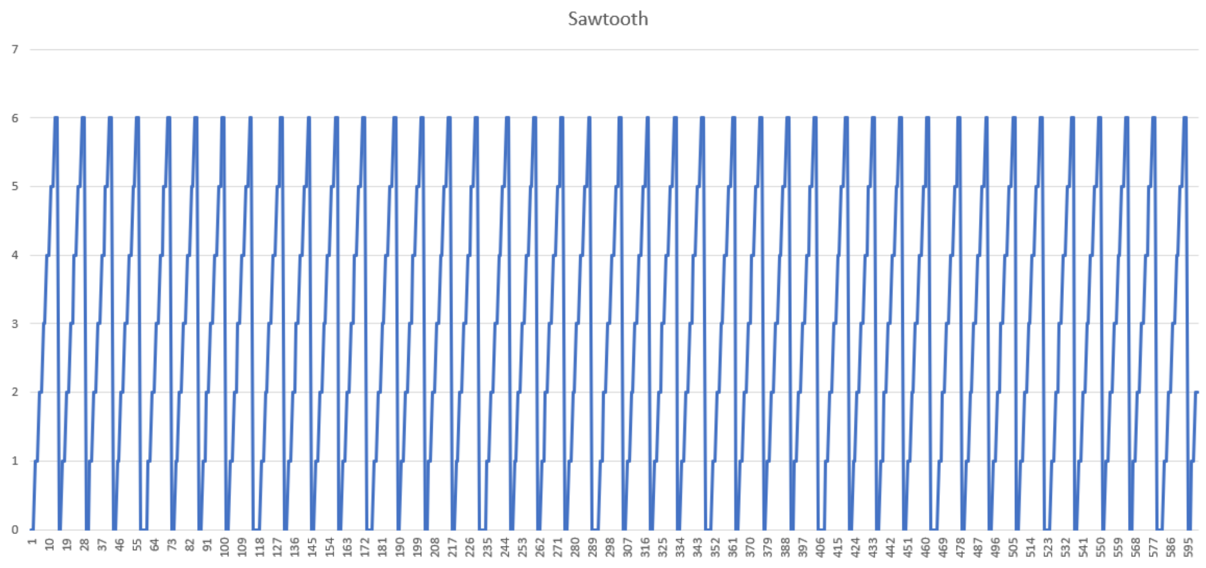


Figure 5.10: Unmodulated Saw Tooth Wave vs Time(ns) - 600 samples

modulation and then passed through a low pass filter to demodulate. Higher frequencies are easier to separate from the sent signal. The modulation was successfully executed.

Figure 5.12 graphs a saw tooth wave, with a much lower frequency than the previous test, modulated with the same sine wave. The oscillations from the sine wave can be seen which shows the high frequency component of the sine wave. The saw tooth wave had a much longer period and it completed approximately one wavelength within the 600

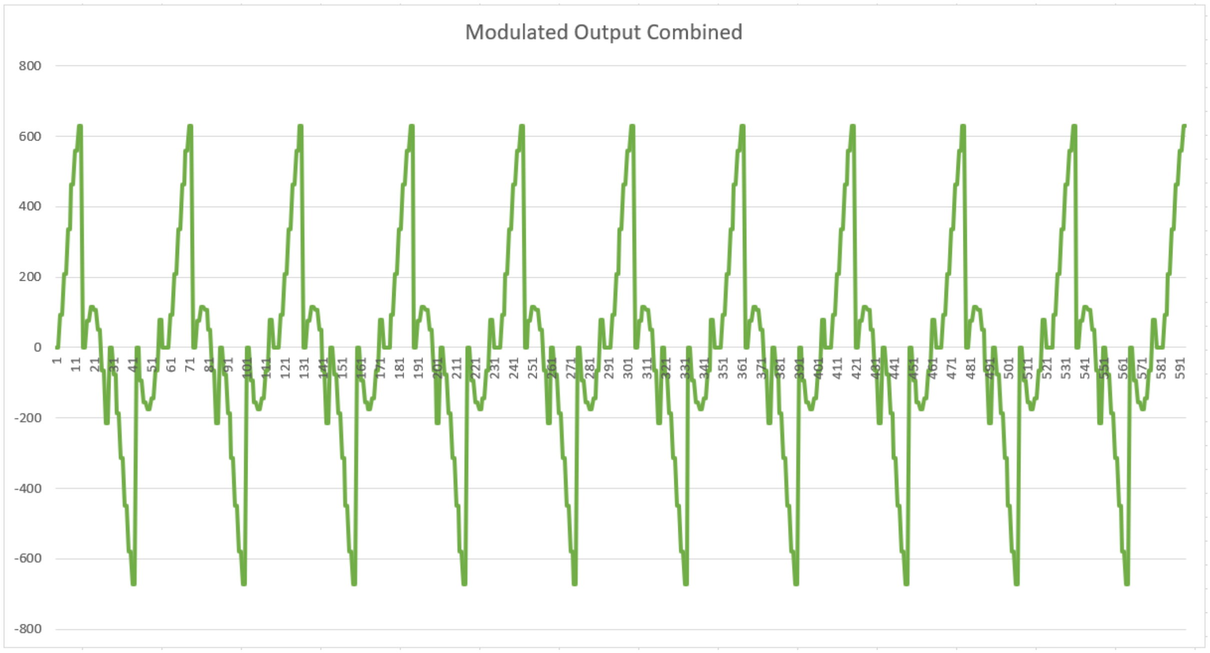


Figure 5.11: Quadrature Modulated Output vs Time(ns) - 600 samples

samples. The 600 samples were executed over 600 nanoseconds which means this wave had a frequency of 1,7kHz. This is still a fairly high frequency and the program is able to modulate it to a wave that would likely be easily demodulated.

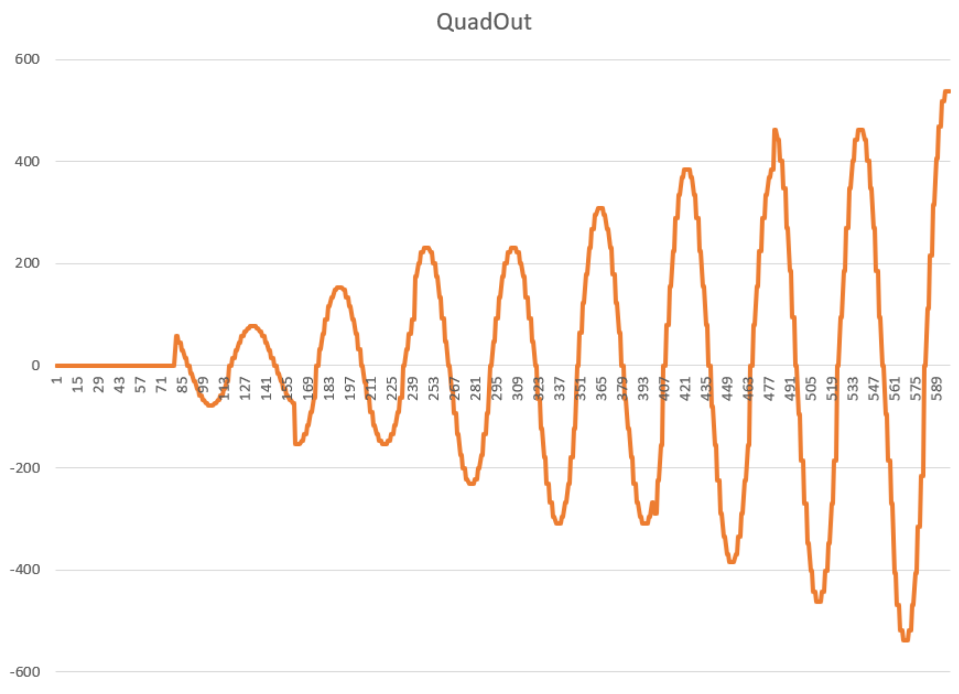


Figure 5.12: Sine Modulated Output vs Time(ns) - 600 samples

Figure 5.13 shows the quadrature modulated wave of the slower saw tooth wave. The result of adding the sine and cosine wave was a reduced amplitude. The higher frequency carrier wave is still evident in this output.

