

Adjustable Precision Processor

Performance Cost Analysis



Presented by:
Keegan Crankshaw

Prepared for:
Dr. Simon Winberg
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 fulfillment of the academic requirements for a Bachelor of Science degree in
Electrical and Computer Engineering.

October 2018

Key words:
FPGA, Floating Point, Size, Weight, Power, Heterodyning

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another person'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:.....

Keegan Crankshaw

Date:.....

Acknowledgements

I would like to express my gratitude to the following people who's contribution has been invaluable over the course of this project and my degree:

My family, and specifically my parents for their love and support through all times, good and bad. I could not ask for better role-models.

Dr Simon Winberg for his support and guidance through the course of this project. Thank you for always providing quick and useful insights and feedback.

Mikhaeel, Byron, Austin, Azraa and to all my other friends who have helped me maintain my sanity, provided feedback on various aspect on this project, and been pillars of strength in trying times.

The Allan Gray Orbis Foundation for their decision to fund me and afford me this education at UCT.

Abstract

This paper details an investigation that took place over the course of a semester into the trade offs that occur when using reduced precision for processing.

The IEEE754 format defines various word sizes for implementation in floating point calculations, but oftentimes the additional accuracy or range provided by longer word sizes results in extra processing time and higher power consumption.

In an effort to holistically capture the effects of reducing precision for floating point operations, HDL is written for parameterised implementations of floating point operations. These are synthesised to measure the changes in resource use, power consumption, and execution time for a set of operations. The modules are compared to Xilinx IP to measure the differences in the same metrics.

Heterodyning, a use case relating to signal processing, is considered, and is implemented at varying levels of precision. This same application is implemented on a Raspberry Pi at different precisions using different floating point instruction sets.

The paper found that a decrease in bit-width increased speed of execution, decreased power consumption, and decreased logic elements used on the FPGA at the cost of some precision. The heterodyning case was found to run at lower precisions, given that the inputs to the system were constrained within a particular range.

The designs presented in this paper could be further developed and used to implement parameterised floating point units in the form of block IP, implementable on an FPGA for a use case requiring non-standard IEEE-754 floating point implementations.

Terms of Reference

During the course of this project I am to:

1. Develop a framework in an HDL that can be used to test calculations at varying precisions
2. Use that framework to investigate the relationships between speed, size, and power for various bit widths
3. Measure variances in accuracy and precision when converting between a set of word sizes
4. Develop a real-world example that makes use of various bit-widths, and report on the performance and cost analysis
5. Analyse the results using quantitative and comparative methods
6. Draw conclusions from the results about the speed of execution, size of implementation, power consumption and accuracy for varying word-sizes
7. Make recommendations for further investigations

Contents

Declaration	i
Acknowledgements	ii
Abstract	iii
Terms of Reference	iv
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Objectives	1
1.2 Motivations	2
1.3 Background	3
1.4 Scope and Limitations	3
1.5 Plan of Development	4
2 Literature Review	5
2.1 Word Size	5
2.1.1 A Brief History of Word Sizes in Recent Processors	5
2.1.2 The Benefits of Increasing Word Size	6
2.1.3 Costs Associated with Larger Word Sizes	7
2.1.4 Conclusion	7
2.2 Representing Numbers in Hardware	7
2.2.1 Integer Representations	8
2.2.2 Fixed Point	8
2.2.3 Floating Point and the IEEE-754 Representation	9
2.2.4 IEEE 754 Special Representations	11
2.2.5 Additional Considerations for Implementation	12
2.2.6 Conclusion	13
2.3 Size, Weight and Power	14
2.3.1 Overview	14
2.3.2 The Advantages and Disadvantages of Each Metric	14
2.3.3 Measuring SWAP	15
2.3.4 Measuring of SWAP in this investigation	15
2.4 Field Programmable Gate Arrays	16
2.4.1 What is an FPGA?	16

2.4.2	FPGA Applications	16
2.4.3	Physical Structure of an FPGA	16
2.4.4	Performance Measurement on an FPGA	18
2.4.5	Performance Comparison of an FPGA to Other Technologies	18
2.5	Past Efforts at Implementing IEEE-754 on an FPGA	20
2.5.1	Fagin et al. (1994)	20
2.5.2	Shirazi et al. (1995)	21
2.5.3	Louca et al. (1996)	22
2.5.4	Taher et al. (2007)	22
2.5.5	Conclusions	23
2.6	Methods of Optimising Bit-Width in Custom Hardware Designs	23
2.6.1	MiniBit	23
2.6.2	The BitSize Tool	24
2.6.3	Conclusions	24
2.7	Previous Work Using Reduced Precision Implementations	24
2.7.1	Overview	25
2.7.2	Results of the Experiment	25
2.7.3	Conclusion	26
2.8	Conclusion of Literature Review	26
3	Methodology	27
3.1	Phase 1: Initial Research and Literature Review	27
3.2	Phase 2: Overview and Planning	28
3.3	Running of experiments	28
3.3.1	Phase 3: Research and Planning	29
3.3.2	Phase 4: Experimentation and Simulation	29
3.3.3	Phase 5: Data collection and Analysis	30
3.4	Phase 6: Discussion and Conclusion	31
3.5	Phase 7: Notes on Future Work	31
3.6	Conclusion	31
4	High Level Design	32
4.1	Design Overview	32
4.2	Platform and Tool Selections	33
4.2.1	HW01 - Development Environment	33
4.2.2	HW02 - FPGA Platform	34
4.2.3	HW03 - Embedded System	34
4.2.4	SW01 - HDL Development Environment	35
4.2.5	SW02 and SW03	35

4.2.6	SW04 - Embedded System Use Case Development	36
4.3	Design Decisions	36
5	Detailed Design	37
5.1	Mathematical Operators in Hardware	37
5.1.1	Addition	39
5.1.2	Multiplication	40
5.1.3	Division	40
5.2	Design of a Arbitrary Base Converter for IEEE-754 Implementations	41
5.2.1	Requirements of the Python Script	42
5.3	FPGA and Verilog Implementation Testing	42
5.3.1	Resource Measurements	42
5.3.2	Power Measurements	43
5.3.3	Timing Measurements	43
5.3.4	Creation of Test Benches	44
5.3.5	Testing Speed up	44
5.3.6	Comparing Changes in Precision	44
5.4	Comparing Xilinx IP to the Implemented Modules	45
5.5	Comprehensive Test: Heterodyning in MATLAB	45
5.5.1	Heterodyning	46
5.5.2	MATLAB Implementation	46
5.5.3	Design of the Heterodyning Chain	47
5.5.4	The Information Bearing and Carrier Signals	47
5.5.5	The Low Pass Filter	48
5.5.6	Multiplication Blocks	48
5.5.7	Comparisons and Effectiveness of Outputs	49
5.5.8	Advantages and Drawbacks of this Experiment	49
5.6	Speed of Execution Experiment	50
5.6.1	Listing of Tools and Set Up of Development Environment	50
5.6.2	Confirming the Applicability of The Raspberry Pi 3B	51
5.6.3	C++ Code Used	52
5.6.4	Measuring Speed of Execution	53
6	Results	54
6.1	Testing the Veilog Framework	54
6.2	Python Script for Arbitrary Precisions	56
6.2.1	Initial Testing	56
6.2.2	Comparison to Ensure Correctness	56
6.2.3	Accuracy to Decimal Representations	57

6.3	Parameterisation of the Verilog Modules	58
6.4	SWAP at Varying Precisions	59
6.4.1	Resource Use Per Module	60
6.4.2	Effect of DSP Units on Resources Used	61
6.4.3	Power Usage	61
6.4.4	Time Required to Generate Bitstream	63
6.4.5	Timing - WNS	64
6.4.6	Timing - Speed of Execution	65
6.4.7	Accuracy and Precision	66
6.5	Comparison to Xilinx IP	67
6.5.1	Comparison of Results	67
6.5.2	Resource Use	68
6.5.3	Power Usage	70
6.5.4	Timing	71
6.6	Comprehensive Test: Heterodyning in MATLAB	72
6.6.1	Justification of Using MATLAB as a Comparative Test Case	72
6.6.2	Designs and Outputs of the Golden Measure	73
6.6.3	Comparing Outputs Across Bit Widths	76
6.7	Achievable Speed Up of Use Case	79
6.7.1	Speed Up Results	80
7	Discussion and Conclusion	81
7.1	Size of Implemented Design	81
7.2	Time for Implementing Design	81
7.3	Power Use	81
7.4	Precision and Accuracy	81
7.5	Speed of Execution	82
7.6	Comparison of Implemented Designs and Xilinx Developed IP	82
7.7	Implementation on an Embedded System at Reduced Precision	82
7.8	Concluding remarks	82
8	Future Work	85
	References	87
	Appendix	92
A	Link To Resources	92
B	Testbenches	92
B.1	32-bit Precision Test Bench	92

B.1.1	32-bit Waveforms	93
B.1.2	16-bit Waveforms	94
C	Verilog Code	94
C.1	Individual Parameterized Modules	94
C.1.1	Add	94
C.1.2	Multiply	98
C.1.3	Divide	102
C.2	Implementations for SWAP Measurements	107
C.3	Test Benches	108
C.3.1	Coefficient File Used	108
C.3.2	IP Test bench	108
C.3.3	IP Comparison Test bench	110
D	Python Scripts	112
D.1	First Conversion Attempt	112
D.2	AnyFloat Module	113
E	Vivado	116
E.1	Creating a project	116
F	MATLAB Code	117
F.1	Justification Experiment	117
F.2	Heterodyne Experiment	117
G	IP Comparison Results	119
G.1	All Results Recorded	119
H	SWAP of Modules	121
H.1	Power and Timing	121
H.2	Resource Use	123
I	Embedded System Use Case	123
I.1	CPU Info	123
I.2	Batch File for Cross-Compilation and Transfer	124
I.3	The C Code Used	124
I.4	Timing Module	125
J	Ethics Clearance	127
K	ELO Tracking Form	131

List of Figures

1.1	A Gantt chart showing the expected time to be spent on each aspect of the project	4
2.1	A graph showing the word size of commercially available processors over the years	6
2.2	IEEE-754 Half-precision implementation	9
2.3	The stages involved when using IEEE-754 Floating point implementation . .	13
2.4	Showing how guard (G), round (R) and sticky (S) bits are appended to the floating point representation during calculation	13
2.5	An in-line USB-C Power Meter	15
2.6	Simplified FPGA architecture	17
2.7	The architecture of a DSP48E1 slice	18
2.8	Comparison of various technologies in terms of flexibility and efficiency	19
2.9	Comparison of energy efficiency to flexibility of the system	19
2.10	High-level view of the Add/Multiply circuit implemented by Fagin et al. . . .	20
3.1	Phases of the project	27
3.2	The Spiral Model	28
3.3	The V-Model	29
4.1	Break down of experiments	32
5.1	A high-level view for processing IEEE-754 operations	37
5.2	The IO signals in each mathematical operator	39
5.3	The states in Jon Dawson's Addition module	39
5.4	The restoring-division algorithm for 32-bit floating point numbers	41
5.5	An example of the power report	43
5.6	A simple heterodyne receiver	46
5.7	The implementation of heterodyning in this experiment	47
5.8	The Raspberry Pi Connection Set-Up Used	51
5.9	The Programming Flow Used in Experiment 6	51
6.1	Division Waveform for Jon Dawson's 32-bit floating point divider	55
6.2	The process followed in this experiment	57
6.3	Changes in accuracy of representation in hardware for 0.1, 0.8, 12.35 and -0.09	58
6.4	Flip flops required to implement various operations at varying precision . . .	60
6.5	Look up tables required to implement various operations at varying precision	60
6.6	Change in dynamic power for Addition as bit width increases	61
6.7	Change in dynamic power for multiplication as bit width increases	62
6.8	Change in dynamic power for division as bit width increases	62
6.9	Change in dynamic power for the system as bit width increases	63
6.10	The time taken to generate a bitstream for various bit widths	64

6.11	Change in WNS as bit width increases. Higher is better.	65
6.12	Total resources used across implementations	68
6.13	Look up tables used across implementations	68
6.14	Flip flops used across implementations	69
6.15	DSP units used across implementations	69
6.16	Change in dynamic power per signal type for addition	70
6.17	Change in dynamic power per signal type for multiplication	70
6.18	Change in dynamic power per signal type for division	71
6.19	Change in total dynamic power between parameterized and Xilinx IP implementations	71
6.20	The resultant waveforms at IEEE-754 Double Precision (64-bits)	73
6.21	Comparison of original and recovered waveforms	74
6.22	Scaled and shifted recovered waveform compared to original information signal	75
6.23	A zoomed scale-and-shift to highlight the differences between original waveform and recovered waveform	75
6.24	A graph showing the reconstructed information signal when using 8, 16, 32 and 64 bits throughout the process	77
6.25	A graph showing the reconstructed information signal when using 8, 16, 32 and 64 bits throughout the process	78
B.1	Addition Waveform	93
B.2	Multiplication Waveform	93
B.3	Addition Waveform - 16 bit floating point	94
B.4	Multiplication Waveform - 16 bit floating point	94
B.5	Division Waveform - 16 bit floating point	94
E.1	The process for creating a new project in Vivado Design Suite	116

List of Tables

I	Requirements for the Project	2
II	Comparing the three representations for three integers	8
III	Rounding Modes in IEEE754-2008	11
IV	CLB Resources in the Artix-7A100T	17
V	Advantages and Disadvantages of various Computing Platforms	19
VI	The size and speed results of 16 bit FP Units as designed by Shirazi et al.	22
VII	Results of the improvements as suggested by Taher et al.	23
VIII	Relating experiments back to initial project requirements	31
IX	A table showing the major hardware and software requirements	33
X	Comparison of the computing systems available	34
XI	Comparison Between Micro-controllers and Embedded Systems Available	34
XII	Comparison of the software options available	35
XIII	Design Decisions	36
XIV	A table showing the breakdown of parameters for a given word size	38
XV	The signals used in the parameterised implementations	38
XVI	Table showing the maximum values that certain bit-widths can support	48
XVII	Feature sets supported by the Raspberry Pi 3B	52
XVIII	Compiler Flags for the Floating Point Uni	53
XIX	Comparing 32-bit Operations Between Excel, Verilog Modules and an Online Calculator	55
XX	Comparing the output of the anyfloat script to that of an online tool	56
XXI	Hexadecimal representations of numbers at arbitrary precisions	57
XXII	The accuracy of floating point implementations at varying precisions	57
XXIII	Comparing 16-bit Operations Between Verilog Modules and an Online Calculator	59
XXIV	Comparison of Resources when Using DSP in the Multiplier	61
XXV	Change in WNS for varying bit widths	64
XXVI	Clock cycles taken for operations at 16 and 32 bits	65
XXVII	Results produced by executing operations at different precisions resulting in speed-up	66
XXVIII	Speedup available by moving from 32 to 16 bit implementations	66
XXIX	Comparison of precision for calculations involving speedup	66
XXX	The difference between 16 and 32 bit values	67
XXXI	Comparison of results produced by the Vivado IP and the parameterized modules	67
XXXII	The Worst Negative Slack for the Given Implementations	72
XXXIII	Difference in Execution Speeds for the Given Implementations	72

XXXIV	Comparison of fp16 Cast in MATLAB to Results Generated by 16-bit Calculator	73
XXXV	Correlation between double implementation and other bit-widths . . .	76
XXXVI	Table showing differences in peak values of the scaled and shifted recovered signals	78
XXXVII	Table showing differences in peak values of the scaled and shifted recovered signals for double amplitude information-bearing signal . . .	79
XXXVIII	The effect of continually increased amplitude on the output signals . .	79
XXXIX	Types Supported by the Raspberry Pi	79
XL	Time Taken (in ms) for Multiplication Using Various Compiler Options	80
XLI	Speed up obtained for Various Implementations	80
XLII	Summarized Raspberry Pi Results	80
XLIII	The results for comaprison between IP and parameterized implementations	120
XLIV	Timing and Power Metrics for Jon Dawon's Veilog modules at Varying Precision	122
XLV	Physical Resource Use for Jon Dawon's Veilog modules at Varying Precision	123

1 Introduction

In years gone by, general purpose computing has been (arguably) sufficiently powerful to solve most problems adequately. With the rise of big data, the demand for greater processing speeds and throughput, and with pressure on metrics such as power consumption, a shift in processor architecture is needed. This paper investigates the benefits and costs of using reduced and arbitrary precision in floating point calculations, with an investigation into a signal processing use case.

1.1 Objectives

The objective of this study is to investigate how various implementations of IEEE-754 floating point calculations on an FPGA affect speed of execution, size of implementation, power requirements, and accuracy in comparison to standard formats. By doing so, the study aims to investigate the feasibility of using non-standard IEEE-754 based floating point implementations as a means of increasing speed, decreasing size of hardware required, and reducing power consumption.

The requirements of this project, as shown in the Terms of Reference, are as follows:

1. Develop a framework in a hardware descriptor language (HDL) that can be used to test calculations at varying precisions
2. Use that framework to investigate the relationships between speed, size, and power for various bit widths
3. Measure variances in accuracy and precision when converting between a set of word sizes
4. Develop a real-world example that makes use of various bit-widths, and report on the performance and cost analysis
5. Analyse the results using quantitative and comparative methods
6. Draw conclusions from the results about the speed of execution, size of implementation, power consumption and accuracy for varying word-sizes
7. Make recommendations for further investigations

From this, the following requirements are obtained:

Table I: Requirements for the Project

Requirement Number	Description
R01	A suitable development environment
R02	An HDL framework that can perform suitable IEEE754 implementations of basic operations at varying bit widths
R03	A means of measuring the metrics to be reported on
R04	A framework that can be used to implement a use-case scenario

1.2 Motivations

Applications don't all require that computing be done at the same level of accuracy. For some, you could use lower-precision floating-point arithmetic instead of the commonly used IEEE-754 standard.

- David Patterson [1]

Throughout the years, the requirements for processing data have changed drastically. As more data is required to be processed, the way in which it needs to be processed has changed. In the past, the fastest means of execution was preferable. More recently, power-conscious hardware designs are being favoured. No longer is processing speed the bottleneck, rather the memory wall. This shift from conventional wisdoms to "new wisdoms" may bring with it a change in the way hardware is designed. However software and backwards-compatibility has meant that hardware changes have not come as quickly as perhaps thought initially possible. The x86 architecture, discussed briefly in this paper, has been the underlying architecture of most processors since 1985, and has some inherent limitations as a result of backwards-compatibility.

It can be argued that edge computing, particularly with the rise of "big data", will become vital in day-to-day life. Driver-less cars, for example, are expected to produce upwards of four terabytes of data a day. Machine learning is another modern application which requires vast amounts of computing resources. Edge processors with rapid execution speeds will be required to process that data and reduce network congestion. Faster processors are not the only means by which speed of execution can be increased. Increasing parallelism, or reducing the amount of bits required to represent data and hence perform calculations are also means by which the time required to perform calculations can decrease. Both of these methods are implementable and testable on an FPGA.

Adjusting word size in a processor has both advantages and disadvantages. An increase in word size results in an increase in precision, but at the cost of more resources, such as die size, power consumption and time taken to perform the calculation. Given the significance

throughput and efficiency, this paper seeks to conduct a cost-benefit analysis of adjusting word size by using various signal processing algorithms as a case study due to their significance in the real world. This paper will investigate the effect of adjusting word size on execution speed, size of implemented design, power consumption, and the accuracy of the results produced.

1.3 Background

This project is based on research by John Collins on the topic of investigating the numerical precisions required to execute real world programs [2]. Standard 32-bit floating point or fixed point numbers potentially provide more precision than what is needed, meaning more data is being stored and handled than necessary; and the extraneous bit switching that results can cause the system to utilise more power than necessary, as well as possibly taking longer to complete calculations (for example, managing bit carries, transferring data etc.). This project sets out to measure costs of computation for the basic mathematical operators (addition, subtraction, multiplication and division) at varying word sizes.

1.4 Scope and Limitations

This project sets out to achieve the following:

- Implement suitable floating point operations on an FPGA
- Parameterise the operations as to implement them with varying precision
- Compare the following metrics for each implementation:
 - Speed of execution
 - Power requirements
 - Size of implementation
 - Accuracy
- Compare the logic-gate implementations to dedicated hardware implementations (on-chip DSP resources) and industry-developed implementations (Xilinx IP Blocks)
- Create a real-world example as a demonstration that reduced precision floating point implementations have applicability

This project will not investigate:

- The effect of different algorithms on execution speed and SWAP metrics at different word sizes.

- The effect of various FPGAs on SWAP metrics

1.5 Plan of Development

The plan is to first investigate the current landscape of word sizes in hardware, as well as other appropriate literature. Then, implement an HDL framework by which to experiment with varying precision, and generate data from experiments. The operations are to be implemented in Verilog in order to be able to run on an FPGA. Calculations (such as addition, subtraction, division and multiplication) are to be provided to work for varying levels of precision and size (for example, 8 bit, 12 bit, 16 bit, 24 bit and 32 bit floats). The designs will be compiled to see changes in compile (trace and route) times, logic elements used, maximum clock speed, etc. and will form part of the results reported on. A use-case must also be implemented in MATLAB as a means to investigate real-world applicability. Results will be drawn from these experiments, and conclusions made. Finally, recommendations for future experimentation will be made. Figure 1.1 below shows the estimated time to be spent on each aspect of the project.

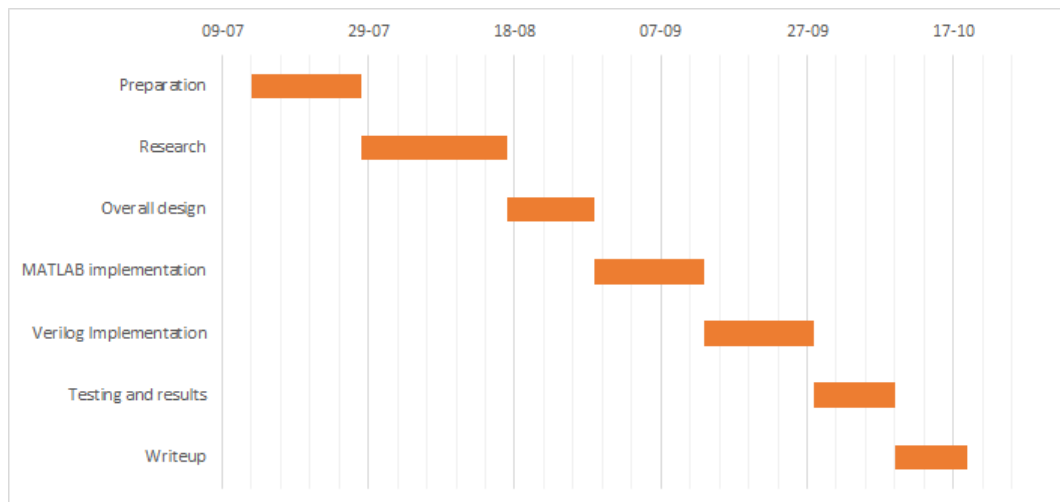


Figure 1.1: A Gantt chart showing the expected time to be spent on each aspect of the project

2 Literature Review

This section covers review of literature pertinent to the project. It begins by examining the history of word sizes in hardware and how it has changed through various architectures through the years. An investigation into how numbers are represented in hardware is done, followed by discussions about SWAP and the FPGA as a platform. Investigations into previous IEEE-754 implementations on FPGAs is done, as well as an investigation into previous methods of bit-width optimizations. The section concludes with a discussion on a previous paper that investigated the relationship between bit width, speed, and resource use.

2.1 Word Size

A "word" in computing can have various meanings, depending on context. Generally, word size refers to the size of the general purpose registers used by the CPU. However, the term can be ambiguous. For example, AMD's Opteron processor, the first 64-bit architecture, developed in 2003 [3], while being a 64-bit architecture, can still be referred to as having a 16-bit word size as it is based on the 16-bit x86 architecture used in the Intel 8086 processor [4]. Microsoft goes on to define other terms for use when developing C-based (C and C++) applications, such as "DWORD" (a 32-bit unsigned integer) and "QWORD" (an unsigned 64-bit integer). These definitions are independent of architecture [5] .

2.1.1 A Brief History of Word Sizes in Recent Processors

The first commercially available processor, the Intel 4004, developed by Intel in the 1970's, had a 4-bit word size [6]. Since then there have been various changes and advances in architecture design, with the development of the x86 architecture in 1978 (for the Intel 8086) being arguably one of the most important developments. This is because the x86 architecture currently dominates the market, particularly the commercial and cloud computing markets [7]. The reason for this is historical - when Intel developed their 16-bit architecture for the 8086, they made it backwards compatible with their 8-bit architecture. This enticed many developers to move over to Intel based chips, as they did not need to rewrite their software to work on new architectures. Since then, the x86 architecture has been extended to 32-bit (by Intel in 1985, now known as IA-32 or i386) [8] and 64-bit (by AMD in 1999, now known as AMD64) architectures [9] . Shown below in figure 2.1 is a chart of first release Intel CPUs with increases in word sizes, starting with the first commercially available Intel 4004 in 1965, up until the release of x86-64 in 2003. A more comprehensive list, including other architectures, can be found on Wikipedia [10].

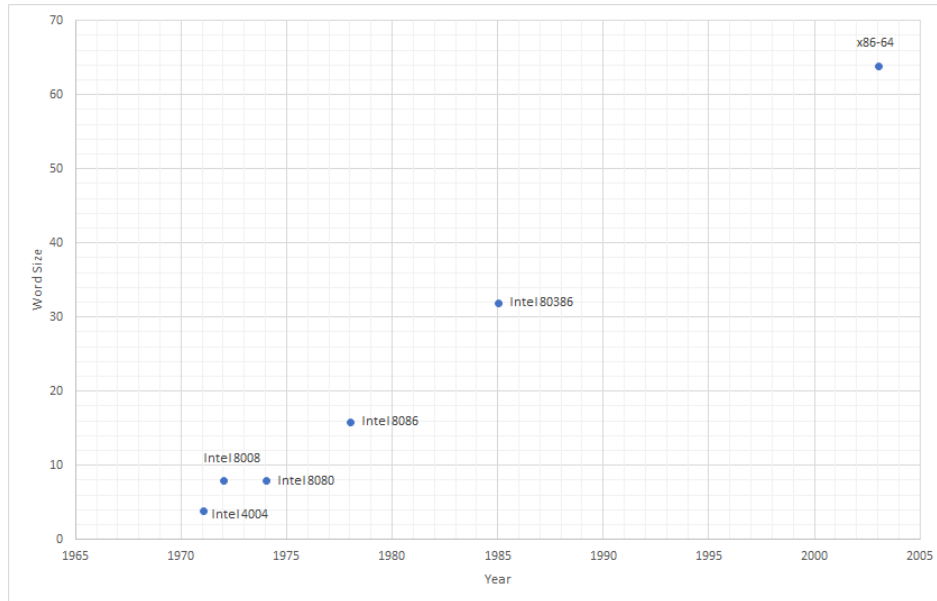


Figure 2.1: A graph showing the word size of commercially available processors over the years [10]

2.1.2 The Benefits of Increasing Word Size

Increasing the size of the general purpose registers in a processor offers a few benefits. The one of obvious significance is that by enabling bigger registers, more memory can be addressed. With an 8-bit word size, 2^8 , or 256 memory addresses can be referenced. On a 64-bit architecture, 2^{64} , or 16 exbibytes of memory, can be addressed. Word size does not always dictate the amount of addressable memory as there are methods, such as Physical Address Extension (PAE), which can be used to allow larger addressable spaces [11].

Another benefit of larger word sizes is the possibility of longer or more complex instructions. Assuming a set number of operands in a given instruction, increasing the word size (and hence instruction size) allows for more bits to be used, which can be used in either the instruction itself (more bits allow for longer instruction codes, which in turn allows for more complex instructions and as a result more complex architectures) [12], in the addresses (which allows for more immediate memory to be addressed), or in the immediate data in the instruction, which allows larger values to be worked with. Longer instruction words opens itself to another type of architecture, known as Very Long Instruction Word (VLIW) architectures, in which a processor can perform multiple operations at the same time in parallel hardware [13]. Another advantage of longer instructions is the ability to handle greater precision representations of data in fewer instructions [14].

Somewhat tied to larger values being worked with in the instruction, larger word sizes also

allow for larger representations of numbers, and, as a result, more precise and accurate results in calculations. This will be covered in more detail in Section 2.2 - Representing Numbers in Hardware.

2.1.3 Costs Associated with Larger Word Sizes

Alongside the advantages offered, an increase in word size naturally has some disadvantages. For each additional bit to be represented, additional hardware is required. A larger instruction size means more memory is required to perform basic instructions. These factors result in increases in the physical size, weight, power consumption and cost of the device. Given the increase in complexity, more time is required to design the hardware. As an example, busses need to be larger, and more considerations need to be taken into account with regards to speed of transfer, as longer distances (caused by the larger chip size due to more hardware) result in longer flight times, and hence higher latencies, as implied by Equation 1 [15].

$$Latency_{Total} = Overhead_{Sending} + Time_{Transmission} + Time_{Flight} + Overhead_{Receiver} \quad (1)$$

It may also be that the application in use does not require long word sizes, and as a result more data may be being handled than necessary, which can result in longer time for computation and unnecessary power usage. Increased power usage, due to increased size, can also be as a result of unnecessary bit switching [16].

2.1.4 Conclusion

In closing, it can be noted that while an increase in word size may offer various benefits, those benefits may not be suited to providing the best performance for high performance computing applications due to the potential disadvantages.

2.2 Representing Numbers in Hardware

As was mentioned in *Section 2.1 - Word Size*, a larger word size can provide an increase in accuracy. This does, however, depend on the way in which the numbers are represented - or stored - in the underlying hardware. This chapter seeks to investigate and compare the various methods, looking at advantages and disadvantages of each method, as well as how the methods compare.

2.2.1 Integer Representations

Integer representation is the simplest type of representation that can be performed in a digital system. In this representation, only integer numbers are used. For example, given 8 bits, and no sign bit, the numbers represented could be from $2^0 = 0$ to $2^8 = 256$.

In all below representations, the most significant bit (MSB) is called the sign bit. This bit has value 0 if the number is positive, 1 if the number being represented is negative.

- Sign-Magnitude representation

For Sign-Magnitude representation, the remaining bits simply hold the magnitude of the number to be represented.

- 1's complement

For 1's complement, positive representations are simply the magnitude, but to represent negative numbers, the magnitude is inverted.

- 2's complement

2's complement follows the same format as 1's complement, but after inversion for negative numbers, a 1 is added. This is the method of representation used by digital systems.

The representations for values of 15, 0, and -15 are shown in Table II below, using an 8 bit representation.

Table II: Comparing the three representations for three integers

Integer Value	Signed-Magnitude	1's Complement	2's Complement
15	0000 1111	0000 1111	0000 1111
0	0000 0000 or 1000 0000	0000 0000 or 1111 1111	0000 0000
-15	1000 1111	1111 0000	1111 0001

Integer mathematics is not often used in high-precision applications, as oftentimes fractional representations will be needed. For this, there are two options: fixed point and floating point.

2.2.2 Fixed Point

In fixed point representation, numbers are stored as integers and then scaled by a predetermined scaling factor that remains constant throughout computation. This representation is considered to be "fixed point" because the number of digits after the decimal point is fixed. Usually, fixed point applications are used where speed is more important than precision. This is especially true in digital signal processing (DSP) applications [17].

Fixed point numbers can be represented using Q notation. In Q notation, Qn.m conveys the number of bits used for integer and fractional components of the number, where n refers to the bits in the integer component, and m refers to the number of bits in the fractional component of the number [18]. For example, given Q7.8, it can be determined that 16 bits are required for representation (1 sign bit, 7 integer, and 8 fractional: N=16, n=7, m=8).

The value of a specific N-bit fixed point number is given by the expression in Equation 2. In this equation, x_i represents bit i of x . The range of a given N-bit fixed point number can be given by Equation 3 [19]:

$$x = \frac{1}{2}^m [-2^{N-1}x_{N-1} + \sum_{i=0}^{N-2} 2^i x_i] \quad (2)$$

$$-2^n \leq x \leq +2^n - \frac{1}{2}^m \quad (3)$$

A fixed point number allows greater accuracy than simple integer representations, however the accuracy is limited by the number of bits in the mantissa. In the Q7.8 example, the precision can be calculated as $2^{-8} = \frac{1}{256} = 3.90625 \times 10^{-3}$. This problem is more apparent where the fixed point number has a lower number of bits in the mantissa, which may be required if large numbers are needed to be represented (which will require more bits in the mantissa). As an example, in an 8 bit, Q5.2 fixed point number, the smallest magnitude that can be represented will be $2^{-2} = 0.25$. The number 8.375, for example, could not be represented or stored accurately. In order to address this issue and provide higher accuracy and range, floating point representations are used.

2.2.3 Floating Point and the IEEE-754 Representation

For higher range precision applications, floating point numbers are used. Floating point number specification is defined by IEEE standard IEEE 754-2008 [20]. A floating point number is represented as shown in Figure 2.2 below.



Figure 2.2: IEEE-754 Half-precision implementation [20]

A floating point number consists of three parts: The sign, a biased exponent, and a significand (or mantissa). Wider exponent fields enable greater range, and wider significand fields enable

greater precision. The formula for determining the value of a given floating point number is shown in Equation 4:

$$Value = -1^{sign} \times 1.significand \times 2^{(exponent-bias)} \quad (4)$$

The standard specifies a few levels of precision:

- Half precision (16 bits) implementation (As shown in figure 2.2)
- Single precision (32 bits), with one sign bit, 8 bits for the exponent, and 23 bits for the significand
- Double precision (64 bits), consisting of 1 sign bit, 11 bits for the exponent, and 52 bits for the significand
- Extended precision - a format that extends a supported basic format with both wider precision and wider range. 128 bits, with 1 for sign, 15 for the biased exponent and 112 bits for the significand
- Extendable precision - a format with precision and range under user control

Normalisation

In addition to the exponent being biased, the significand must be normalised. This means that the digit to the left of the decimal point in the significand must be 1. For example, $0.00101_2 \times 2^4$ normalised would be expressed as $1.01_2 \times 2^2$. Because all numbers must be normalised, there is an implied 1 in the significand, which essentially extends precision by one bit. This bit is not included in the representation, and must be included by hardware when performing arithmetic calculations [21].

Rounding

IEEE754-2008 defines a number of rounding modes to be used when the number of bits required to represent the value being stored is greater than the number of bits available. Table III shows the rounding types and offers a brief explanation.

Table III: Rounding Modes in IEEE754-2008 [22]

Rounding mode	Description
roundTiesToEven	Rounds the result to the nearest representable floating-point number and selects the number with an even least significant digit if a tie occurs
roundTiesToAway	Rounds the result to the nearest representable floating-point number and selects the number with the larger magnitude if a tie occurs (this is a requirement for decimal FP arithmetic, but not for binary FP arithmetic)
roundTowardPositive	Rounds the result toward positive infinity
roundTowardNegative	Rounds the result toward negative infinity
roundTowardZero	Truncates the result

2.2.4 IEEE 754 Special Representations

The IEEE standard also defines several unique definitions. These include denormal numbers, zero, infinity, and NaN (not a number).