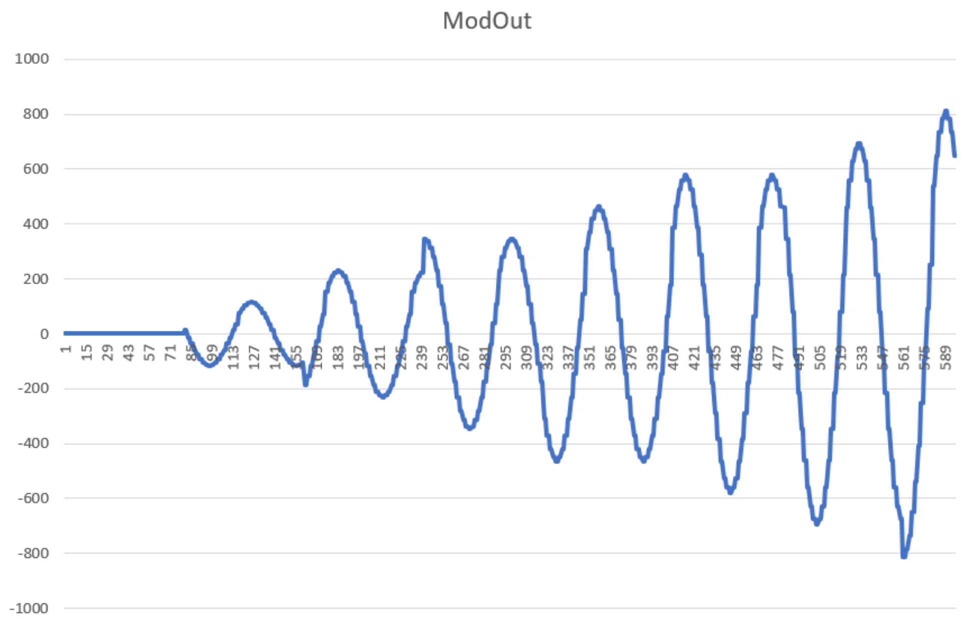


Figure 5.13: Quadrature Modulated Output vs Time(ns) - 600 samples

5.3 Simulation Results - High Frequency Model

5.3.1 Sine Wave

Figure 5.14 shows the highest frequency available of the sine wave on this version of the device, which is 125kHz. The value oscillates between 10 and -10.

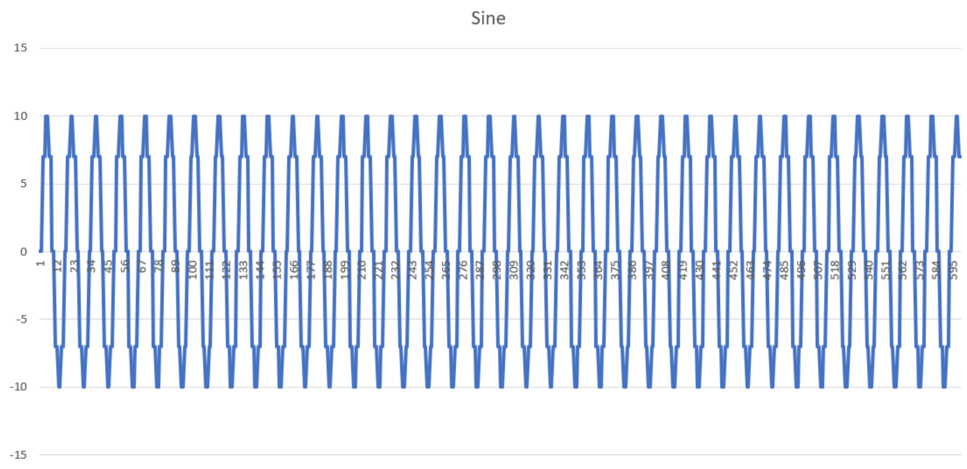


Figure 5.14: High Frequency Sine Output vs Time(ns)

5.3.2 Linear Frequency Sweep

The linear sweep can be performed with this version but as the frequency decreases it no longer resembles a sine wave.

5.3.3 Quadrature Modulation

Modulation was where the high frequency model became useful. Signals are usually modulated using very high frequency waves. Figure 5.16 depicts a high frequency saw tooth wave modulated with a sine wave.

The quadrature modulated version of this wave also benefits from the separation of the frequency of the signal and carrier waves. To demodulate a quadrature modulated signal the signal must be multiplied by the carrier wave and then low pass filtered. Because

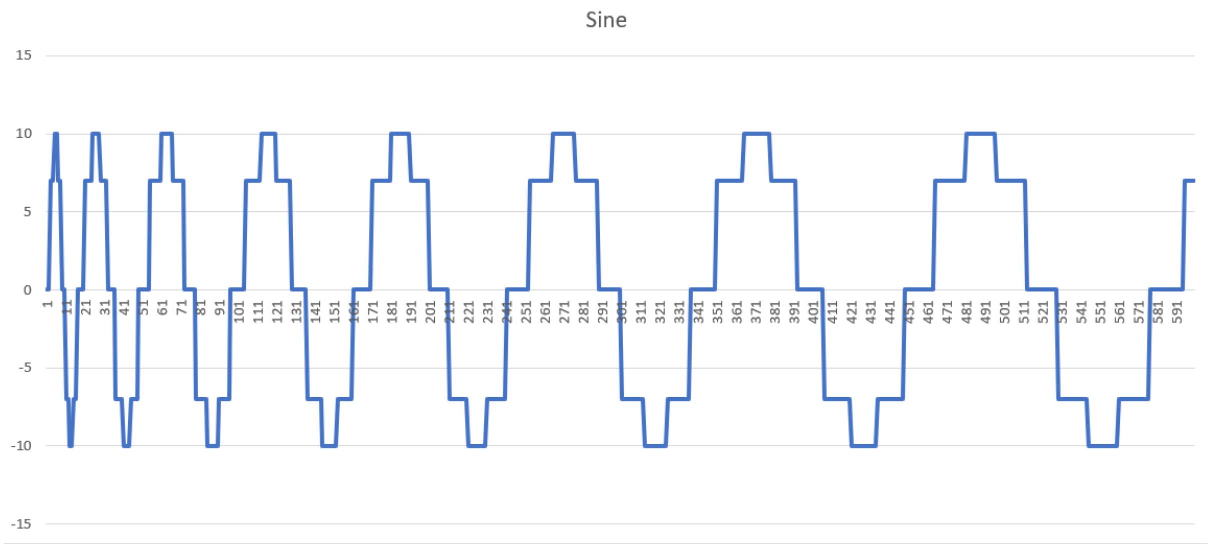


Figure 5.15: Lower Resolution Linear Sweep vs Time(ns)

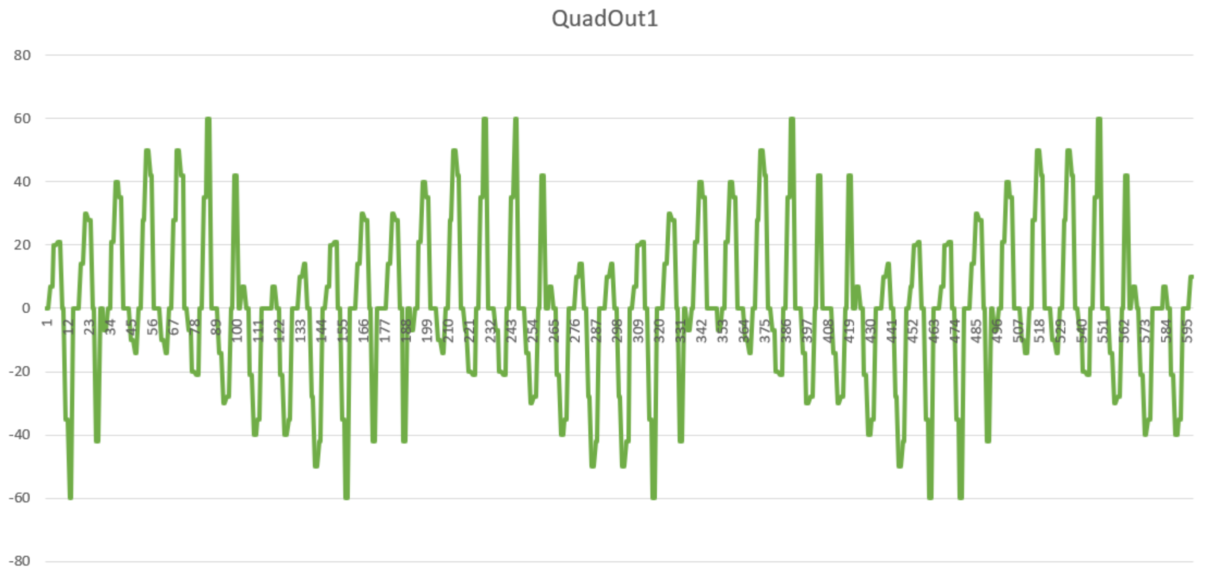


Figure 5.16: Lower Resolution Single Modulated Output vs Time(ns)

the frequency bands of the waves are so different there is little chance that a low pass filter would retain any of the carrier wave. Real life low pass filters do not have the steep rolloff that an ideal filter would have so separation of the frequency of the signal and the carrier waves is key.

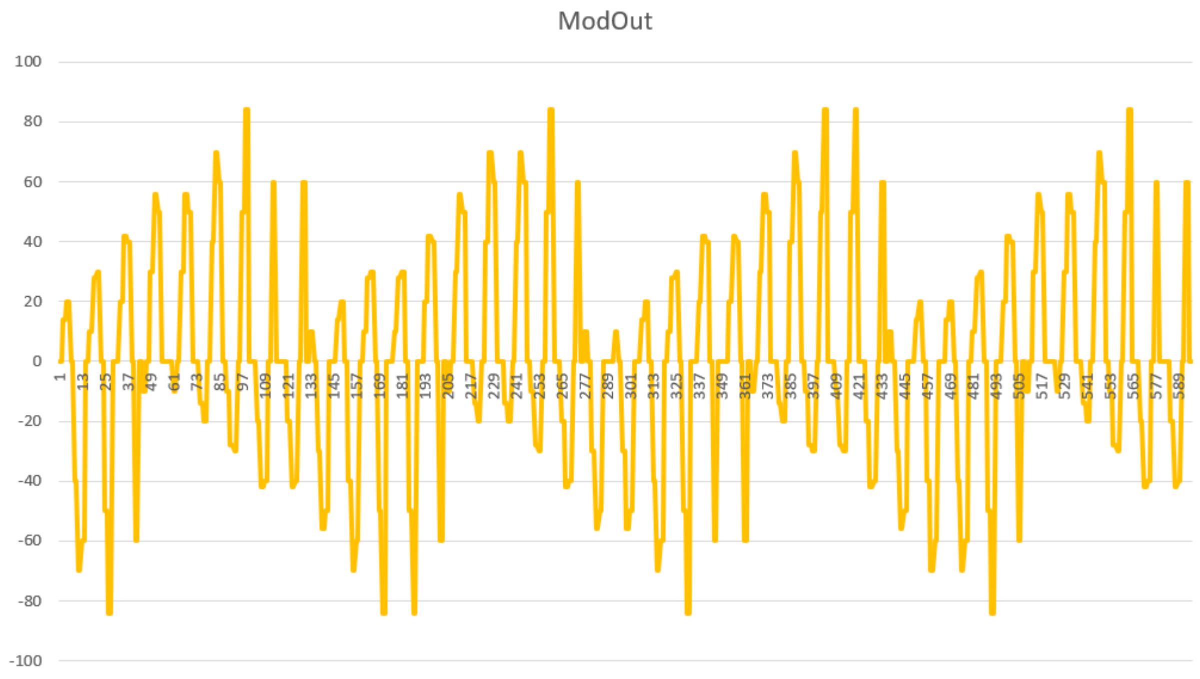


Figure 5.17: Lower Resolution Quadrature Modulated Output vs Time(ns)

5.4 Simulation Results - Ideal Model

The resultant graphs for this are unremarkable as they show the same values as the first set of simulation results with a different timescale. In Figure 5.18 the simulation results show the changed clock frequency and the resultant modulated outputs. As stated earlier, this implementation is impossible on an FPGA. The simulated processor frequency was adjusted to be approximately 1GHz which is a possible processor frequency and is exceeded by most computers. This processor frequency is likely to eventually be possible in FPGAs and a tuned frequency with a higher resolution could be developed. It would be possible to modulate waves with higher frequencies using this model.