Denormal Numbers

Denormal (or subnormal) numbers are used to represent very small numbers. If the exponent consists of all zeroes, the number being represented is considered denormalised, or subnormal. In this case the value of the exponent is not $0 - bias$, rather $0 - bias + 1$. The significand also does not have the implied 1. In the example of half precision, the number 1000000010101011 can be interpreted as follows:

- The sign bit is 1, hence the number is negative
- The exponent is (00000) equal to zero. The number is subnormal, and the exponent is thus equal to $0 - 15 + 1 = -14$
- The significand is hence 0.0010101011
- Thus the value (after rounding) equates to $-1.02 * 10^{-5}$

Zero

Zero is represented with a value of zero in the mantissa, and zero in the significand. The sign bit can still be used, allowing for both positive and negative zero.

Infinity

To represent infinity, the exponent field must consist entirely of ones, and the significand zeroes. The sign bit can still be used, allowing for both positive and negative infinity.

Not-a-Number (NaN)

If the mantissa consists entirely of ones and the significand is not equal to all zeroes, the value stored in the register is considered to be not-a-number. This results from performing operations which produce results which cannot be represented or are undefined, such as dividing a number by zero, or performing arithmetic where both operand are infinity.

2.2.5 Additional Considerations for Implementation

In order to implement mathematical operations, a better understanding of the IEEE754 format is required. This subsection covers stages of calculation as implemented in the Verilog modules used in this investigation.

There are various stages to be stepped through when implementing IEEE-754 compliant operations. The stages, which can be seen in graphically in Figure 2.3 below, occur as follows:

1. Receive the numbers

This is simply a matter of receiving the operands as inputs.

2. Unpack into sign, exponent and mantissa

As discussed, IEEE-754 floating point implementations have three components: the sign, exponent and mantissa. This stage separates the three components so that they can be worked on as required.

3. Consider special cases

Special cases to be considered are when an operand is zero, infinity, or not a number (NaN). These are all specifically defined by the IEEE standard, and can save on execution speed. For example, if division is being completed and the dividend is zero, zero can be returned as a result immediately as opposed to wasting execution time on the mathematical operation.

4. Perform the required operation

There are various ways of implementing operands in hardware. These will be discussed later in this chapter.

5. Pack and transmit the result

This is simply placing a correctly formatted result on an output port. Correct formatting includes normalisation and rounding.

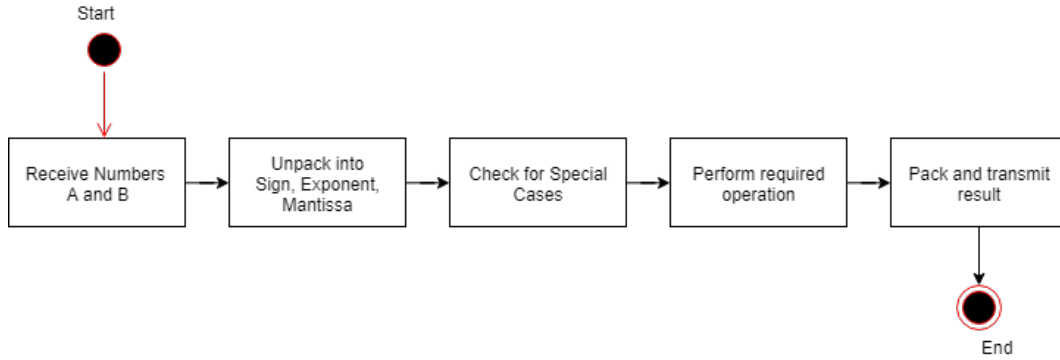


Figure 2.3: The stages involved when using IEEE-754 Floating point implementation

Understanding Guard, Round and Sticky bits

Guard, round and sticky bits are extra bits appended to the mantissa as shown in Figure 2.4. These bits are used to ensure the correct rounding procedure takes place when necessary. Guard and round bits are two extra bits used in calculations for added precision. If the round bit is high, the operation performs the relevant rounding. The guard bit helps prevent overflow. If a value of 1 ever is shifted into the sticky bit position, that sticky bit remains a 1 ("sticks" at 1), despite further shifts [23]. The sticky bit adds as an "or" for all bits that have shifted through it, and can act as an indicator in loss of precision.

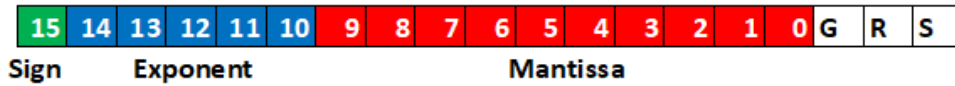


Figure 2.4: Showing how guard (G), round (R) and sticky (S) bits are appended to the floating point representation during calculation

2.2.6 Conclusion

Floating point representations require more resources than fixed point, primarily due to pre- and post-normalisation stages requiring the use of encoders and shifters, which tend to be larger in area, have large combinatorial delays and require more power. As a result, fixed point implementations will usually be less costly in terms of speed, area and power consumption [24].

While fixed point representations may be faster, easier to implement and take up less space on a chip, floating point offers accuracy and range advantages that may be critical to the application at hand.

Despite the disadvantages, floating point implementations are required for greater dynamic range which may be critical to the task at hand. Thus optimisation of floating-point bit-width

is an important task. Optimising word size enables more parallelism, as there are more FPGA resources to be utilised, or, if the degree of parallelisation and instantiations remains constant, there will be less power consumed for smaller word sizes.

2.3 Size, Weight and Power

This section briefly touches on size, weight, and power, and why they are important factors to consider when designing and implementing systems.

2.3.1 Overview

While many trends in computing in the past have been to continually generate more powerful systems capable of higher throughput, in recent years there has been a greater focus on a set of metrics known as SWAP - size, weight and power.

There is no absolute optimal selection between these metrics - which to optimise are entirely dependant on use case. Mobile phone users, for example, will want a small size, low weight and low power consumption device. A computer gamer, however may not necessarily care about any of these metrics, as their primary concern is performance, whereas design of a gaming console may optimise power to prevent overheating and then size to prevent the system being too large (and as a result, unaesthetic). Military and aerospace applications demand high computational throughput devices with low power consumption and small form factors. This is especially true for unmanned vehicles and autonomous systems, which are becoming increasingly prevalent.

2.3.2 The Advantages and Disadvantages of Each Metric

There are no absolute bests when it comes to SWAP. While it could be argued that minimising all three metrics is always advantageous, doing so may impose limitations on the system. Reducing power may have the unwanted effect of reducing clock speed, while increasing clock speed may cause excessive power consumption and, as a result, thermal problems (see AMD Bulldozer and Piledriver CPUs, which had high clock speeds, high thermal dissipation power, but low instructions per clock [25]). Setting size and weight constraints of a system can greatly affect performance. Currently, the world's fastest computer, Summit, weighs an estimated 340 tons. While the average consumer may be impressed with the performance the system offers, it is unlikely they have the 5600 square feet required to house the system. In terms of power, Summit requires 13 megawatts of power [26] [27].

While this example is an extreme, they show that SWAP should be decided on a per-case

basis, rather than attempting to optimise a metric before considering implications in the intended use case.

2.3.3 Measuring SWAP

SWAP can be measured in various ways, depending on the system at hand. It can be estimated from software tools, or measured using physical devices . Size does not only refer to physical size. It may also refer to node size (the size of the transistors used to make a chip) or, as in this investigation, the size of the implemented design on an FPGA - which can be determined by measuring the number of resources an implemented design uses (for example, flip flops and look up tables). Weight refers to the mass of the device, and can be used when recording a power-to-weight ratio. Power can refer to power consumption, or the power of the device in terms of it's execution speed. Common methods of power measurement include using a clamp meter, an in-line USB meter for USB powered devices (see Figure 2.5), or a specialised power measurement circuit, such as one based on the INA169 [28]. For execution speed, wall clock time is a common metric, but the number of clock cycles taken to complete a calculation can also offer insights into the performance of the system.

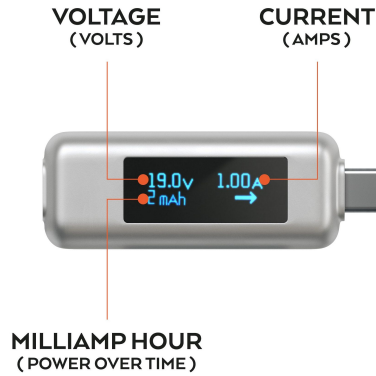


Figure 2.5: An in-line USB-C Power Meter [29]

2.3.4 Measuring of SWAP in this investigation

For this investigation, SWAP constraints will not be set. Rather, designs will be implemented and SWAP metrics measured. The metrics to be measured in this investigation and how the measurements are to be taken is given in Section 5.3.

2.4 Field Programmable Gate Arrays

This subsection provides a brief introduction into what a field programmable gate array (FPGA) is. It begins by investigating what an FPGA is, what it is composed of and who the major manufacturers of FPGAs are. It then delves into where FPGAs are used and how they compare to similar solutions. The subsection concludes by investigating some performance metrics of FPGAs related to SWAP.

2.4.1 What is an FPGA?

A field-programmable gate array is a programmable logic device which can be reconfigured for various functions [30]. Primary manufacturers of FPGAs include Altera and Xilinx.

In order to implement a design, HDL (hardware descriptor language) is used to specify the design. This can be done in a number of languages, the most common of which are VHDL (VHSIC HDL - Very high speed integrated circuit HDL) and Verilog. These descriptors are generated down to bitstreams, which are used to configure the FPGA for the specific task (or tasks) described.

2.4.2 FPGA Applications

FPGAs are much cheaper than application-specific integrated circuits (ASICs) yet can provide similar performance benefits. "A good rule of thumb is that an FPGA implemented in the latest logic family has the potential to provide the same level of performance as an ASIC implemented in the technology of one previous generation" [30]. The added benefits of reuseability and flexibility mean that FPGAs are finding their way into more and more commonplace applications. Microsoft has recently placed FPGAs into their data centers [31]. Amazon Web Services has had FPGAs available in EC2 F1 instances since 2017 [32]. While these data-center based FPGAs are primarily for AI and machine learning applications, FPGAs are well suited to any task that involves parallelism, or where an algorithm can benefit from co-processing or edge execution.

2.4.3 Physical Structure of an FPGA

FPGAs consist of a number of logic elements, the exact amount and structure of which depends on the FPGA vendor and family. The focus in this text will be on Xilinx FPGAs and the structure they use. Specifically, the Xilinx Artix 7 XC7A100T-CSG324 will be used as an example, as this is the target board in this project. The data obtained comes from the *Series 7 FPGA CLB User Guide* [33].

Xilinx FPGAs consist of configurable logic blocks (CLBs). Developed on the 28nm process, the Xilinx Artix 7 XC7A100T has the following CLB resources:

Table IV: CLB Resources in the Artix-7A100T [33]

Slices	SLICEL	SLICEM	6 input LUT	Distributed RAM (Kb)	Shift Register (Kb)	Flip-Flops
15850	11100	4750	63400	1188	594	126800

Each CLB consists of two slices, either a SLICEM and SLICEL, or two SLICEL. Each CLB also contains 16 flip flops and 2 arithmetic and carry chains. If the the CLB contains a SLICEM, the CLB also has 256 bits of distributed RAM and and 128 bits shift registers.

Each slice contains four logic-function generators (or look up tables - LUTs), eight storage elements, wide-function multiplexers, and carry logic. SLICEM also support storing data using distributed ram, and shifting data with 32-bit registers.

The look up table (LUT) is what makes logic functions implementable on an FPGA. In 7-series FPGAs, these are 6-input, 2-output LUTs. Logic functions are converted to binary functions, and stored on the LUTs. When an input is applied, the LUT is used to determine the output.

CLBs are connected via an interconnect. Interconnects connect CLBs to other CLBs, as well as connecting CLBs to input/output blocks (IOBs). The type of interconnect that connects CLBs to other CLBs is known as a programmable switch matrix (PSM).

Figure 2.6 shows the interaction of these components.

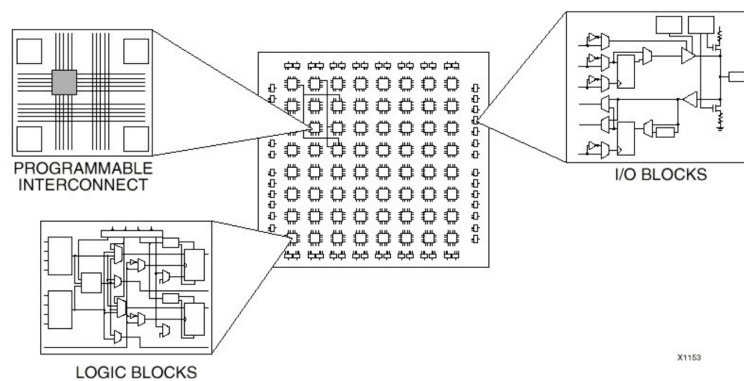


Figure 2.6: Simplified FPGA architecture

The Xilinx Artix 7 also contains DSP slices, specifically DSP48E1. There are 48-bit DSP units with the structure as seen in Figure 2.7. They are able of supporting the following

functionality: multiply, multiply accumulate, multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detection, and wide counter. The architecture also supports cascading multiple DSP48E1 slices to form wide math functions, DSP filters, and complex arithmetic functions. DSP units can be used explicitly or inferred through setting of flags during synthesis [34].

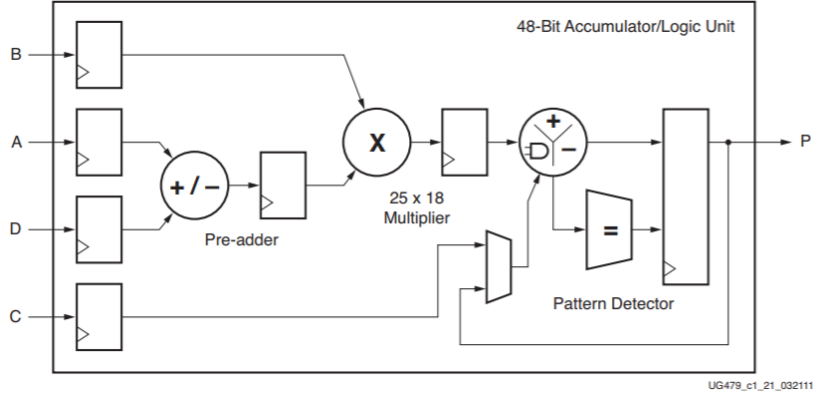


Figure 2.7: The architecture of a DSP48E1 slice [34]

2.4.4 Performance Measurement on an FPGA

Based on the results recorded in previous papers (see Section 2.5) it seems pertinent to record and compare the following:

- Logic elements used
- Maximum clock speed
- Power consumption

Vivado 2081.2, the software package used to develop the HDL for this project, offers various tools for analysis of the implemented hardware design. All the required results can be obtained through synthesis and simulations in Vivado. This process for obtaining this data is described in Section 5.3.

2.4.5 Performance Comparison of an FPGA to Other Technologies

When it comes to implementing solutions to computation problems, a number of platforms exist. These include traditional computing options (Intel and AMD based processors), using GPUs as accelerators, DSP chips, FPGAs and ASICs. All of these chips have various advantages and disadvantages. The advantages and disadvantages can be summarised, as shown in Table V below. Figure 2.8 compares flexibility and efficiency of the various platforms.

Table V: Advantages and Disadvantages of various Computing Platforms [35]

Platform	Software	Re-configurable Computing	Hardware
Advantages	Flexible	Faster than software	High speed
	Adaptable	More flexible than hardware	High performance
	Can be cheaper	Can be more flexible than software	Efficient
		Parallelizable	Parallelizable
Disadvantages	Hardware is static	Expensive	Expensive
	Clock speed limit	Both software and hardware is complex	Static
	Mostly Sequential		

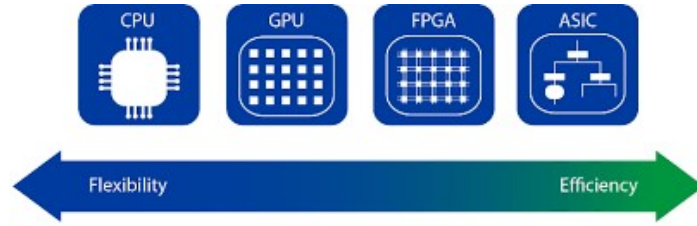


Figure 2.8: Comparison of various technologies in terms of flexibility and efficiency [36]

It can be seen that FPGAs are in the unique position of offering benefits from both ends of the solution spectrum. However this comes at financial and complexity cost.

Figure 2.9 shows the speed of a specific type of hardware in comparison to how flexible the system is. Flexibility, in this context, can be described as how capable a system is in performing different tasks. An ASIC, for example, is inflexible, as it can only perform a specific task. A general-purpose CPU is very flexible, as it can run code to solve various problems. In short, FPGAs and reconfigurable processors provide high flexibility and high MOPS (millions of operations per second) for the given power consumption.

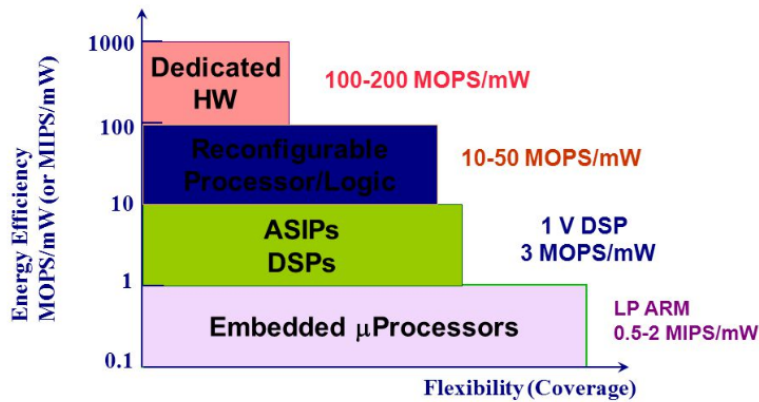


Figure 2.9: Comparison of energy efficiency to flexibility of the system [37]

2.5 Past Efforts at Implementing IEEE-754 on an FPGA

This section covers past attempts at implementing and optimising IEEE-754 implementations on reconfigurable hardware. The papers will be reviewed chronologically, as to show how the improvements in hardware and tools have allowed improvements over time. Finally, the section will end with a brief overview of notable takeaways from these papers.

2.5.1 Fagin et al. (1994)

In 1994, Fagin et al. [38] attempted to implement a 32-bit IEEE-754 standard floating point adder and multiplier on an FPGA. The ultimate goal was to implement an adder and multiplier into a single system with the least amount of area used. The operation (either addition or multiplication) was then selected using a particular op-code. The circuit with inputs and outputs can be seen in Figure 2.10.

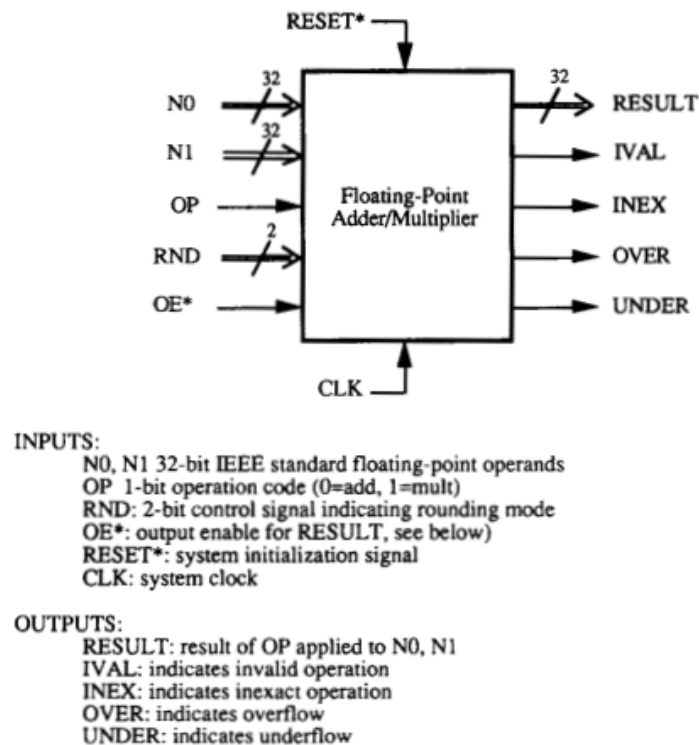


Figure 2.10: High-level view of the Add/Multiply circuit implemented by Fagin et al. [38]

Fagin et al. looked at five different types of adder circuitry: ripple-carry, carry-lookahead, carry-skip, carry select, and "hard macro" adders. Their experimentation showed that the carry select had the best performance for their implementation. For a multiplier, they

considered a shift and add, a carry-save adder, eight carry-save adders, or a combinatorial multiplier. The eight carry-save adder circuitry was selected.

The team also showed that implementing the rounding stage of the IEEE-754 standard in to the design required changes to the exponent and normalisation stages due to the addition of guard, round and sticky bits, and as a result the area required for the design increased. Adding the rounding circuitry also resulted in a performance loss.

The final design was partitioned over 4 Actel A1280 FPGAs with a three stage pipeline and a cycle time of 245ns. Addition had a three cycle latency, whereas multiplication took six cycles. The time taken for multiplication could not be reduced further as the most efficient design could not be synthesised on hardware available at the time, due to the design being too large.

The team showed that FPGAs can be used to empirically determine the cost-benefit analysis of particular design choices, but that an understanding of architectural features is not enough on it's own to determine design trade-offs; a knowledge of the way these systems interact is critical. For example, the performance advantage of the eight carry-select adders was due to more efficient mappings between it and the basic cell structure of the target FPGA. Existence of these architectural nuances complicates possible design decisions. Certain features may require more resources when combined as opposed to when implemented separately, as the team demonstrated with rounding and underflow circuitry.

2.5.2 Shirazi et al. (1995)

In 1995, Nabeel Shirazi et al. [39] did a quantitative analysis on FPGA custom computing machines (CCMs) focusing on area consumption and rate of execution. The motivation was that, even though there are space constraints on (what was then) current day FPGA architectures, scientific operations require fast floating point calculations and the sorts of applications were strong candidates for CCMs due to the high repetition of computations. The nature of the computations also enabled use of custom floating point formats, specifically 18 bits (1 sign, 7 bit exponent and 10 bit mantissa) and 16 bits (known as half-precision, specified by 1 sign bit, a 6 bit exponent, and a 9 bit mantissa). The team implemented addition/subtraction, multiplication and division for a Xilinx 4010 FPGA.

The methodology dictated that the team would rely on the VHDL synthesizer to do the mapping of the components to the FPGA. As the team reported, the constructed circuitry was highly sensitive to the manner in which the original behavioural or structural description is expressed.

In order to increase speed of execution, the team substituted a hard-macro adder/subtractor in place of defined VHDL and unrolled loops into if and case statements. This had the added

benefit of decreasing the size of the design.

Some bottlenecks in the system were identified. In the adder/subtractor (pipelined to produce a result every clock cycle), the bottlenecks were the exponent and mantissa adder/subtractor, as well as the normalisation circuit. In the multiplier (designed to produce a result every three clock cycles), the bottleneck was the integer multiplier. Four different methods were tested to attempt to optimise the multiplier: the integer multiply available through the Synopsis 3.0a VHDL compiler, an array multiplier composing of ten 11-bit save-carry adders, and two methods involving pipelining. The division circuit had an added bottleneck in the recipicator (in this implementation, division was implemented as multiplication by the reciprocal).

Final results of the investigation can be seen in Table VI.

Table VI: The size and speed results of 16 bit FP Units as designed by Shirazi et al. [39]

	Adder/Subtractor	Multiplier	Divider
Function Generators Used (% of total)	26%	36%	38%
Flip Flops	13%	13%	32%
Pipeline Stages	3	3	5
Speed	9.3MHz	6.0MHz	5.9MHz

2.5.3 Louca et al. (1996)

In 1996, Louca et al. [40] ran a study implementing single precision floating point arithmetic on Altera FLEX8000 FPGAs (specifically the Altera FLEX 81188). An adder was implemented using a bit-parallel adder and the multiplier implemented using a digit-serial multiplier. Using the *arithmetic mode* of the FLEX 8000, the total area for the adder was reduced from 72% of the board to 47%. The majority of the area was consumed by the normalisation unit and the shift unit. For multiplication circuitry, 49% of the FLEX81188's area was used. The team noted that the area could be reduced by using the same normalisation unit for multiple multipliers. The team also noted that pre-assigned pins on the board at times corrupted the design, resulting in more area being required when routing. The team managed to achieve 7MFlops for 32 bit addition and 2.3MFlops for 32 bit multiplication.

2.5.4 Taher et al. (2007)

In 2007, M. Taher et al. [22] compared a previous study conducted in 2005 to a new proposed design. The same FPGA (Xilinx Virtex II XC2V6000bf957) was used to compare results. The results are recorded in table VII below.

Table VII: Results of the improvements as suggested by Taher et al. [22]

Circuit	Old design		New Design		Improvements	
	Function Generators	Speed (MHz)	F.G.	Speed	Area reduction	Speed increase
Multiplication	452	20.4	202	89	55%	336.60%
Addition	521	19.4	490	32.6	6%	68%

2.5.5 Conclusions

The following are key takeaways from the four papers discussed above:

- The specific hardware makes a big difference, due to architecture dependent units and designs.
- There is an almost guaranteed improvement in performance over time. This is both to more dense FPGAs being released (due to improvements not only in technology but also node size)
- The exact method of implementing a circuit in HDL can make a considerable difference, depending on the compiler used.

2.6 Methods of Optimising Bit-Width in Custom Hardware Designs

One of the primary goals when designing hardware to optimise area, latency, throughput and power. It is known that smaller bit-widths in FPGA designs can lead to faster execution, and reducing bit-widths in a design means resources can be freed for other aspects of the design. While small designs can be optimised by hand, the work and effort required to optimise larger designs can be difficult and time-consuming. As a result, optimising bit widths is a commonly researched problem. This subsection presents two methods of bit width optimisation and the results of these optimisations.

2.6.1 MiniBit

MiniBit, developed by Lee, et al. [41] performs static analysis using affine arithmetic in order to reduce the number bits required in fixed-point implementations. Static analysis provides conservative bit-width estimates by using descriptors of the signals used in the design. This is more advantageous than dynamic analysis, which, although provides more optimal bit-widths, requires a large set of input stimuli.

MiniBit runs as follows: range analysis is first performed and results passed to the precision analysis phase. Range analysis is then run a second time, as the ranges may have changed with

changes in precision. Range analysis is performed by affine arithmetic and calculates integer bit-width required for each signal in the design. Error and cost functions are also included in the optimisation process. Precision analysis occurs in two phases: Firstly, using the error function generated by MiniBit to find the optimum uniform fraction bit-width, which is then passed to the second stage, where the error and cost functions are used to find the optimum bit-width for the signals using adaptive simulated annealing.

The team was able to reduce the size of their design (implemented on a Xilinx Virtex 4) by 20%, and latency by 12%.

2.6.2 The BitSize Tool

The BitSize tool, developed by Gaffar, et al. [24], is a "Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs". While previous tools mentioned only cater for integer and fixed point optimisation, BitSize also works for floating point implementations. The design works by utilising automatic differentiation.

BitSize is a more advanced tool in comparison to MiniBit, providing details on parameters such as accuracy, dynamic range, area and speed. BitSize can, given a set of constraints, also report on which representation (fixed or floating point) is more suited to the application. It is implemented in C++ as an object library, and supports two front ends: an overloaded C++ object interface, and a Xilinx System Generator interface.

BitSize uses dynamic analysis: it requires that a set of sample data is used as input. The tool performs both fixed and floating point design runs, and allows the user to select the best implementation. Arbitrary word sizes can be used in this tool.

Running a ray-tracing design through BitSize provided up to a 40% decrease in look up table usage.

2.6.3 Conclusions

Bit-width optimisation is a highly researched field, and justifiably so. Optimising bit width has been shown to result in speed increases of up to 12% and decrease resource usage considerably, allowing for more features to be included in the design.

2.7 Previous Work Using Reduced Precision Implementations

This subsection covers a research paper written by Duben, et al. [42]. The paper covers using reduced precision floating point implementations on an FPGA in order to measure the changes in performance in comparison to a general-purpose system.

2.7.1 Overview

In this investigation, a simple model for geophysical fluid dynamics (specifically, the Lorenz 1995 Model) is implemented on a Xilinx Virtex 6 SXT475 FPGA running at 150MHz. An FPGA was chosen as it allows the adjusting of floating point representation to arbitrary precisions, and the ability to trade the accuracy of results with area of implemented design, power consumption, and operating frequency, something not possible on commercial CPUs. The results were used in a comparative study with a two 6-core Intel Xeon X5650 processors running at 2.67 GHz of similar power consumption as a means of a performance-precision trade-off.

The motivations behind this paper were previous studies along similar lines, where it was found that four FPGAs could provide a 330 times speed up over a 6-core x86 CPU.

Implementations of floating point were done in accordance to IEEE-754 floating point format, with the exception that there was no support for denormal numbers. The main disadvantage of the model chosen is that there is no real world data to compare it to. Thus, the paper chose execution of the model on a Xeon system using 64-bit precision as the golden measure.

2.7.2 Results of the Experiment

The results of the experiments conducted were as follows:

- Running at full precision for an initial set-up period and then reducing precision for further calculations resulted in better precision than running at full precision for the duration of the experiment;
- CPU single precision execution was found to be twice as fast as double precision;
- FPGA single precision was 2.8 times faster than CPU single precision, but no claim is made that the CPU was executing at peak performance;
- Speed up of 1.9 over FPGA single precision is possible when reducing precision in the FPGA with hardly any increase in model error; and
- If small error is acceptable in model, performance can be increased to 2.46 times by further reducing precision.

The paper concluded with the following:

- Performance improvement and reduction in power consumption make FPGAs very attractive for modelling.
- Precision can be reduced to make further increases in performance.

- FPGAs are still much more complicated and consumes significantly more time when designing implementations, as opposed to programming CPUs or GPUs for the same task.

2.7.3 Conclusion

The paper shows that FPGAs can be used to greatly improve the speed of execution for a given application. It further shows that reducing precision and using arbitrary floating point can provide even further improvements in execution speed, as long as there is an allowance for introduction of error caused through reduction in precision.

2.8 Conclusion of Literature Review

This section has covered literature relevant to the project at hand. The chapter started with discussion of how processing architectures have changed over the years, followed by representations of numbers in hardware with special mention of the IEEE-754 Floating Point format. Size, weight and power as metrics in design are covered. Field Programmable Gate Arrays are discussed, and some attempts at implementing the IEEE-754 standard are covered. Methods of bit width optimization are discussed. The section ends with an investigation into a previous study that was done along similar lines to this one.

3 Methodology

This chapter outlines the general methodology used to complete this project, from inception to completion. Figure 3.1 shows the overview of the processes described in this chapter.

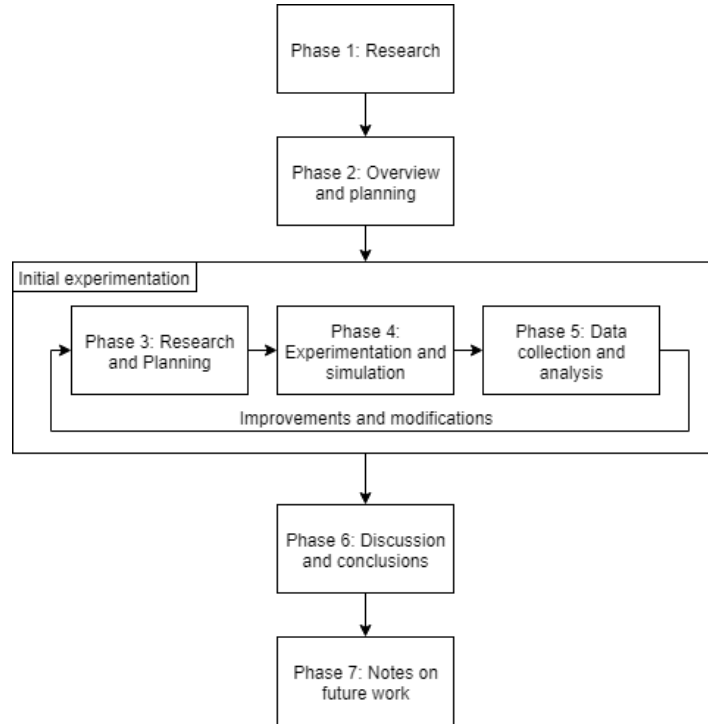


Figure 3.1: Phases of the project

3.1 Phase 1: Initial Research and Literature Review

In order to ensure meaningful experiments were conducted, a review of existing literature was done to see what experiments had been run and what future work may have been required. After recounting the history of word sizes, research was done on the way numbers were stored and represented in hardware. Details on current methods of bit-width optimisation were investigated. Methods of reporting on size, weight and power of systems as well as precision and range of representations was also investigated. Reviews on previous implementations of floating point operations in FPGAs were covered, as well as a case study of a previous similar study into investigating trade off between word size, speed, accuracy, power and size. All of these reviews of literature are available in Section 2.

3.2 Phase 2: Overview and Planning

The experiments were used to determine how the size, weight and power of a device are affected by altering the word size. These experiments are described in Sections 4 and 5. This initial set of experimentation involved running simulations, as well as certain scripts to obtain the necessary data. The scripts were used to determine the effectiveness of different word sizes when representing data (i.e. how well certain word sizes handled accuracy and range).

3.3 Running of experiments

Various experiments were run in order to effectively answer the questions raised in the introductory chapters of this paper. The overall design process follows a Spiral Model, while each feature to be designed and experimented followed the V-Model. The spiral model was initially developed by Barry Boehm in 1986. The model has the advantages of allowing features to be developed systematically, rather than all requirements needing to be defined in the initial stages. Because of the nature of the spiral, features can continuously be added, iterated and improved upon. The spiral aids in mitigating risk by not requiring a full implementation before the verification and validation stages. Feedback is continual and the design can be adjusted at almost any point in the process. The traditional spiral model follows the process as shown in Figure 3.2.

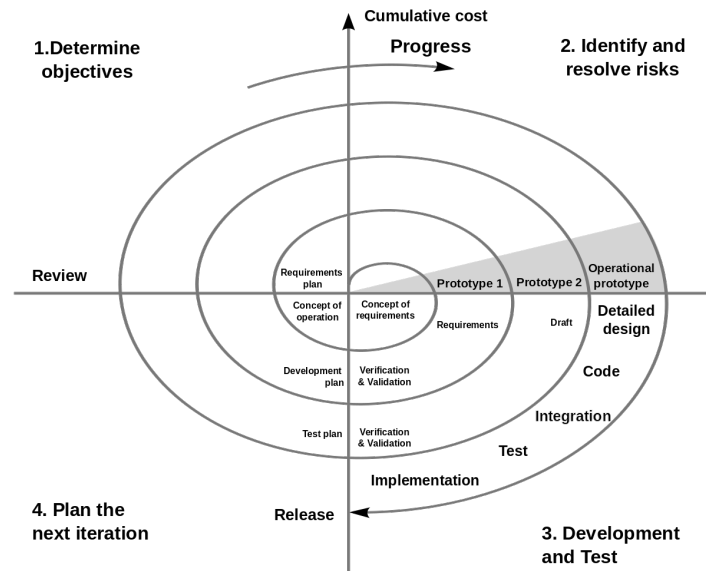


Figure 3.2: The Spiral Model [43]

While the overall project follows the Spiral Model, many of the features and functions within the design will follow the V-Model, shown in Figure 3.3. An example of the components which

follow the V-Model include the Verilog modules to be implemented. While the spiral model is useful for iterative feature additions, it can be expensive (in this project, the primary expense is time). The V-model is useful as it allows for a sequential design flow in places where an iterative processes is not possible. It emphasizes testing and verification of smaller, complete modules before incorporation into a larger system, which assists with risk mitigation.