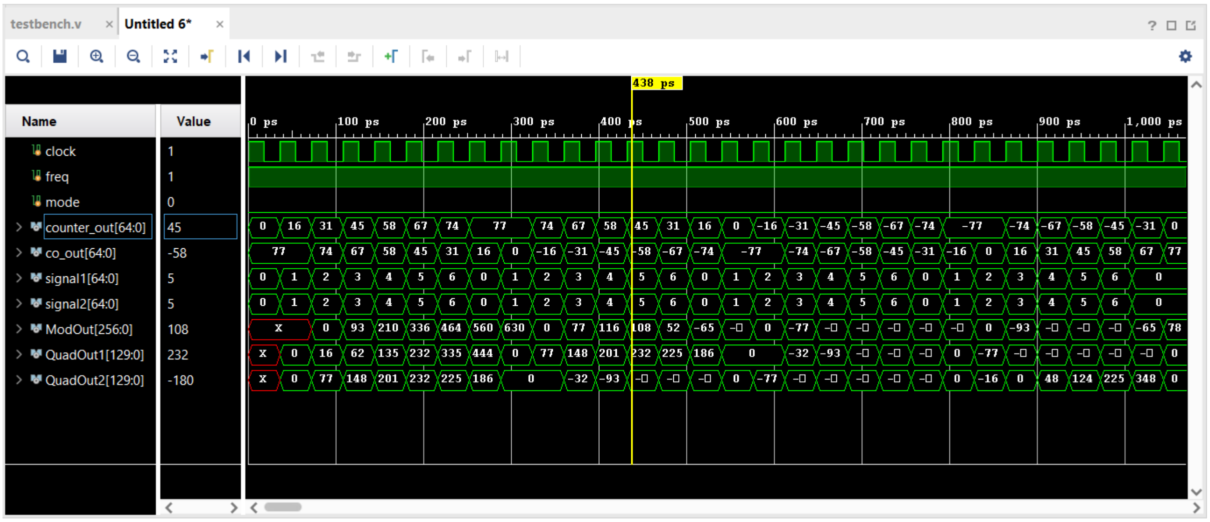


Figure 5.18: Ideal Model Simulation Output

5.5 Experimental Results

These results were calculated on Microsoft Excel using the same log data as the graphs. The operations performed by the project were replicated in excel and compared to ensure that the output was entirely correct. The simulated values were then compared with the calculated values.

5.5.1 Period and Frequency

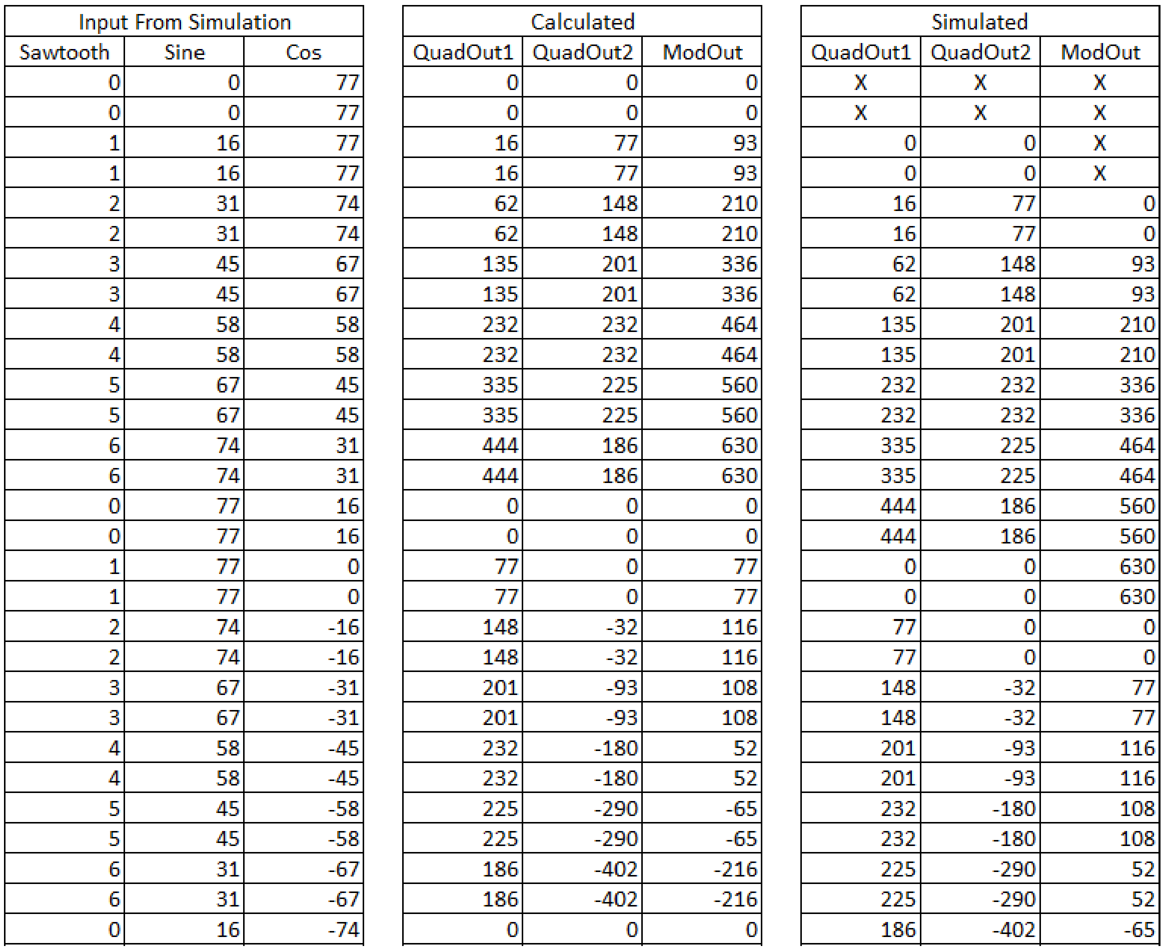
The periods of the sine wave and cosine waves should be 58ns as there are 29 sample values and the value of the output changes every 2ns. With the values taken from the log file the period was calculated manually as per the table below(figure 5.19). Each wavelength takes 58ns which was the predicted value. The frequency was then calculated using the period. These values are the lowest period and highest frequency that can be obtained with this device. This proves that the waves are being produced with the correct period, and therefore frequency.

	Period(ns)	Freqency(Hz)
Sine	58	17241379.31
Cosine	58	17241379.31

Figure 5.19: Period and Frequency calculated from log values

5.5.2 Accuracy of Quadrature Modulation

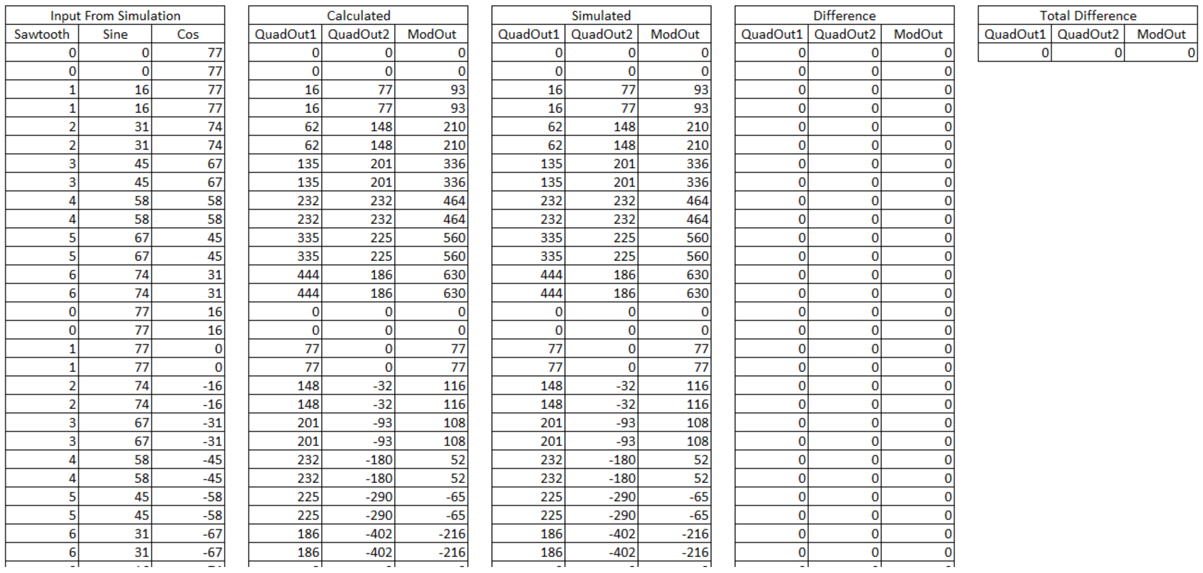
To test the accuracy of the results from the system, for the quadrature modulation, the input values were used to manually calculate the output. The sine and cosine wave output values and the saw tooth wave values from the log files were used as inputs. The values of the saw tooth wave were multiplied by the sine and cosine waves separately and then the resultant values were added together. In the simulated results there is repetition of the values as there are log values for each nanosecond and values only change every two nanoseconds. The simulated modulated waves have a blank or x values as the first two outputs. This is because it took one clock cycle (or two nanoseconds) for an output to reach the component. The original table is included as Figure 5.20.



Input From Simulation			Calculated			Simulated		
Sawtooth	Sine	Cos	QuadOut1	QuadOut2	ModOut	QuadOut1	QuadOut2	ModOut
0	0	77	0	0	0	X	X	X
0	0	77	0	0	0	X	X	X
1	16	77	16	77	93	0	0	X
1	16	77	16	77	93	0	0	X
2	31	74	62	148	210	16	77	0
2	31	74	62	148	210	16	77	0
3	45	67	135	201	336	62	148	93
3	45	67	135	201	336	62	148	93
4	58	58	232	232	464	135	201	210
4	58	58	232	232	464	135	201	210
5	67	45	335	225	560	232	232	336
5	67	45	335	225	560	232	232	336
6	74	31	444	186	630	335	225	464
6	74	31	444	186	630	335	225	464
0	77	16	0	0	0	444	186	560
0	77	16	0	0	0	444	186	560
1	77	0	77	0	77	0	0	630
1	77	0	77	0	77	0	0	630
2	74	-16	148	-32	116	77	0	0
2	74	-16	148	-32	116	77	0	0
3	67	-31	201	-93	108	148	-32	77
3	67	-31	201	-93	108	148	-32	77
4	58	-45	232	-180	52	201	-93	116
4	58	-45	232	-180	52	201	-93	116
5	45	-58	225	-290	-65	232	-180	108
5	45	-58	225	-290	-65	232	-180	108
6	31	-67	186	-402	-216	225	-290	52
6	31	-67	186	-402	-216	225	-290	52
0	16	-74	0	0	0	186	-402	-65

Figure 5.20: Simulated and Calculated Values before orientation - 600 samples

For the sake of clarity, and to simplify the calculations, the blank values were erased. With these 'x' values erased, the simulated values correspond to the calculated values. In the 'Differences' columns the calculated values are subtracted from the simulated values to identify any discrepancies. The sum of these difference values can be seen in the next set of columns. The sum of the differences is zero, thus indicating that the calculated values and the simulated values are the same.



Input From Simulation			Calculated			Simulated			Difference		
Sawtooth	Sine	Cos	QuadOut1	QuadOut2	ModOut	QuadOut1	QuadOut2	ModOut	QuadOut1	QuadOut2	ModOut
0	0	77	0	0	0	0	0	0	0	0	0
0	0	77	0	0	0	0	0	0	0	0	0
1	16	77	16	77	93	16	77	93	0	0	0
1	16	77	16	77	93	16	77	93	0	0	0
2	31	74	62	148	210	62	148	210	0	0	0
2	31	74	62	148	210	62	148	210	0	0	0
3	45	67	135	201	336	135	201	336	0	0	0
3	45	67	135	201	336	135	201	336	0	0	0
4	58	58	232	232	464	232	232	464	0	0	0
4	58	58	232	232	464	232	232	464	0	0	0
5	67	45	335	225	560	335	225	560	0	0	0
5	67	45	335	225	560	335	225	560	0	0	0
6	74	31	444	186	630	444	186	630	0	0	0
6	74	31	444	186	630	444	186	630	0	0	0
0	77	16	0	0	0	0	0	0	0	0	0
0	77	16	0	0	0	0	0	0	0	0	0
1	77	0	77	0	77	77	0	77	0	0	0
1	77	0	77	0	77	77	0	77	0	0	0
2	74	-16	148	-32	116	148	-32	116	0	0	0
2	74	-16	148	-32	116	148	-32	116	0	0	0
3	67	-31	201	-93	108	201	-93	108	0	0	0
3	67	-31	201	-93	108	201	-93	108	0	0	0
4	58	-45	232	-180	52	232	-180	52	0	0	0
4	58	-45	232	-180	52	232	-180	52	0	0	0
5	45	-58	225	-290	-65	225	-290	-65	0	0	0
5	45	-58	225	-290	-65	225	-290	-65	0	0	0
6	31	-67	186	-402	-216	186	-402	-216	0	0	0
6	31	-67	186	-402	-216	186	-402	-216	0	0	0

Total Difference		
QuadOut1	QuadOut2	ModOut
0	0	0

Figure 5.21: Simulated and Calculated values after orientation - 600 samples

In figure 5.22 the values of the two sets have been graphed. The graphs completely overlap as the values are identical. To illustrate this better, the first 300 outputs of the simulated outputs were plotted over the full 600 outputs of the calculated set. The first half (blue) of the graph illustrates the simulated output and the second half of the graph (orange) illustrates the calculated output.

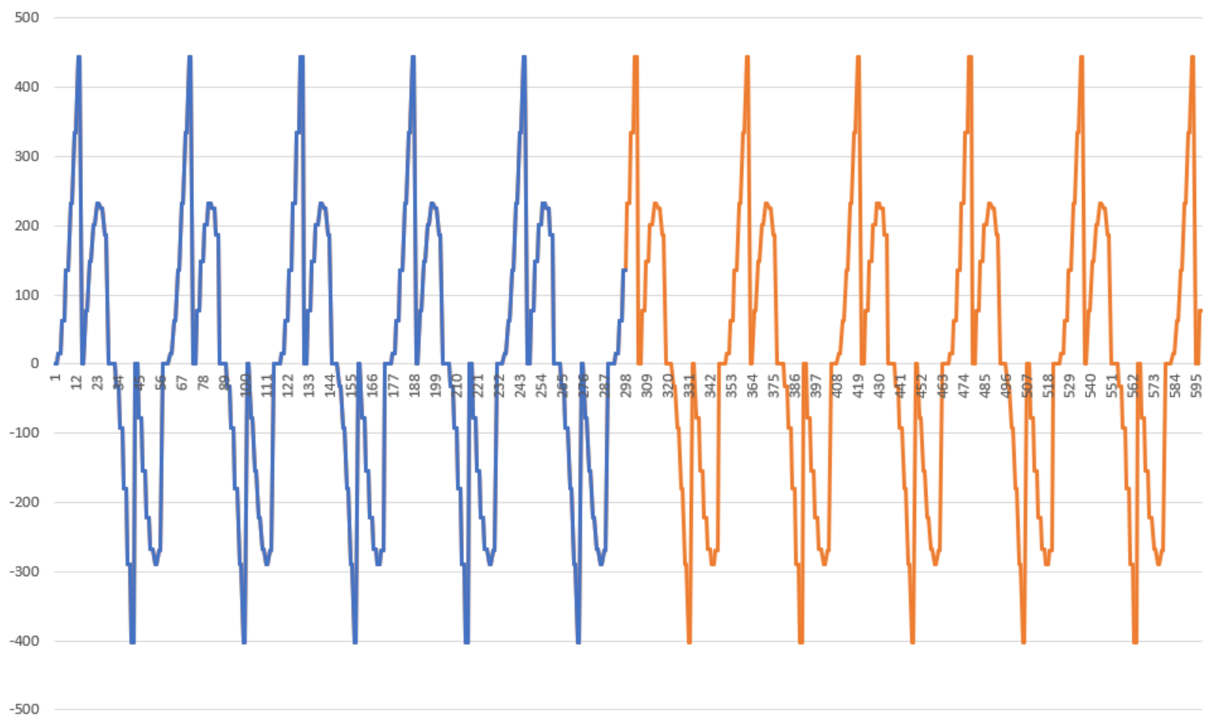


Figure 5.22: Simulated and Calculated Modulated Outputs - 300 Simulated Outputs(1 - 300) and 300 Calculated Outputs(301 - 600