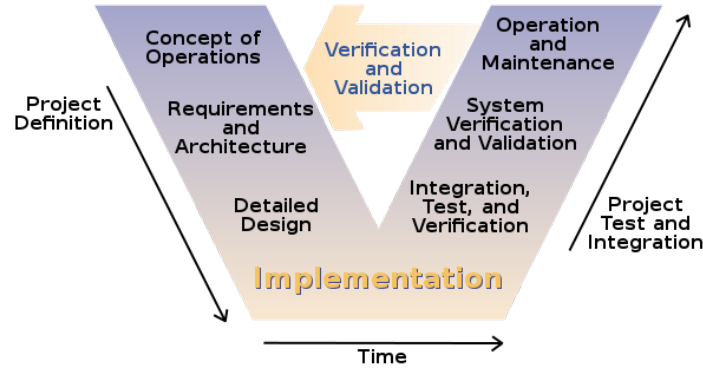


Figure 3.3: The V-Model [44]

3.3.1 Phase 3: Research and Planning

This section starts the detailing of the experimentation phase. In order to conduct meaningful experiments, a strict testing methodology was used. All experimentation was conducted with the intention of obtaining meaningful data. Before experiments were run to gather data, tests were run on the outputs of Verilog modules to ensure correct and expected output was received. Research was done to find the most appropriate tools for the given tasks of power and size estimation. Mathematical operator implementations which could be parameterised with relative ease were found.

3.3.2 Phase 4: Experimentation and Simulation

To effectively test different word sizes and the impact, modules are to be synthesised for varying bit widths. Size, weight and power metrics were recorded. Python scripts were used to ensure the correct results were being produced by the parameterised Verilog modules. If time allows, implementation on another embedded platform was run as a means of comparison between an FPGA as a hardware accelerator and the ease and cost benefits of available embedded platforms. The experiments are constructed to meet the requirements in Table I (Section 1.1) as follows:

Experiment 1 - HDL Maths Modules

HDL modules that use IEEE-754 must be obtained (or, if not found, simple ones developed). These are to be synthesised in order to determine power consumption and fabric requirements. Design of this experiment can be found in Section 5.1. Results can be found in Section 6.1.

Experiment 2 - Develop Script to Check Accuracy and Correctness

A Python script (or other suitable implementation) is to be developed to convert between different word sizes. It should implement IEEE-754 standards. It can also be used to test the precision and range that can be obtained when using a specific bit width. Design of this experiment can be found in Section 5.2, with results in Section 6.2.

Experiment 3 - Parameterisation of Modules

The modules in Experiment 1 should be parameterised for any arbitrary word size. The numbers used as operands should be constant across all word sizes. The operands should be generated using the script developed in Experiment 2. The results of the operations should be verified using the same script. Design of this experiment can be found in Sections 5.1 and 5.3, and results in Section 6.3 and 6.4 .

Experiment 4 - Comparison of Implementations

The parameterised implementation developed in Experiment 3 will be compared against Xilinx's IP implementation of the floating point calculations. Design of this Experiment can be found in 5.4, with results in Section 6.5.

Experiment 5 - Development of Use Case Scenario

A suitable use case should be developed and run at varied precisions to indicate the performance cost break down. The use case should be related to a field where FPGAs are used, such as signal processing. The design of this experiment can be found in Section 5.5, with results in 6.6.

Experiment 6 - Measuring Achievable Speed Up

This experiment is run in order to measure the speed up that can be obtained by running a design at a reduced bit-width. The use case developed in Experiment 5 should be used. Design of this experiment is detailed in Section 5.6 with results available in Section 6.7.

3.3.3 Phase 5: Data collection and Analysis

Through the various experiments, relevant data must be recorded in order to use as comparative study of quantitative data. Results can be found in Section 6.

3.4 Phase 6: Discussion and Conclusion

The results recorded will be discussed and justified. Notes on the analysis will be made. Conclusions on the effectiveness of varying precision and the effects on size, weight and power will be made. The discussion and conclusion can be found in Section 7.

3.5 Phase 7: Notes on Future Work

Parts of research which were under-developed or require future work as identified through the duration of the project will be noted and discussed here. The notes on future work can be Found in Section 8.

3.6 Conclusion

In order to meet the requirements as laid out in Table I (Section 1.1), the experiments mentioned above were formulated. Table VIII below shows the relationships between the requirements of the projects, and the experiments.

Table VIII: Relating experiments back to initial project requirements

Requirement Number	Description	Related Experiment
R01	A suitable development environment	Experiment 0
R02	An HDL Framework that can perform suitable IEEE754 implementations of basic operations at varying bit widths	Experiments 1, 3 and 4
R03	A means of measuring the metrics to be reported on	Experiments 1 through 4
R04	A framework that can be used to implement a use-case scenario	Experiment 5

4 High Level Design

This chapter details the design of the components required to fulfill the outline of the project as laid out in Section 1.4 - Scope and Limitations as per the experiments described in Section 3.3.2. The chapter begins by discussing what is required from the experiments. The options for implementing the requirements are investigated and compared. The exact way in which the experiments will be run is described in Section 5.

4.1 Design Overview

This section will break down the design requirements on a per-experiment basis. Figure 4.1 shows how the core aspects of experiments relate to each other, and how one experiment feeds into another.

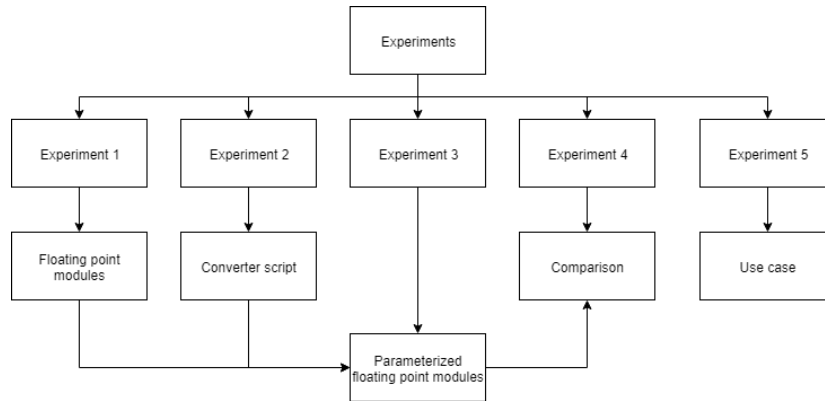


Figure 4.1: Break down of experiments

The requirements of the experiment blocks shown above are as follows:

- Floating point modules
The floating modules should implement the IEEE-754 standard. Careful consideration must be given when choosing implementations as they must be parameterisable (or at the very least have multiple options for the same algorithms implemented at varying levels of precision).
- Conversion script
The script should be able to run for any given level of precision with mantissas and exponents of any specified length.
- Parameterisation and testing
The modules as used in the first experiment should be parameterised. The results of

outputs should be tested to ensure accuracy. The results should also be compared to the expected results.

- Use Case

The use case should be relevant to industry, and able to indicate the importance and significance of choosing word sizes. The use case should be implementable on the FPGA and in a chosen simulation environment. A golden measure should be established as a means of establishing a standard.

The requirements can be broken down into hardware and software requirements. These are shown (in no particular order) in Table IX below.

Table IX: A table showing the major hardware and software requirements

Requirement	Description
HW01	A system on which to develop software
HW02	An FPGA platform on which to run implementations
HW03	An embedded system on which to run the use case
SW01	HDL Development environment
SW02	Scripting language and environment
SW03	Software to establish the golden measure for the use case
SW04	Software to develop the use case for the embedded system

4.2 Platform and Tool Selections

This section iterates over each hardware and software requirement in Table IX and considers some options available to meet the requirements.

4.2.1 HW01 - Development Environment

This requirement relates to the tools used to develop implementations required for the experiment. There are two general-purpose computing platforms available on which to develop software, a lab computer and a personal laptop. The specifications of each system are shown in Table X:

Table X: Comparison of the computing systems available

	Lab Computer	Personal Computer
Operating System	Ubuntu 16.04	Windows 10/Ubuntu 18.04 (Dual boot system)
CPU	Intel Core i5-4690 (Quad core 3.5GHz)	Intel Core i5-7300HQ (Quad core 2.5GHz)
Memory	8 Gb DDR3	8Gb DDR4
Storage Technology	Harddrive	Solid State Drive
Graphics Processor	Nvidia GeForce GTX 465	Nvidia GeForce GTX 1050

4.2.2 HW02 - FPGA Platform

Like Altera vs. Xilinx, the better one is the one you aren't currently using and complaining about!

- u/thecapitalc

While an FPGA is not strictly required for this project, a target board is useful when comparing implementations and power consumption. There are multiple boards available, such as the Spartan 3e, and Nexys 2 and Nexys 3. The board available is the Nexys 4 DDR Evaluation board, containing a Xilinx Artix XC7A100T-CSG324 FPGA.

4.2.3 HW03 - Embedded System

There are a multitude of embedded systems available. Three separate systems will be considered in this evaluation: An Arduino, an STM Evaluation board, and a Raspberry Pi. Comparisons are shown in Table XI below.

Table XI: Comparison Between Micro-controllers and Embedded Systems Available

Board	Arduino Leonardo	STM32-F4 Discovery	Raspberry Pi 3B
Processor	ATMega 32u4	ARM Cortex-M4	ARM Cortex-A53
Memory	32Kb	1Mb	1GB
Word Size	8 bits	32 bits	64 bits
Floating point processing	None	FPU Core - VFP (single precision only)	FPU - VFPv4 (half, single and double precision)
Programming Languages	C/C++ (Python also possible)	C (using Atollic TrueStudio)	Python, C/C++
Power Requirements	5V DC	3V or 5V DC	5V DC

4.2.4 SW01 - HDL Development Environment

For developing HDL for the FPGA, there are two primary choices, VHDL or Verilog.

- VHDL
VHDL (Very High Speed Integrated Circuit Hardware Descriptor Language) is strongly typed, and more verbose than Verilog.
- Verilog
Verilog is weakly typed, and follows a C-style syntax, making the language more familiar to a greater number of people.

While the language used primarily depends on the modules which are found to be suitable, it is worth noting that the advantage of Verilog being syntactically similar to C is considerable. It can somewhat simplify the process of conversion between an embedded system (if C is chosen as a development language), establishment of the golden measure and development of the HDL. The 2016 Wilson Research Group Functional Verification Study also places Verilog as the most popular HDL language for ASIC and IC design [45].

Once the language is chosen, there are various options as a development environment. Xilinx Vivado Design Suite [46] is free to use and offers a lot of information relating to synthesised designs.

4.2.5 SW02 and SW03

Software requirements SW02 and SW03 (scripting language for converting between arbitrary bases and framework to establish a golden measure) can be decided on together, as the options for both requirements are the same. The options for these requirements include MATLAB [47], Octave [48], and Python [49] (with specific note of libraries SciPy [50] and NumPy [51]).

Table XII: Comparison of the software options available

Package	Cost	Additional Features
MATLAB	Licensing	Many packages (Simulink, etc) Well documented
Octave	Free	Some packages Less documentation
Python with libraries	Free	Well Documented

Python (especially with libraries such as NumPy and SciPy) is becoming increasingly popular. Many scripts can be found online, as it is an easily accessible language that many people use for development. MATLAB is considered an important tool in industry, as it is often relied

on by professionals to do modelling and run simulations. It is a very powerful tool, with sufficient documentation (albeit a slightly steeper learning curve than NumPy and SciPy). Octave falls somewhere in-between these two options. It is similar to MATLAB in function, but is not as well optimised or supported. It is also free and open source, as is Python.

4.2.6 SW04 - Embedded System Use Case Development

The programming language chosen largely depends on embedded system chosen. C/C++ are quite popular, so these will be used in the comparison below. IEEE also lists Python as the most popular embedded programming language of 2018 [52], and as a result that will also be added to the comparison.

Python's execution speed is relatively slower compared to that of C, but it is better understood by most people with limited experience in the programming field, and it is easier to write Python. C, beyond its faster execution speed, has a wealth of resources and a significant legacy in embedded systems development. Various compiler options are available to change the way in which the C code is run on the Raspberry Pi.

The primary difference, it seems, is between speed of development or speed of execution.

4.3 Design Decisions

After weighing up the benefits and disadvantages of each option for each design choice, the following decisions (shown in Table XIII) were made:

Table XIII: Design Decisions

Requirement	Description	Choice
HW01	A system on which to develop software	Personal Laptop
HW02	An FPGA platform on which to run implementations	Nexys 4 DDR
HW03	An embedded system on which to run the use case	Raspberry Pi 3B
SW01	HDL Development environment	Verilog Xilinx Vivado HDL Suite
SW02	Scripting language and environment	Python
SW03	Software to establish the golden measure for the use case	Matlab
SW04	Software to develop the use case for the embedded system	C

5 Detailed Design

This section covers the components of experimentation in detail. Each subsection covers one component of design. It starts by investigating the Verilog module used for the mathematic operators, and how these modules were parameterised. It goes on to detail attempts at creation of a Python script for arbitrary base conversion. The section ends by detailing design of the MATLAB golden measure for the use case as well as the HDL design and C-based implementation.

5.1 Mathematical Operators in Hardware

In this subsection, the four basic operators (addition, subtraction, multiplication and division) and the means of implementation will be considered. The operations to be implemented are designed by Github user Dawsonjon, and are available at [53]. The algorithms are "optimised for space" and are not the fastest methods for performing these operations. The Verilog code can be found in Appendix C.

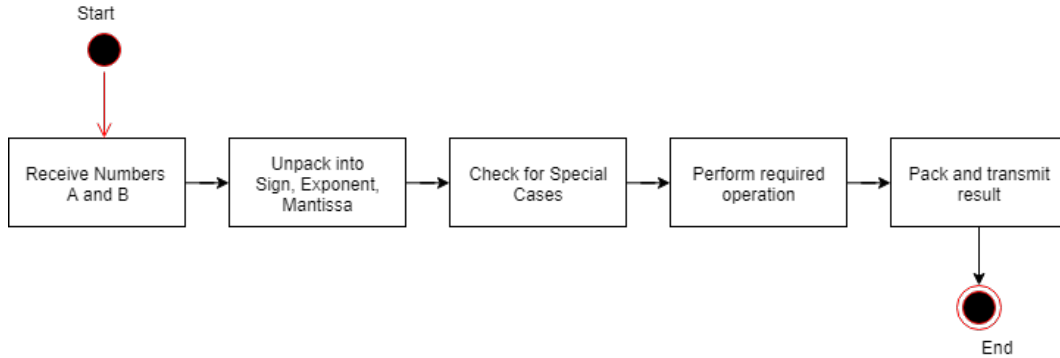


Figure 5.1: A high-level view for processing IEEE-754 operations

In all the basic operations, the process followed is akin to what is shown in Figure 5.1 above. In order to parameterise each module, the "primary parameters" were worked into each module. These are *MANTISSA_MSB*, *EXPONENT_MSB* and *WORD_MSB*. *MANTISSA_MSB* defines the length of the mantissa in bits (starting at zero), *EXPONENT_MSB* defines the length of the exponent in bits (starting at zero), and *WORD_MSB* defines the length of the word in bits (starting at zero). The reason for starting counting at zero, is because that is how registers are defined in Verilog. For example, an 8-bit mantissa register is defined as follows:

```
reg [7:0] exponent;
```

Table XIV shows the values of these parameters for given word sizes.

Table XIV: A table showing the breakdown of parameters for a given word size

Word Size	WORD_MSB	EXPONENT_MSB	MANTISSA_MSB
40	39	7	30
32	31	7	22
24	23	6	16
20	19	5	12
16	15	4	9
8	7	2	3

Additional parameters used within all modules based on word size were added as follows:

```
parameter EXPONENT_BIAS = (2**(EXPONENT_MSB+1))/2-1;
parameter MANTISSA_MAX = 2**(MANTISSA_MSB+2)-1;
```

Each module is implemented as a block diagram, with the following input and output signals shown in Table XV along with a short description of the signal. Figure 5.2 shows the block diagram view of each module. The universality allows the same test bench to be used for each operator.

Table XV: The signals used in the parameterised implementations

Direction	Name	Description
input	input_a	Operand A
input	input_b	Operand B
input	input_a_stb	Indicates Operand A Valid
input	input_b_stb	Indicates Operand B Valid
input	output_z_ack	Acknowledgement of Result
input	clk	Clock signal
input	rst	Reset Signal
output	output_z	Result of Operation
output	output_z_stb	Indicates Result Valid
output	input_a_ack	Acknowledgement of Operand A
output	input_b_ack	Acknowledgement of Operand B

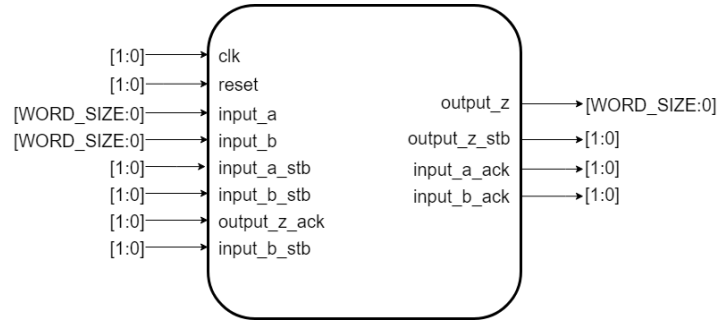


Figure 5.2: The IO signals in each mathematical operator

Each module will now be investigated in depth, and modifications in order to parameterise each module will be detailed.

5.1.1 Addition

Addition was the first module that was to be parameterised. Due to the fact that IEEE754 has a sign bit, the addition module also implements subtraction as addition of a negative. The module is implemented as a finite state machine and goes through the following states: `get_a`, `get_b`, `unpack`, `special_cases`, `align`, `add_0`, `add_1`, `normalise_1`, `normalise_2`, `round`, `pack`, `put_z`. Special cases considered in this module include addition of zero, nan and inf. The basic structure of the FSM is depicted in the flowchart in Figure 5.3. The remaining operators also follow this basic structure. The final Verilog addition module can be found in Appendix C.1.1.

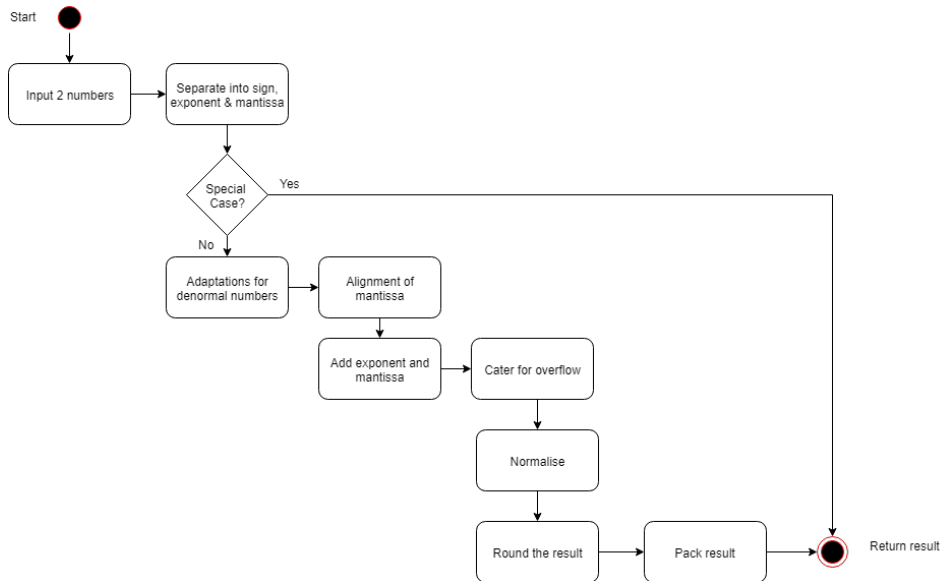


Figure 5.3: The states in Jon Dawson's Addition module [53]

5.1.2 Multiplication

Alongside the previous parameters mentioned, multiplication also required definition of *PRODUCT_MSB*, which was used to define a register of a size large enough to handle the multiplication of two registers of length *MANTISSA_MSB*. The module is implemented as a finite state machine and goes through the following states: *get_a*, *get_b*, *unpack*, *special_cases*, *normalise_a*, *normalise_b*, *multiply_0*, *multiply_1*, *normalise_1*, *normalise_2*, *round*, *pack*, and *put_z*. Special cases considered in this module include multiplication of zero, nan and inf. The final code used can be found in Appendix C.1.2.

5.1.3 Division

The division algorithm was implemented using a restoring division algorithm. The restoring division algorithm for the 32-bit implementation in the original divider can be seen in Figure 5.4 on the following page. In this particular implementation, shift-subtract logic is used, and the same logic is used for each iteration of the algorithm. This means that division takes much longer, but uses less area on the FPGA. For 32-bit division, each division takes roughly 100 clock cycles, but uses 1/50th of the area [54]. Additional parameters defined included *DIV_MSB*, which was used to create a register big enough to shift *MANTISSA_MSB*. This register is also responsible for storing overflow and the guard, round and sticky bits. The final code used can be found in Appendix C.1.3.

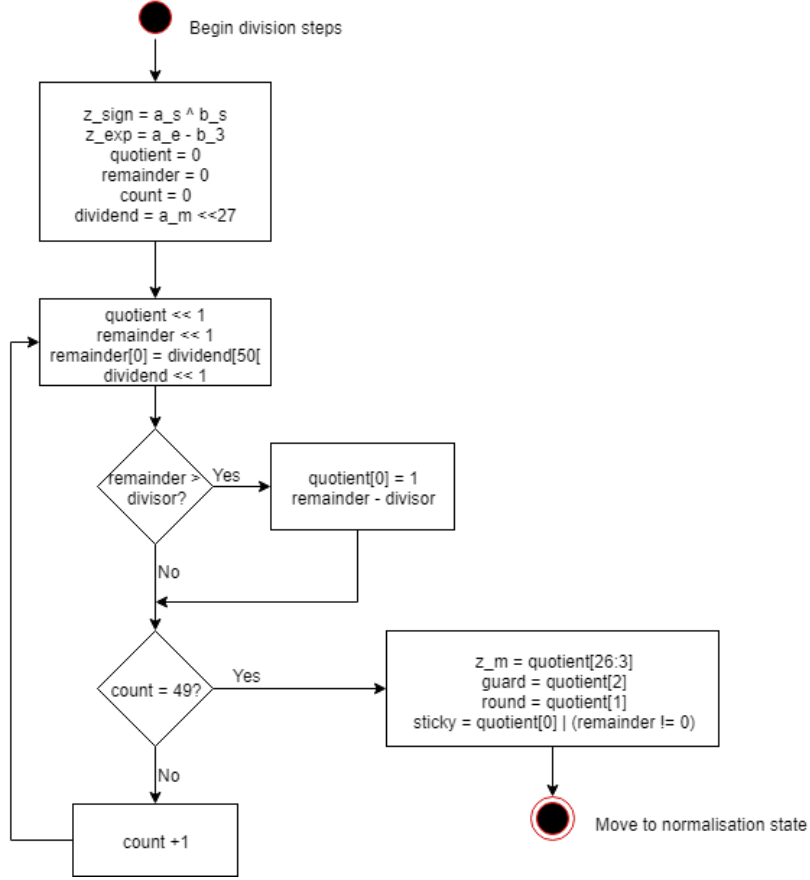


Figure 5.4: The restoring-division algorithm for 32-bit floating point numbers [53]

5.2 Design of a Arbitrary Base Converter for IEEE-754 Implementations

This subsection details the design of creating a Python Script for arbitrary-sized IEEE-754 floating point conversions. The module should be able to convert between a decimal representation and any arbitrary word size. The intention of this module is to use it to convert data generated by the MATLAB golden measure into formats usable by the FPGA, or convert the results produced by the testbench in Vivado into readable decimal formats. This module essentially serves as a sanity check for data in and out of the implementations. Tools exist to verify half, single and double precision online [55], but none for the arbitrary sized implementations used in this project.

5.2.1 Requirements of the Python Script

The python script should be able to do the following:

- Convert between arbitrary IEEE-754 floating point implementations when provided with a specified mantissa and exponent size
- Convert between hexadecimal, binary, and decimal representations

Initial attempts were made to implement a library. Success was initially gained using string handling techniques. The program produced correct results for numbers that did not require rounding, but because rounding was not implemented in the script and instead simply truncated the remaining bits, incorrect results were produced when numbers were required to be rounded. This initial attempt can be seen in Appendix D.1.

A second, in-depth online search yielded the *anyfloat.py* module [56]. Some additions were made in order to streamline the process. The *anyfloat.py* module can be seen in Appendix D.2.

5.3 FPGA and Verilog Implementation Testing

As mentioned in Section 2.3, size, weight, and power constraints can greatly influence design choices. In an effort to incorporate the notion of such limitations on the designs to be implemented, these metrics will be measured and reported on so that the information may be better used going forward.

This section details how various results for the FPGA experiments were obtained and recorded, the manner in which the experiments were done and the primary goal behind obtaining this specific data.

Vivado is a powerful synthesis tool and provides much information. Running synthesis allows estimation of resource use for the given application. Running implementation post synthesis allows for a more accurate report not only on the resources required, but also, if given timing constraints, a report on timing and power use for the given application.

5.3.1 Resource Measurements

Resource measurement is reported in Vivado as soon as synthesis is complete. A more accurate result is produced by running implementation. As mentioned in Section 2.4, core resources on an FPGA include look up tables (LUTs), flip flops (FFs), Block RAM (BRAM), Ultra RAM (URAM), and DSP units. With the exception of URAM, these metrics will be recorded and compared for each of the experiments conducted.

5.3.2 Power Measurements

An example of a high-level power report as provided by Vivado is shown in Figure 5.5 below. The power recorded is shown for the parameterized multiplier running at 16 bits, with inputs supplied by reading BRAM.

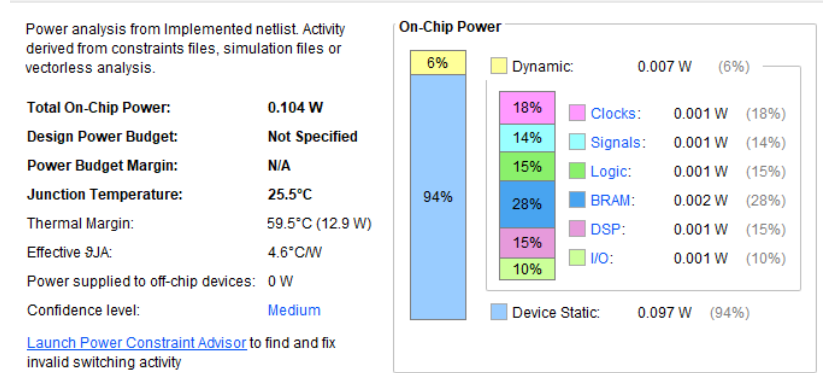


Figure 5.5: An example of the power report

The Nexys 4 DDR's base (static) power consumption is 0.097mW. In order to effectively measure the changes in power consumption, only the dynamic power will be reported on. This will give a better indicator of changes in power per change in word size. Dynamic power will also be broken down into where the power is distributed (for example, signals, clock, or DSP units). Total dynamic power will be reported on, and other measurements will be reported as percentage of dynamic power.

5.3.3 Timing Measurements

All designs will be synthesised to run on the Nexys 4 DDR, which has a 100MHz clock. As a means of comparison of maximum timing, worst negative slack (WNS) will be used. Slack is an indication of the deviation of the time specified by the constraints versus the critical path of a timing signal. Worst negative slack (if positive) means there are no timing violations in the design, and shows how much time is available as "slack" for the worst signal. Essentially, it can be used as a means of measuring the max clock speed achievable for the given implementation on a design. As per [57]:

$$Period_{min} = Period_{actual} - WNS \quad (5)$$

$$F_{max} = \frac{1}{Period_{min}} \quad (6)$$

The actual period of the Nexys 4 DDR given 100MHz clock is 10 nanoseconds. Simplified, the formula for F_{max} becomes:

$$F_{max} = \frac{1}{10^9 - WNS} \quad (7)$$

Because the WNS is the core indicating value of the possible speed increase of a design, this value alone will be recorded and reported on.

Alongside this, the time taken to synthesise the design will also be recorded.

5.3.4 Creation of Test Benches

Test benches will be used as a means of testing all implementations in this project. They are useful, as they can be implemented and developed fairly easily without having to continuously wait for large projects to synthesise, which can be time-consuming. They can be used as a form of debugging and sanity checks before instantiating larger projects. All test benches used will be available in the appendix.

5.3.5 Testing Speed up

It is expected that by moving to smaller word sizes, speed up can be achieved. In order to measure speed up, the following equation is used:

$$Speedup = \frac{T_{old}}{T_{new}} \quad (8)$$

A speed up greater than one implies that the new implementation runs faster than the old one. A speed up of less than one implies that the previous implementation was faster.

5.3.6 Comparing Changes in Precision

For some of the upcoming experiments, it will be necessary to measure change in precision and accuracy, and compare the two results. In order to do this, the following method is proposed:

- Obtain the value using a 64-bit float calculation (for example, in Excel or MATLAB)
- Determine the Euclidean distance between the 32 bit and 64-bit result, and the 16-bit and 64-bit result.
- Define which is more accurate by Equation 9

- A larger (greater than 1) result for 9 implies that the number is more accurate in 16-bit representations, whereas a smaller result (less than 1) implies the number is more accurate in 32-bit representations
- A result of 0 means that the 32-bit representation produces the same result as the 64-bit representation, whereas a result of 2 means that the 16-bit representation produces the same result as the 64-bit representation.
- If 16, 32 and 64 bit representations are all equal, the result is 1

The equation used to define percentage accuracy is as follows:

$$Change\ in\ Accuracy = \frac{Distance_{32}}{Distance_{16}} \quad (9)$$

5.4 Comparing Xilinx IP to the Implemented Modules

This section details the design of Experiment 4, made to compare the Xilinx IP in Vivado to the implemented modules shown in Section 5.1. The point of this experiment is to examine the difference in resource usage, speed of execution, and power consumption between the two.

In order to ensure a fair comparison between the two implementations, 16-bit calculations will be used for both implementations. BRAM will be set up to transfer the values to the module under test, and a finite state machine will be defined in order to test the implementation. From this implementation, a test bench will be developed. All the required metrics (size of implemented design, power consumption, time to compute results) will be recorded.

The test benches used to test the Xilinx IP is available in Appendix C.3.2 and Appendix C.3.3. The coefficient file used for the BRAM is available in Appendix C.3.1.

5.5 Comprehensive Test: Heterodyning in MATLAB

This section outlines how MATLAB was used to develop a golden measure for the heterodyning application. The section begins by defining the system used in this experiment. Each component of the experiment is then explored, and the design explained in detail. The chapter also covers some nuances of the frameworks used. The code used in this experiment can be found in Appendix F.2.

5.5.1 Heterodyning

An analogue heterodyne receiver is shown in Figure 5.6.

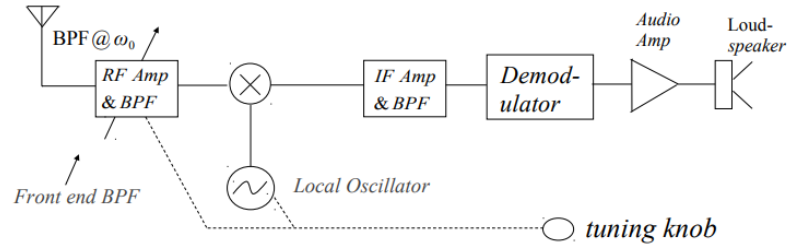


Figure 5.6: A simple heterodyne receiver [58]

In order to simplify the task at hand and investigate only the effect of word size on the given signal, only the multiplication of the signals was considered for a single stage heterodyne receiver and transmitter. Other parts of the heterodyne system were excluded for the sake of efficiency.

5.5.2 MATLAB Implementation

While MATLAB has a library for signal processing known as *Simulink*, this package could not be used as it does not allow custom IEEE754 formats to be used.

The tools available in MATLAB scripting allowed implementations using quarter, half, single and double precisions. This relates to 8, 16, 32 and 64 bits. Quarter and half precision were made available through *Cleve's Laboratory Toolbox* [59], and will be referred to in the text as fp8 and fp16.

The fp8 and fp16 implementations unfortunately do not do any calculations in the defined word size, but rather convert the number from either fp8 or fp16 to double (64 bits), perform the calculation, and then convert the number back into the original precision. This is how the library implements various operators. An example is shown in the listing below where the overloaded multiply operation for fp8 is shown.

```
% Multiply two fp8 numbers
function z = mtimes(x,y)
    z = fp8(double(x) * double(y));
end
```

5.5.3 Design of the Heterodyning Chain

The system (as implemented for this experiment) is shown in Figure 5.7 below. The design of each component will be investigated in detail in the remainder of this section.

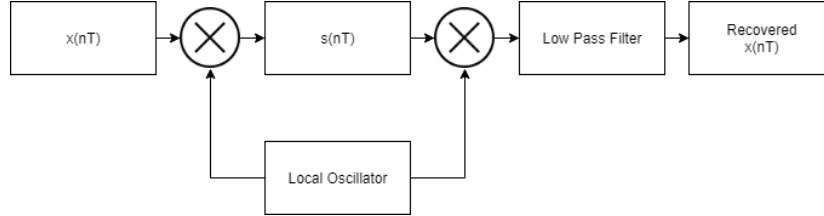


Figure 5.7: The implementation of heterodyning in this experiment

5.5.4 The Information Bearing and Carrier Signals

The frequency bearing signal, $x(t)$ is simply generated using the MATLAB `sin()` function. The samples per second was chosen based on the Linear Technologies LTC1606 [60], which is a 16 bit 250ksps ADC. The parameters are shown in the listing below, with complete code available in Appendix F.2.

```

% Define parameters
Fs = 3.2E6;           % samples per second
StopTime = 0.005;     % seconds
Ac = 2^1;             % Amplitude of carrier
Fc = 10000;           % Frequency of Carrier

Ax = 2^15;            % Amplitude of x(t)
Fx = 2000;            % Frequency of x(t)

dt = 1/Fs;            % seconds per sample
t = (0:dt:StopTime-dt)'; % seconds

%% Generate signals
X = Ax*sin(2*pi*Fx*t);
C = Ac*sin(2*pi*Fc*t);

```

Determining the minimum required precision for the waves

The calculation for the maximum value of any IEEE-754 based floating point implementation can be given as:

$$\text{Maximum Value} = (2 - 2^{-\text{mantissa bits}}) \times 2^{\text{exponent offset}} \quad (10)$$

Using this calculation, it can be determined that the maximum values for the available bit widths is as follows:

Table XVI: Table showing the maximum values that certain bit-widths can support

Bit Width	Bits in mantissa	Exponent offset	Max Value	Bits Required for integer representation
8	4	3	15.5	4
16	10	15	65504	16
32	23	127	3.40282E+38	128
64	52	1023	1.79769E+308	1024

It can thus be seen that so long as none of the waves exceed a value of 15.5, there should be no problem representing them with fp8 precision.

5.5.5 The Low Pass Filter

The low pass filter implemented in this block is known as a moving-average filter. This is a finite-impulse response filter (FIR). Despite it being one of the worst low pass filters, is easy considered to implement, hence the reason for this choice [61].

A moving average filter is defined as follows:

$$y[n] = \frac{1}{L} \sum_{k=0}^{L-1} x[n-k] \quad (11)$$

Where L is the length of the filter, also known as the number of taps, and $x[n]$ is the input signal.