Chapter 6

Conclusions

The objective of this study was initially to configure an FPGA to function as a Direct Digital Synthesis integrated circuit. The device designed to output a linear frequency sweep and accept an input bit stream which would be modulated using quadrature modulation. The methodology shifted from using physical hardware to a simulated device that produces digital waves with a tunable frequency. The simulated device needed to produce a linear frequency sweep and perform quadrature modulation on an input wave or bit stream.

Three final versions were created;

- a high frequency which sacrificed resolution for higher frequency
- an ideal version which was implemented with an unrealistic and very high speed clock
- a main version which balanced resolution and frequency

The high frequency model performed well at high frequencies but poorly at low frequencies. The ideal version was impractical and would be impossible to implement on an FPGA. The main version had a high resolution and performed well at low frequencies and at the high frequencies that it could reach.

The main version was implemented in parts; a sine wave generator, an input signal generator, multipliers and an adder.

The sine wave generator is tunable although not quite as finely tunable as a direct digital synthesis integrated circuit. The frequency is calculated based on a period in nanoseconds that differs based on the clock frequency. As such it is only capable of multiples of this period (58ns) which does not give fine control of the frequency. With the small period of 58ns a wide range of frequency values can be generated. A highly tunable model was included but it can not be implemented with an FPGA. The frequency translator module can be used to increase the accuracy of tuning in the main version.

The sine wave generator did produce a linear frequency sweep which decreased in frequency after each period. The quadrature modulation was accomplished and proven to be correct (through manual calculations). The multiplier used to multiply signals generated with the sine or cosine waves. The adder then summed the two resultant signals. The adder and multiplier both produced reliable results based on testing.

The modulation of an input bit stream was implemented with two methods to input the bit stream. The values in the array within the signal generator can be replaced with the values from the bit stream. Alternatively, a component was developed that accepts the entire bit stream as a single number and turns it into a usable signal.

Due to the facts presented above, the simulated model is not a perfect version. It would need to be tested with hardware before it could have any real world applications. Improvements would need to be made before it could fully replace an integrated circuit for this application. As a simulation it produces insightful results in line with the requirements. Thus it theoretically would serve the intended purpose.

Chapter 7

Recommendations

7.1 Implementation

7.1.1 Hardware

This project was only tested using simulations due to the lack of lab equipment. The first area of improvement for this study would be to apply the theory and simulated results to create a working prototype. In order for this to be implemented on an FPGA the memory constraints would need to be assessed. The simulation used large values for the stored numbers, registers and wires. The number of bits may need to be reduced to cater for the hardware limitations. The clock frequency will need to be accounted for. The clock frequency of the simulation is five times higher than that of the FPGA that was considered so the calculated input frequencies would need to be recalculated (for the period and the frequency translator). A FPGA with a high clock frequency is ideal but cost would need to be considered as it would likely be more expensive.

The Input and output wires need to be properly assigned. In the case of the Nexys 4 DDR (that was since replaced with the simulator) the inputs and outputs would need to be specified clearly and would not be automatically placed. The code would need to communicate with the XADC chip to access the IO ports and dedicate an appropriate number of bits per wire.

7.1.2 Analog Signals

All values in this project were kept digital. In the real world there is a need for analog signals with direct digital synthesis. The Nexys 4 DDR doesn't have a digital to analog converter so an external converter would be necessary. A built converter, using the schematics provided, would work but a cheap DAC chip would allow for larger inputs and outputs than a 'DIY' version. The analog version would be able to convert the sine wave by scaling it down to a fraction of the voltage of a pin.

7.1.3 Frequency Tuning and Sample Sets

The frequency of the sine wave created needs to be more finely tuned for practical application. A useful addition would be to automatically output frequently used frequencies (like 2kHz, 5KHz, 10kHz, etc). The number of samples needed could be examined to find a better equilibrium between resolution of the wave, frequency and a period that can be easily manipulated. The high frequency version had too few samples for some applications but the main version had an initial frequency that wasn't very high and was difficult to adapt to a desired frequency.

7.2 Testing

7.2.1 Simulation Software

The simulation software used for this project is not the best software available. ModelSim is one such program that would interface with Vivado and produce better graphs within Vivado simulations. The ModelSim version acquired wasn't a version that could work with Vivado. A similar software could be implemented to produce better simulations before implementation.

7.2.2 Lab Equipment

In addition to the previously mentioned analog signals, an oscilloscope and a signal generator could allow for better testing with more accurate results.

7.2.3 Demodulation

Demodulation was not within the scope of this project but it would be a worthwhile test to ensure the signals can be demodulated. The frequencies of the carrier waves can be changed determine the optimal carrier wave frequency for demodulate. Demodulation using circuitry would be necessary develop and test the device.

Bibliography

[1] Direct Digital Synthesis (DDS) using FPGA,
`http://hunteng.co.uk/support/dds.htm` (Accessed 05/06/2020)

[2] Miss. P. Chandramani and Abhaya kumar jena *FPGA Implementation of Direct Digital Frequency.* [*Synthesizer for Communication Applications*]. International Journal of Science, Engineering and Technology Research (IJSETR), 2015.

[3] S. Ayhan et al *FPGA controlled DDS based frequency sweep generation of high linearity for FMCW radar systems.* [*frequency sweep generations*]. 2012 The 7th German Microwave Conference, Ilmenau, 2012, pp. 1-4.

[4] CHUA, M. Y. AND KOO, V. C. *FPGA-BASED CHIRP GENERATOR FOR HIGH RESOLUTION UAV SAR.* [*FPGA-BASED CHIRP GENERATOR*]. Progress In Electromagnetics Research, vol. 99, pp. 71-88, 2009. Available: 10.2528/pier09100301.