It is defined and implemented in MATLAB as follows:

```
% Pass through LPF
B = 1/NumTaps*ones(NumTaps,1);
out = filter(B,1,InputSignal);
```

5.5.6 Multiplication Blocks

The element-wise operator was not overridden by the fp8 and fp16 libraries. In order to speed up the execution, the vectors were converted to type *double*, element-wise multiplication done, and then converted back to the reduced precision type, as shown in the listing below. This has the same as the overloaded multiplication operator shown above. The listing below shows how the signals were created.

```

% Generate x(t)
X = Ax*sin(2*pi*Fx*t);
X8 = fp8(Ax*sin(2*pi*Fx*t));
X16 = fp16(Ax*sin(2*pi*Fx*t));
X32 = single(Ax*sin(2*pi*Fx*t));

%% Generate L0
C = Ac*sin(2*pi*Fc*t);
C8 = fp8(Ac*sin(2*pi*Fc*t));
C16 = fp16(Ac*sin(2*pi*Fc*t));
C32 = single(Ac*sin(2*pi*Fc*t));

%% Multiply with L0
Y = double(X) .* C;
Y8 = fp8(double(X8) .* double(C8));
Y16 = fp8(double(X16) .* double(C16));
Y32 = X32 .* C32;

```

5.5.7 Comparisons and Effectiveness of Outputs

In order to effectively determine what the effect of varying word sizes has on the modulated signal and output, the following steps were taken:

1. Run a full implementation of the system
This is to get a baseline, or "golden measure" of the system for a means of comparison.
2. Run simulations
This stage simply involves gathering the data. Four experiments will be run. One experiment for each bit-width, where the bit-width remains constant throughout the chain.
3. Other assumptions:
 - The receiver will have no limitations on word size, as it is assumed to be an SDR-type receiver with a 64-bit processor.
 - The LPF will have no limitations, as once the signal reaches the low-pass filter, it is assumed to have access to a 64 bit processor. As a result, all signals are converted to type double before being filtered.

As a means of comparison, correlation to the full-precision output versus each implementation will be measured and recorded.

5.5.8 Advantages and Drawbacks of this Experiment

This experiment is not holistically representative of using reduced precision operations in a signal processing application. The following issues are raised:

- Calculations are not done in limited precision.
This is problematic as it may result in incorrect rounding when converting between

the limited precision and a different result from what may have been produced if the calculation were done in limited precision.

- The full chain of a heterodyne transmitter and receiver are not simulated
While not entirely problematic, the experiment does not run with an accurate representation of a heterodyne chain. The system misses out on key steps, such as sampling of a real-time audio signal and rate determination.
- The benefits of using reduced precision cannot fully be measured
This experiment aims to prove that lower precisions can be considered good enough for certain real-world applications, but does not provide significant information on key metrics, such as speed up.

Despite the drawbacks, this experiment offers the following advantages and insights:

- A rapid prototype showing the effectiveness of reduced precision
This experiment aims to demonstrate that reduced precision calculations in floating point are meaningful, and that further investigation is warranted.
- A key framework which can easily be expanded on
The additional heterodyning stages can be added to this experiment in the future as a means of better investigation.
- Proof of concept
The experiment aims to show that reduced precision calculations can be applicable in real-world applications.

5.6 Speed of Execution Experiment

This section details the design of the implementation to be run on the Raspberry Pi. The same heterodyning procedure run in Experiment 5 (detailed in Section 5.5) will be used, with the exception that the transmitted signal will not be multiplied a second time, as the goal of this experiment is not comparing output, rather the change in wall clock time for varying precision widths using different compiler flags.

5.6.1 Listing of Tools and Set Up of Development Environment

In order to develop on the Raspberry Pi, cross compilation using the Windows Toolchain for Raspberry Pi (raspberry-gcc6.3.0-r3), available at [62]. C++ code was written in a simple text editor, compiled on a laptop computer, and then transferred to the Raspberry Pi via a local Wi-Fi network using pscp. The flow is shown in Figures 5.8 and 5.9 below.

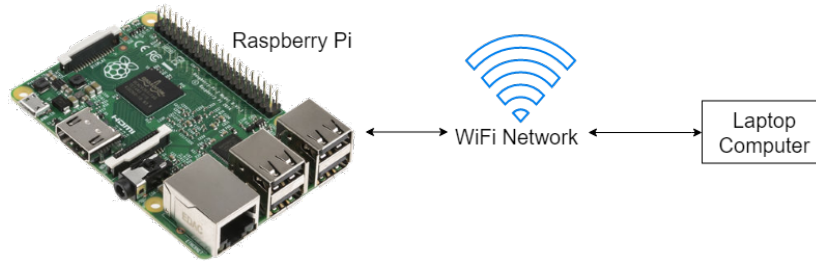


Figure 5.8: The Raspberry Pi Connection Set-Up Used

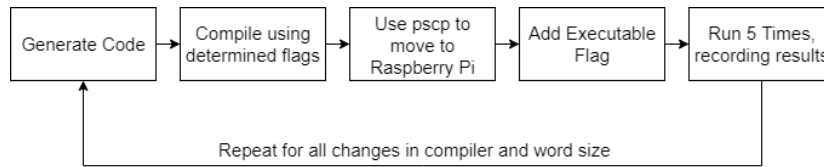


Figure 5.9: The Programming Flow Used in Experiment 6

5.6.2 Confirming the Applicability of The Raspberry Pi 3B

The implementation uses C++, as it allows for specific hardware floating point implementations to be used. Running `$cat /proc/cpuinfo` on the Raspberry Pi produces the results shown in Appendix I.1. From this output, we can see the feature sets the Raspberry Pi supports. The feature sets supported (along with a short description) is given in Table XVII on the following page.

Table XVII: Feature sets supported by the Raspberry Pi 3B

mfpu Flag	Description
half	Half word (32 bit) load and store
thumb	16 bit thumb instruction set
fastmult	323264-bit multiplication
vfp	floating point instructions
edsp	DSP instructions
neon	Advanced SIMD instruction set for media and signal processing applications
vfpv3	VFP version 3
tls	Thread-local storage register
vfpv4	VFP version 4
idiva	SDIV and UDIV hardware division in ARM mode
idivt	SDIV and UDIV hardware division in Thumb mode
vfpd32	VFP with 32-bit decimal registers
lpae	Large Physical Address Extension (>4GB physical memory on 32-bit architecture)
evtstrm	kernel event stream using generic architected timer
crc32	hardware-accelerated CRC-32

The output of `$ dpkg --print-architecture` produces "armhf". This means that the system uses hardware based floating point instructions alongside a 32-bit instruction set. From these results we can confirm that the Raspberry Pi has hardware support for floating point numbers. The Pi is running Raspbian Stretch (version 9).

5.6.3 C++ Code Used

The program used is available in Appendix I.3. The C++ code was simply a for-loop, iterating over data and multiplying the values together. The values were the C and X signals from Experiment 5 - The Carrier and Data signals, respectively. They were obtained by exporting the values from the MATLAB environment. The variables were instantiated as global arrays defined in a separate file. These values were cast to whatever bit-width was being tested, as follows:

- `_fp16` for 16-bit floating point values
- `float` for 32-bit floating point implementations
- `double` for 64-bit floating point implementations

In order to enable half-word (16-bit) floating point calculations on the Raspberry Pi, the compiler flag `-mfp16-format=ieee` is required. The final batch file used to run cross compilation and transfer the file to the Raspberry Pi can be seen in Appendix I.2.

The file was compiled for 16, 32 and 64 bits, all using the following FPU implementations described in Table XVIII

Table XVIII: Compiler Flags for the Floating Point Unit [63]

mfpv Flag	Description
none specified	Default implementation
vfpv3	Version 3 of the floating point unit
vfpv3-fp16	Equivalent to VFPv3 but adds hfp16 support
fpv4	Version 4 of the floating point unit
neon-fp-armv8	Advanced SIMD with Floating point
neon-fp16	Advanced SIMD with support for half-precision
vfpv3xd	Single Precision floating point
vfpv3xd-fp16	Single precision floating point, plus support for fp16

As shown in Figure 5.9, once the file had been transferred across, the following procedure took place:

1. Apply execution permission by running `$chmod +x <file>`
2. The file was then run five times.
3. The times recorded were averaged to obtain a result

The reason for averaging five results is that it provides a better view of the execution time on the system. Run only once, there may be disadvantages with incorrectly configured cache, or a particular background task may be running and cause the task to run slower. By running all implementations five times and taking the average, a more accurate result for all implementations is assured.

With five runs for each implementation, running three different word sizes for eight different floating point methods, a total of 120 runs were completed. The results are recorded and discussed in Section 6.7.

5.6.4 Measuring Speed of Execution

There are various means by which timing could be measured on a CPU. A measure of simply subtracting system time at the start from system time at the end of execution, for example. However, a library that provides high precision (down to 1ns on the Raspberry Pi) was used. It is available in Appendix I.4. Formula 8 is used to calculate the speed up between implementations.

6 Results

This section details the results for each of the experiments done as outlined in Section 3.3.2. Results are not necessarily presented on a per-experiment basis, rather they are collated into sets of useful results with reference back to the experiments from which they were obtained.

6.1 Testing the Veilog Framework

In order to determine whether or not Jon Dawson's [53] floating point operations were accurate, they were tested prior to parameterisation, and the results compare to an online tool [55]. The operations, their result, and the expected result are all documented in table XIX below. The implementation by Jon Dawson is 32-bits, so this is what is used. A task was written to test each function. The task for division is shown below. All tasks followed the same format, as the communications protocol used for each of the mathematical operator implementations is the same. The full testbench is available in Appendix B.1. The first part of the division waveform is shown in Figure 6.1. The remaining waveforms for this test bench can be found in Appendix B.1.1. Table XIX shows the results produced by each of the methods. The results produced by the modules and the online calculator are within rounding accuracy. The biggest outlier in results was the third operation in multiplication, where, instead of producing zero, the module produced $5.87 * 10^{-9}$. This was then corrected in the module, and zero was returned.

```
// Task to complete the operation
task domaths;
  input [WORD_MSB:0] valA;
  input [WORD_MSB:0] valB;
  begin
    A_STB = 0;
    B_STB = 0;
    A_INPUT = valA;
    B_INPUT = valB;
    A_STB = 1;
    B_STB = 1;
    wait(A_ACK == 1);
    wait(B_ACK == 1);
    Z_ACK = 0;
    wait(Z_STB == 1);
    Z_ACK = 1;
  end
endtask
```

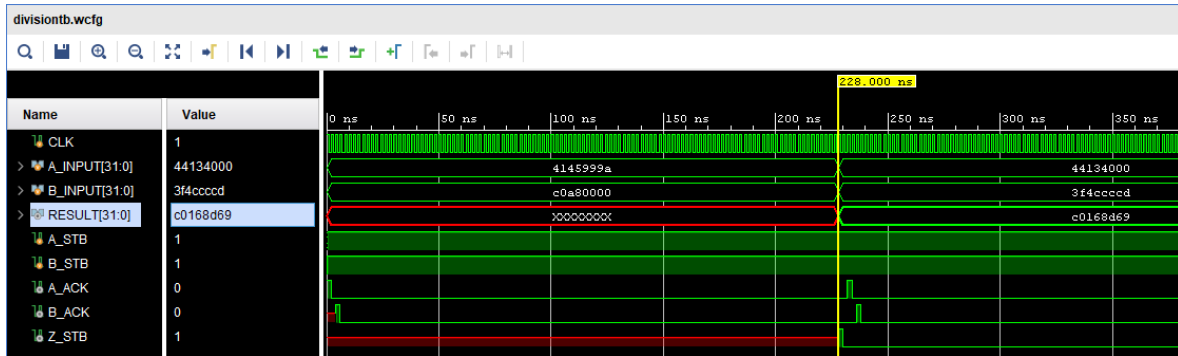


Figure 6.1: Division Waveform for Jon Dawson's 32-bit floating point divider

Table XIX: Comparing 32-bit Operations Between Excel, Verilog Modules and an Online Calculator

Values		Addition			
A	B	Excel	DawsonJon	Online	Cycles
12.35	-5.25	7.1	7.100000381	7.1000004	23
589	0.8	589.8	589.799987792969	589.8	40
0	8950	8950	8950.0	8950	4
82	-0.09	81.91	81.9100036621094	81.91	21
NaN	0.1	-	NaN	NaN	4
12.35	inf	-	Inf	Inf	4
Multiplication					
A	B	Excel	DawsonJon	Online	Cycles
12.35	-5.25	-64.8375	-64.83750153	-64.8375	12
589	0.8	471.2	471.200012207031	471.2	13
0	8950	0	0	0	4
82	-0.09	-7.38	-7.38000011444092	-7.38	23
NaN	0.1	-	NaN	NaN	4
12.35	inf	-	inf	Inf	4
Division					
A	B	Excel	DawsonJon	Online	Cycles
12.35	-5.25	-2.352380952	-2.3528099098206	-2.352381	112
589	0.8	736.25	736.25	736.25	113
0	8950	0	0	0	4
82	-0.09	-911.1111111	-911.111083984375	-911.1111	113
NaN	0.1	-	NaN	NaN	3
12.35	inf	-	0	NaN	3

6.2 Python Script for Arbitrary Precisions

This section details the design of the Python script used for converting between various bases. This script was used for testing the output of the parameterized mathematics modules to compare results between word sizes.

6.2.1 Initial Testing

An initial test was conducted, comparing the Python *anyFloat* library to the results of a converter found online. The calculator can be found at [55]. The test was done to ensure accuracy at IEEE-754 defined precisions, namely 16 and 32 bits. The comparisons between the Python script and the tool are shown in Table XX. The tool was found to produce the correct results.

Table XX: Comparing the output of the anyfloat script to that of an online tool

Decimal Representation	16 bits		32 bits	
	Python	Online Tool	Python	Online Tool
12.35	4a2d	4a2d	4145999a	4145999a
-5.25	c540	c540	c0a80000	c0a80000
589	609a	609a	44134000	44134000
0.8	3a66	3a66	3f4ccccd	3f4ccccd
0	0	0		0
8950	705f	705f	460bd800	460bd800
82	5520	5520	42a40000	42a40000
-0.09	adc3	adc3	bdb851ec	bdb851ec
NaN	7c00	FFFF	7f800000	ffffff
0.1	2e65	2e66	3dcccccd	3dcccccd
inf	7C00	7C00	7f800000	7f800000

6.2.2 Comparison to Ensure Correctness

Based on the previous experiment, it was determined that the Python script was producing the correct results. The next part of this experiment was to produce values for arbitrary precisions required by the upcoming experiments. This involved producing values at 8, 20, 24 and 40 bits. These results can be seen in Table XXI.

Table XXI: Hexadecimal representations of numbers at arbitrary precisions

Decimal representation	8 bits	20 bits	24 bits	40 bits
12.35	69	4145999a	428b33	414599999a
-5.25	d5	c0a80000	c15000	c0a8000000
589	70	44134000	482680	4413400000
0.8	2a	3f4ccccd	3e999a	3f4ccccccd
0	0	0	0	0
8950	70	460bd800	4c17b0	460bd80000
82	70	42a40000	454800	42a4000000
-0.09	86	bdb851ec	bb70a4	bdb851eb85
NaN	70	7f800000	7f0000	7f80000000
0.1	6	3dcccccd	3b999a	3dccccccd
inf	70	7f800000	7f0000	7f80000000

6.2.3 Accuracy to Decimal Representations

IEEE-754 cannot always be entirely accurate to the decimal representation. This is due to the nature of how fractions are represented in binary. The table below shows the accuracy to the decimal representation for each bit size. These values were obtained by applying Equation 4 to the binary result produced by the *anyfloat* library. The process for this experiment is shown in Figure 6.2. Results are recorded in Table XXII.

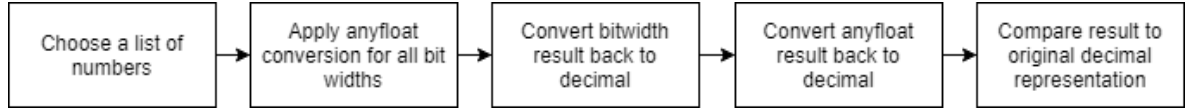


Figure 6.2: The process followed in this experiment

Table XXII: The accuracy of floating point implementations at varying precisions

Decimal	8 bits	16 bits	20 bits	24 bits	32 bits	40 bits
12.35	12.5	12.3515625	12.349609375	12.3499755859	12.3500003815	12.3500000015
-5.25	-5.25	-5.25	-5.25	-5.25	-5.25	-5.25
589	46	589	589	589	589	589
0.8	0.8125	0.7998046875	0.799987792969	0.800003051758	0.800000011921	0.800000000047
0	0		4.65661287308e-10	1.08420217249e-19	5.87747175411e-39	5.87747175411e-39
8950	16	8952	8950	8950	8950	8950
82	16	82	82	82	82	82
-0.09	-0.171875	-0.09002685547	-0.0899963378906	-0.0900001525879	-0.0900000035763	-0.0899999999965
NaN	16	65536	4294967296.0	1.84467440737e+19	3.40282366921e+38	3.40282366921e+38
0.1	0.171875	0.09997558594	0.0999984741211	0.10000038147	0.10000000149	0.100000000006
inf	16	65536	4294967296.0	1.84467440737e+19	3.40282366921e+38	3.40282366921e+38

As is seen in Table XXII above, certain numbers are not well represented in hardware. Figure 6.3 shows how these values become more accurate as bit width increases for various representations. 8 bit representations were intentionally excluded in order to magnify the subtle differences as bit width increases. It can be seen that greater bit-widths allow for more accurate representations.