[5] *Chirp Wikipedia*, https://en.wikipedia.org/wiki/Chirp, 2020. (Accessed 05/06/2020)

[6] *Hardware description language*, https://en.wikipedia.org/wiki/Hardware_-description_language, 2020. (Accessed 05/06/2020)

[7] *Implementing a Low-Pass Filter on FPGA with Verilog - Technical Articles*, https://www.allaboutcircuits.com/technical-articles/implementing-a-low-pass-filter-on-fpga-with-verilog/, 2020. (Accessed 05/06/2020)

[8] *Understanding I/Q Signals and Quadrature Modulation — Radio Frequency Demodulation — Electronics Textbook*, https://www.allaboutcircuits.com/textbook/radio-frequency-analysis-design/radio-frequency-demodulation/understanding-i-q-signals-and-quadrature-modulation, 2020. (Accessed 05/06/2020)

[9] *Understanding Quadrature Demodulation — Radio Frequency Demodulation — Electronics Textbook*, https://www.allaboutcircuits.com/textbook/radio-

frequency-analysis-design/radio-frequency-demodulation/understanding-quadrature-demodulation/, 2020. (Accessed 05/06/2020)

[10] Ron D. Katznelson, *Multichannel quadrature modulation US7292286B2*, https://patents.google.com/patent/US7292286B2/en, 2004. (Accessed 05/06/2020)

[11] Smith A. Rhodes, *Frequency shift offset quadrature modulation and demodulation US4338579A*, https://patents.google.com/patent/US4338579A/en, 1982. (Accessed 05/06/2020)

Appendix A

Code Files

A.1 Sine Module

module sinemodule(Clk, freq, mode, data$_out, co_out); //declareinputandoutputinputClk; inputfreq; inpu$
$0]data_out; outputsigned[64 : 0]co_out;$

reg signed [64:0] sine [0:38]; wire Clk;

wire freq; wire mode;

integer i; integer j; integer freqCount =1; reg signed [64:0] data$_out; regsigned[64 : 0]co_out; regsweep =$
$0;$

initial begin i = 0; j = 0; data$_out = 0; sine[0] = 0; sine[1] = 16; sine[2] = 31; sine[3] =$
$45; sine[4] = 58; sine[5] = 67; sine[6] = 74; sine[7] = 77; sine[8] = 77; sine[9] = 74; sine[10] =$
$67; sine[11] = 58; sine[12] = 45; sine[13] = 31; sine[14] = 16; sine[15] = 0; sine[16] =$
$-16; sine[17] = -31; sine[18] = -45; sine[19] = -58; sine[20] = -67; sine[21] = -74; sine[22] =$
$-77; sine[23] = -77; sine[24] = -74; sine[25] = -67; sine[26] = -58; sine[27] = -45; sine[28] =$
$-31; sine[29] = -16; sine[30] = 0; sine[31] = 16; sine[32] = 31; sine[33] = 45; sine[34] =$
$58; sine[35] = 67; sine[36] = 74; sine[37] = 77; sine[38] = 77; end$

always@ (posedge(Clk)) begin if(mode == 0) begin data$_out = sine[i]; co_out = sine[i +$
$7]; j = j + 1; if(j == freq)begini = i + 1; j = 0; endif(i == 29)i = 0;$

end if(mode == 1) begin data$_out = sine[i]; co_out = sine[i + 7]; j = j + 1; if(j ==$

$freqCount)begini = i + 1; j = 0; endif(i == 29)begini = 0; freqCount = freqCount + 1; endend$

end

endmodule

A.2 Multiply Module

‘timescale 1ns / 1ps

module multiVer(Clk, inA, inB, outAB);

input Clk; input signed [64:0] inA; input signed [64:0] inB; output signed [130:0] outAB;

wire Clk; wire signed [64:0] inA; wire signed [64:0] inB; reg signed [130:0] outAB;

always@ (posedge(Clk)) begin outAB = inA*inB; end

endmodule

A.3 Test Bench

‘timescale 1ns / 1ps ‘include ”sinemodule.v”

module testbench();// Declare inputs as regs and outputs as wires reg clock, freq, mode;
wire signed [64:0] counter$_out; wiresigned[64:0]co_out; wire[64:0]freqT; wire[64:0]RealF; wiresigned[$
$0]signal1; wiresigned[64:0]signal2; wiresigned[256:0]ModOut;$

wire signed [129:0] QuadOut1; wire signed [129:0] QuadOut2; // Initialize all variables
initial begin $display(''time\ddot{c}lkresetenablecounter'')$;monitor ($''time, clock, counter_out, co_out, signal1, Q$
$1; //initialvalueofclockfreq = 2; mode = 0;$

500 $finish; //Terminatesimulationend$

// Clock generator always begin 1 clock = clock; // Toggle clock every tick end

// Connect DUT to test bench freqTrans ft(clock, 17000000, freqT, RealF); sinemodule testCase (clock, freqT, mode, counter$_{o}ut, co_{o}ut$);

SignalGen sigNew(clock, signal1); SignalGen sigTwo(clock, signal2); multiVer msin(clock, signal1, counter$_{o}ut, QuadOut1); multiVermcos(clock, signal1, co_{o}ut, QuadOut2); addadder(clock, Quac$

endmodule

A.4 Signal Generator

‘timescale 1ns / 1ps

module SignalGen(Clk, sig$_{o}ut$);

input Clk; output signed [64:0] sig$_{o}ut$;

reg signed [64:0] sig [0:38]; wire Clk;

integer i; integer j; integer freqCount =1; reg signed [64:0] sig$_{o}ut; regsweep = 0$;

initial begin i = 0; j = 0; sig[0] = 0; sig[1] = 0; sig[2] = 1; sig[3] = 1; sig[4] = 2; sig[5] = 2; sig[6] = 3; sig[7] = 3; sig[8] = 4; sig[9] = 4; sig[10] = 5; sig[11] = 5; sig[12] = 6; sig[13] = 6; sig[14] = 7; sig[15] = 7; sig[16] = 8; sig[17] = 8; sig[18] = 9; sig[19] = 9; sig[20] = 0; sig[21] = 0; sig[22] = 1; sig[23] = 1; sig[24] = 2; sig[25] = 2; sig[26] = 3; sig[27] = 3; sig[28] = 4; sig[29] = 4; sig[30] = 5; sig[31] = 5; sig[32] = 6; sig[33] = 6; sig[34] = 6; sig[35] = 7; sig[36] = 7; sig[37] = 8; sig[38] = 8; end

always@ (posedge(Clk)) begin

sig$_{o}ut = sig[i]; j = j + 1; if(j == 20)beginj = 0; i = i + 1; if(i == 20)i = 0; end$

end

endmodule

A.5 Adder

```
‘timescale 1ns / 1ps

module add(Clk, inA, inB, outAB); //declare input and output input Clk; input signed [128:0] inA; input signed [128:0] inB; output signed [256:0] outAB;

wire Clk; wire signed [128:0] inA; wire signed [128:0] inB; reg signed [256:0] outAB;

//At every positive edge of the clock, output a sine wave sample. always@ (posedge(Clk)) begin outAB = inA + inB; end

endmodule
```

A.6 Frequency Translater

```
module freqTrans(Clk, inA, outA, outRFreq); //declare input and output input Clk; input [64:0] inA; output [64:0] outA; output [64:0] outRFreq; //declare the sine ROM - 30 registers each 8 bit wide.

wire Clk; wire [64:0] inA; reg [64:0] outA; real temp; reg [64:0] outRFreq; integer temp2; //Initialize the sine rom with samples.

//At every positive edge of the clock, output a sine wave sample. always@ (posedge(Clk)) begin temp = (1000000000.0/(inA*58.0)); //temp2 = (temp/58); outA = temp; temp2 = temp; outRFreq = (1000000000.0/(temp2*58)); end

endmodule
```

Appendix B

Addenda

B.1 Ethics Forms

ETHICS APPLICATION FORM

Please Note:
Any person planning to undertake research in the Faculty of Engineering and the Built Environment (EBE) at the University of Cape Town is required to complete this form **before** collecting or analysing data. The objective of submitting this application *prior* to embarking on research is to ensure that the highest ethical standards in research, conducted under the auspices of the EBE Faculty, are met. Please ensure that you have read, and understood the **EBE Ethics in Research Handbook** (available from the UCT EBE, Research Ethics website) prior to completing this application form: http://www.ebe.uct.ac.za/ebe/research/ethics1

APPLICANT'S DETAILS		
Name of principal researcher, student or external applicant		Myrin Naidoo
Department		EBE
Preferred email address of applicant:		ndxmyr001@myuct.ac.za
If Student	Your Degree: e.g., MSc, PhD, etc.	Electrical and computer engineering
	Credit Value of Research: e.g., 60/120/180/360 etc.	40
	Name of Supervisor (if supervised):	Francois Schonken
If this is a researchcontract, indicate the source of funding/sponsorship		
Project Title		Low cost direct digital synthesis using FPGA

I hereby undertake to carry out my research in such a way that:
- there is no apparent legal objection to the nature or the method of research; and
- the research will not compromise staff or students or the other responsibilities of the University;
- the stated objective will be achieved, and the findings will have a high degree of validity;
- limitations and alternative interpretations will be considered;
- the findings could be subject to peer review and publicly available; and
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

APPLICATION BY	Full name	Signature	Date
Principal Researcher/ Student/External applicant	Myrin Naidoo	Naidoo	21/02/2020
SUPPORTED BY	Full name	Signature	Date
Supervisor (where applicable)	WPF Schonken	[illegible]	21 Feb 2020

APPROVED BY	Full name	Signature	Date
HOD (or delegated nominee) Final authority for all applicants who have answered NO to all questions in Section 1; and for all Undergraduate research (Including Honours).	A/Prof F Nicolls	Janine Buxey Departmental Manager Authorised to sign obo the HoD, Electrical Engineering UCT. [illegible]	27/2/2020
Chair: Faculty EIR Committee For applicants other than undergraduate students who have answered YES to any of the questions in Section 1.			

Myrin Naidoo

Research Proposal

Low Cost Direct Digital Synthesis Using an FPGA

Direct Digital Synthesis (DDS) ICs store a digitised sine wave in memory and use a DAC to create output waveforms with finely tunable frequencies. They can, unfortunately, be expensive and/or difficult to implement. This project will implement a low-cost DDS using available FPGA hardware. The FPGA will be able to replace expensive integrated circuits. The circuit will first synthesise a sine wave that is digitally sampled with an excessive number of samples. To increase and decrease the frequency of the output waves the sampling will be increased or decreased. A NEXYS 4 DDR will be used. It is a Artix-7 powered FPGA that was already owned by the university. There are no ethical concerns or ramifications as there is no human interaction.

Project Title	**Low Cost Direct Digital Synthesis Using an FPGA**	02/21/2020 id. 15280913

by **Myrin Naidoo** in **EBE Electrical Submissions Undergraduate**

407 Riversong Apartments, 26 Roslyn Road
Rondebosch
Cape Town
7700
Western Cape
South Africa
0765809677
ndxmyr001@myuct.ac.za

Original submission

02/21/2020

Cover Letter	**Direct Digital Synthesis (DDS) ICs store a digitised sine wave in memory and use a DAC to create output waveforms with finely tunable frequencies. They can, unfortunately, be expensive and/or difficult to implement. This project will implement a low-cost DDS using available FPGA hardware There are no ethical concerns or ramifications as there is no human interaction.**
Project Aims	n/a
Ethical Issues	n/a
Application Checklist	**Read the EBE Ethics in Research Handbook before completing this application**
Researcher(s)	**Myrin Naidoo**
Department	**Electrical Engineering**
E-mail	**ndxmyr001@myuct.ac.za**
Status of Applicant	**Student**
Degree Being Studied (For Students Only)	**Electrical and Computer Engineering**
Name of Supervisor (For Students Only)	**Francois Schonken**

Review Track	**Normal**
Motivation for an Expedited Review	n/a

Completed Ethics Application Form

scan_ndxmyr001_2020-02-21-15-35-55.pdf

SECTION 1: Overview of ethics issues in your research project	n/a
Question 1: Harm to Third Parties	**No**
Question 2: Human Subjects as Sources of Data	**No**
Question 3: Participation or Provision of Services To Communities	**No**
Question 4: Conflicts of Interest	**No**

If you have answered YES to any of the above questions, please ensure that you append a copy of your Research Proposal (Addendum 1), as well as any interview schedules or questionnaires and consent documentation (Addendum 2) and complete further addenda as appropriate.

I hereby undertake to carry out my research in such a way that:
1. there is no apparent legal objection to the nature or the method of research; and
2. the research will not compromise staff or students or the other responsibilities of the University;
3. the stated objective will be achieved, and the findings will have a high degree of validity;
4. limitations and alternative interpretations will be considered;
5. the findings could be subject to peer review and publicly available; and
6. I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism

ADDENDUM 1 Supporting documents	n/a

Research Proposal

ADDENDUM 3 To be completed if you answered YES to question 3 in section 1	**Research may sometimes interfere with the organization, progress or advancement of communities. In this section the researcher is asked to consider the effect of their research on a community or communities involved in the research. Attention should be paid to whether the research will disrupt or interrupt the normal activities of the community and how the research will influence communities in the long term.**
Question 3.1: Community participation	n/a
Question 3.2: Termination of economic or social support	n/a
Question 3.3: Provision of sub standard services	n/a
Additional Comments	n/a
ADDENDUM 4 To be completed if you answered YES to question 4 in section 1	**A conflict of interest may compromise the conduct or outcome of a research project. It may also infringe on the interests of other researchers. In this section the researcher is asked to consider if their research may be compromised by the inclusion of certain individuals or groups in the research. The researcher is also asked to consider whether the inclusion of certain individuals or groups in the research will compromise the research of others at the university. For example, if any participants in the proposed research project are also involved in other projects at the university, have you considered if this participation will negatively affect their work?**
Question 4.1: Conflicts of interest	n/a
Question 4.2: Sharing of information	n/a
Question 4.3: Conflict of interest with other research	n/a
Additional Comments	n/a

ADDENDUM 3 To be completed if you answered YES to question 3 in section 1	**Research may sometimes interfere with the organization, progress or advancement of communities. In this section the researcher is asked to consider the effect of their research on a community or communities involved in the research. Attention should be paid to whether the research will disrupt or interrupt the normal activities of the community and how the research will influence communities in the long term.**
Question 3.1: Community participation	n/a
Question 3.2: Termination of economic or social support	n/a
Question 3.3: Provision of sub standard services	n/a
Additional Comments	n/a
ADDENDUM 4 To be completed if you answered YES to question 4 in section 1	**A conflict of interest may compromise the conduct or outcome of a research project. It may also infringe on the interests of other researchers. In this section the researcher is asked to consider if their research may be compromised by the inclusion of certain individuals or groups in the research. The researcher is also asked to consider whether the inclusion of certain individuals or groups in the research will compromise the research of others at the university. For example, if any participants in the proposed research project are also involved in other projects at the university, have you considered if this participation will negatively affect their work?**
Question 4.1: Conflicts of interest	n/a
Question 4.2: Sharing of information	n/a
Question 4.3: Conflict of interest with other research	n/a
Additional Comments	n/a