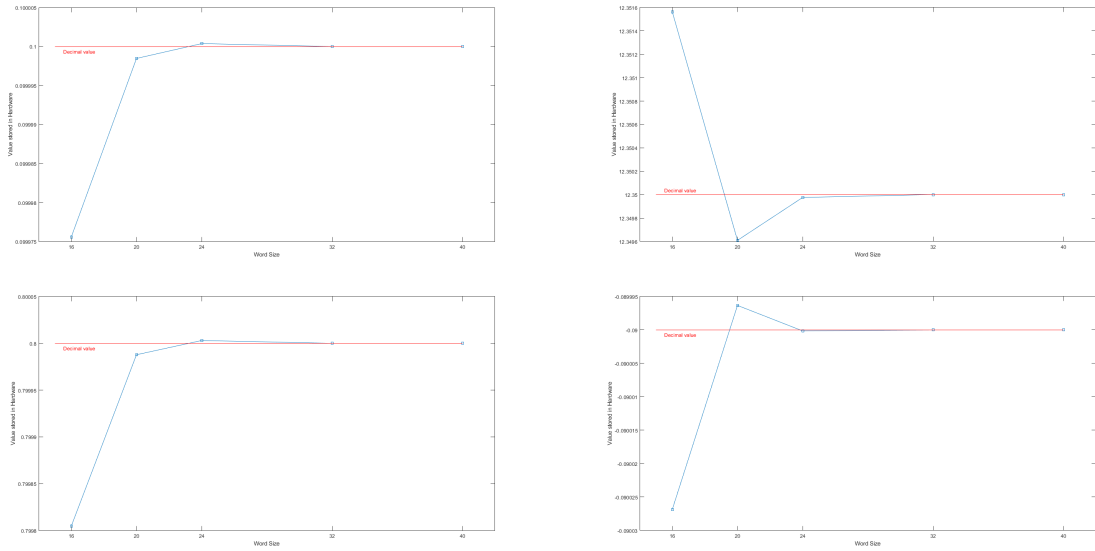


Figure 6.3: Changes in accuracy of representation in hardware for 0.1, 0.8, 12.35 and -0.09

6.3 Parameterisation of the Verilog Modules

The next step was parameterising the Verilog Modules in order to run comparative tests. The modules were parameterised "incrementally" (akin to the spiral model), using git as a means of version control. Once parameterised, 16-bit calculations were run to ensure accuracy. These results were compared and recorded in the same way as the experiments run for the 32-bit implementation in Section 5.1. The test bench was identical, with the exception of the change in parameters and initialised values. The implementation of each module was done as follows:


```

adder #(.WORD_MSB(15), .EXPONENT_MSB(4), .MANTISSA_MSB(9)) divtest (
//multiplier #(.WORD_MSB(15), .EXPONENT_MSB(4), .MANTISSA_MSB(9)) divtest (
//divider #(.WORD_MSB(15), .EXPONENT_MSB(4), .MANTISSA_MSB(9)) divtest (
    .input_a(A_INPUT),
    .input_b(B_INPUT),
    .input_a_stb(A_STB),
    .input_b_stb(B_STB),
    .output_z_ack(Z_ACK),
    .clk(CLK),
    .rst(),
    .output_z(RESULT),
    .output_z_stb(Z_STB),
    .input_a_ack(A_ACK),
    .input_b_ack(B_ACK)
);

```

The same decimal-value operands were used, but at 16 bit floating point, as follows:

```

initial begin
    domaths(16'h4A2D, 16'hC540);
    domaths(16'h609A, 16'h3A66);
    domaths(16'h0, 16'h705f);
    domaths(16'h5520, 16'hADC3);
    domaths(16'hFFFF, 16'h2E66);
    domaths(16'h4A2D, 16'h7C00);
    \${finish};
end

```

The results of the test bench in comparison to the expected values as generated by the online calculator [55] is recorded in Table XXIII. Waveforms are available in Appendix B.1.2.

Table XXIII: Comparing 16-bit Operations Between Verilog Modules and an Online Calculator

Addition			Multiplication			Division		
Online	DawsonJon	Cycles	Online	DawsonJon	Cycles	Online	DawsonJon	Cycles
471a	471a	13	d40e	d40e	12	c0b5	c0b5	60
609c	609c	21	5f5c	5f5c	13	61c1	61c1	61
705f	705f	4	0000	0000	4	0000	0000	4
551f	551f	20	c762	c762	13	e31e	e31e	61
ffff	fe00	4	ffff	fe00	4	ffff	fe00	4
7c00	7c00	4	7c00	7c00	4	0000	0000	4

6.4 SWAP at Varying Precisions

This section compares the resources required for implementations at particular bit widths for the different mathematical operations. The methods of determining these values (for resources, power and timing) is given in Section 5.3. All results are presented in table form in Appendix H.1. 40-bit division did not synthesise, and as a consequence is excluded from the results.

6.4.1 Resource Use Per Module

Figures 6.4 through 6.5 show the resource use (flip flops and look up tables) required for addition/subtraction, multiplication and division at various bit widths.

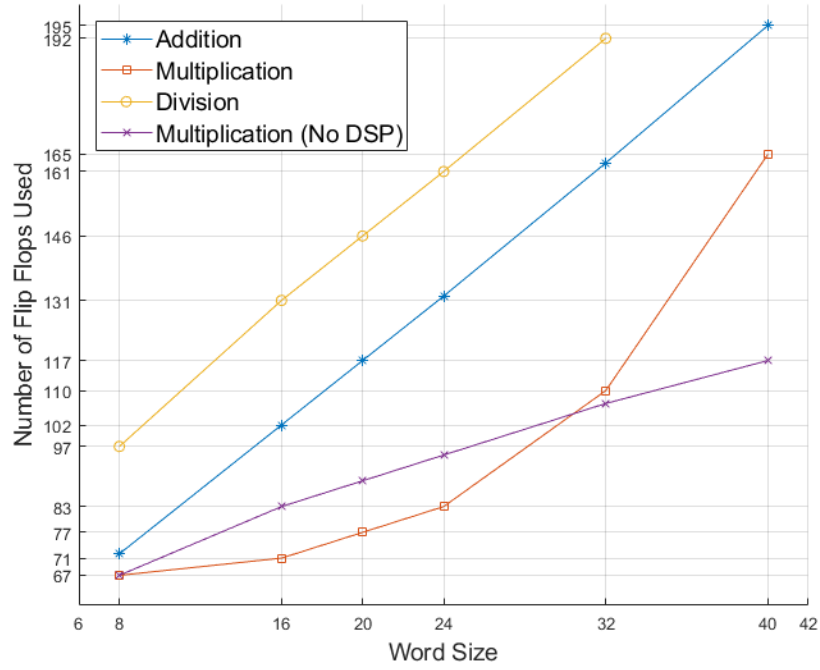


Figure 6.4: Flip flops required to implement various operations at varying precision

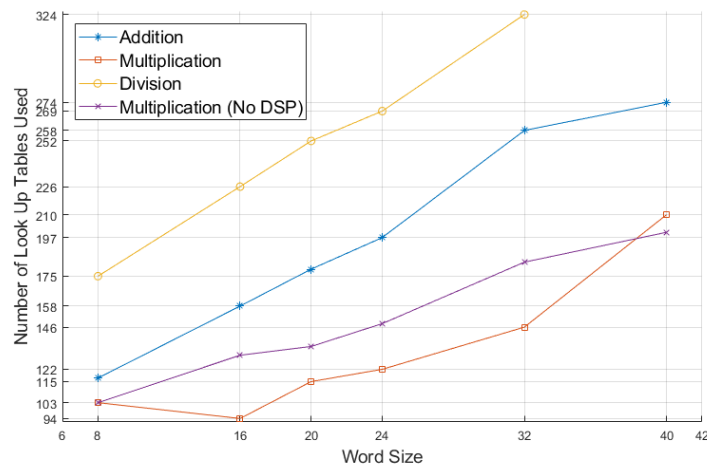


Figure 6.5: Look up tables required to implement various operations at varying precision

6.4.2 Effect of DSP Units on Resources Used

By setting the synthesis tag *-max_dsp 0*, Vivado forces the DSP units to not be used. This is set to compare the resources that can be absorbed by dedicated hardware resources. DSP units were only used in the multiplier.

Table XXIV: Comparison of Resources when Using DSP in the Multiplier

	WNS		LUT		FF	
bit width	With DSP	Without DSP	With DSP	Without DSP	With DSP	Without DSP
8	4.624	4.624	103	103	67	67
16	4.456	4.594	94	130	71	83
20	4.292	4.443	115	135	77	89
24	4.286	4.857	122	148	83	95
32	3.341	2.343	146	183	110	107
40	0.714	1.492	210	200	165	117

6.4.3 Power Usage

This subsection captures the power requirements of the different modules at varying precisions. The dynamic power is considered, as static power remains constant for the Nexys 4 DDR at 0.097 Watts. Figures 6.6 to 6.8 shows the changes in dynamic power for Addition, Multiplication and Division respectively. These provide indications of how power is used in the implementation. Figure 6.9 shows how the total dynamic power increases as bit width increases, and hence the overall requirements of the power consumption change.

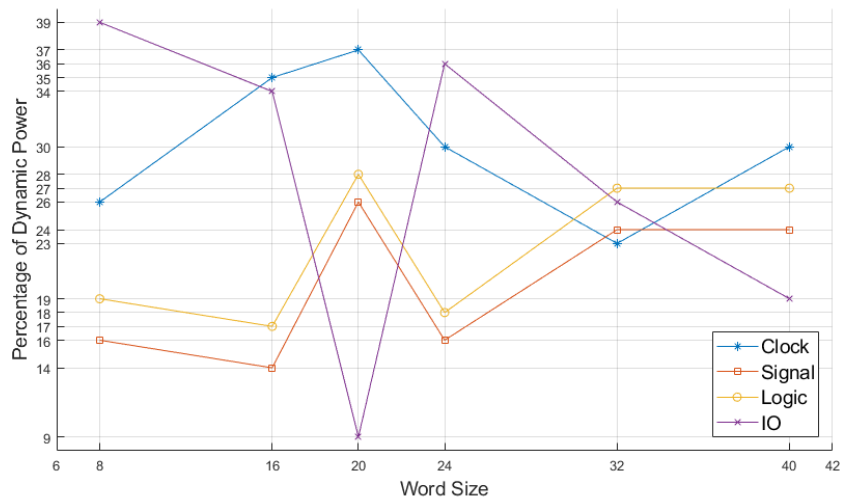


Figure 6.6: Change in dynamic power for Addition as bit width increases

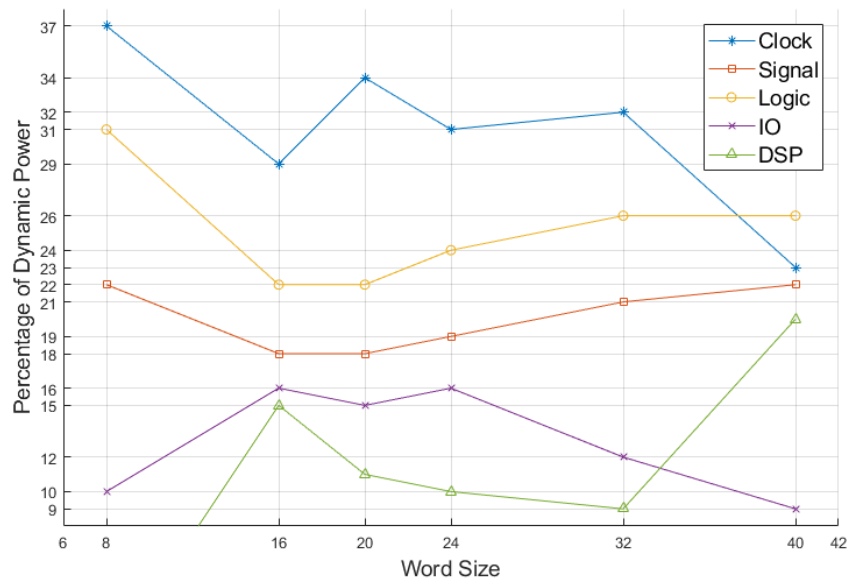


Figure 6.7: Change in dynamic power for multiplication as bit width increases

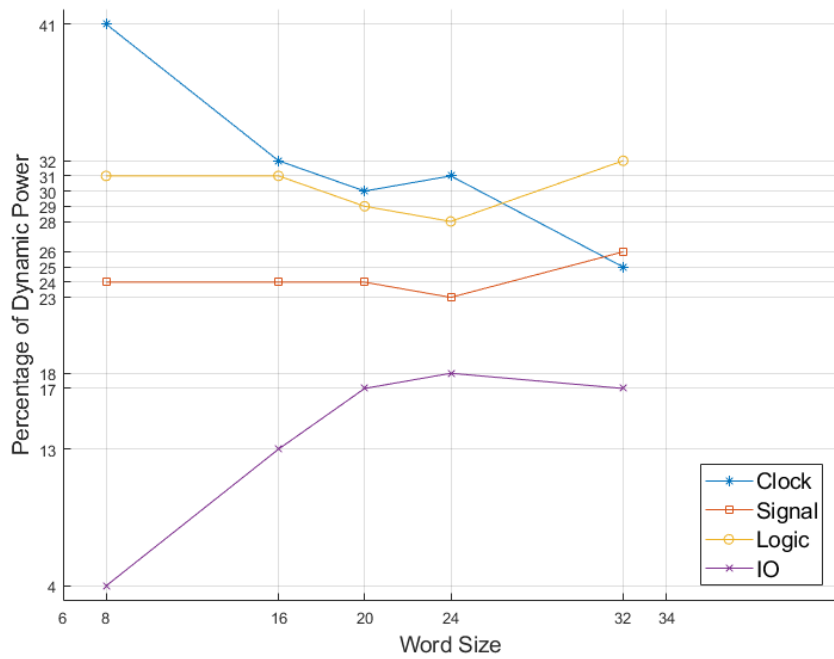


Figure 6.8: Change in dynamic power for division as bit width increases

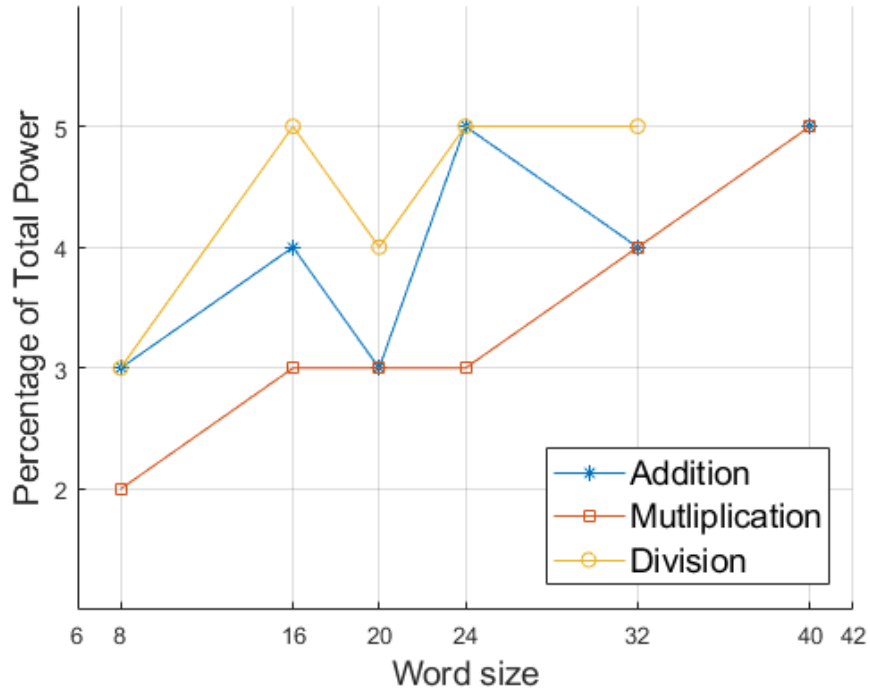


Figure 6.9: Change in dynamic power for the system as bit width increases

6.4.4 Time Required to Generate Bitstream

The time required to generate the bitstream for the various division modules was measured. The module (as implemented) is available in Appendix C.2. The results are shown in Figure 6.10. The minimum time taken is 74 seconds, with the maximum time being 112 seconds, a difference of 38 seconds.

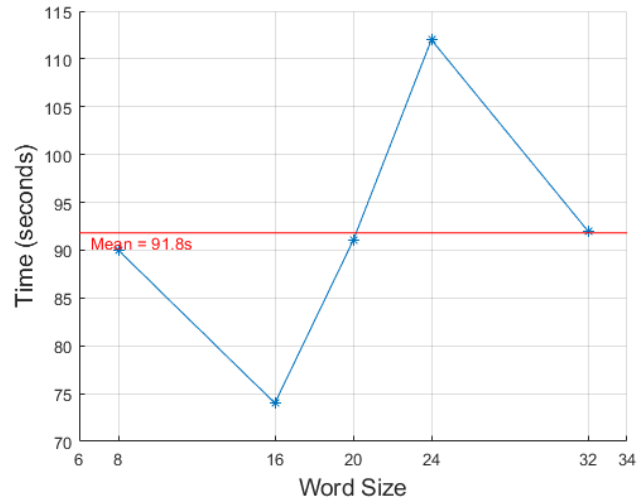


Figure 6.10: The time taken to generate a bitstream for various bit widths

6.4.5 Timing - WNS

For timing, worst negative slack (WNS) was recorded. Results are recorded in Table XXV. Figure 6.11 shows how WNS is affected as bit width is increased. A higher WNS indicates the design can run faster. As shown, greater bit widths decrease WNS, meaning tighter timing constraints. DSP has a large effect on WNS, as can be seen with the multiplication operation.

Table XXV: Change in WNS for varying bit widths

Word Size	Addition	Multiplication	Multiplication (No DSP)	Division
8	4.848	4.624	4.624	4.84
16	4.148	4.456	4.594	4.631
20	4.67	4.292	4.443	4.062
24	4.49	4.286	4.857	4.515
32	4.023	3.341	2.343	4.229
40	3.98	0.714	1.492	N/A

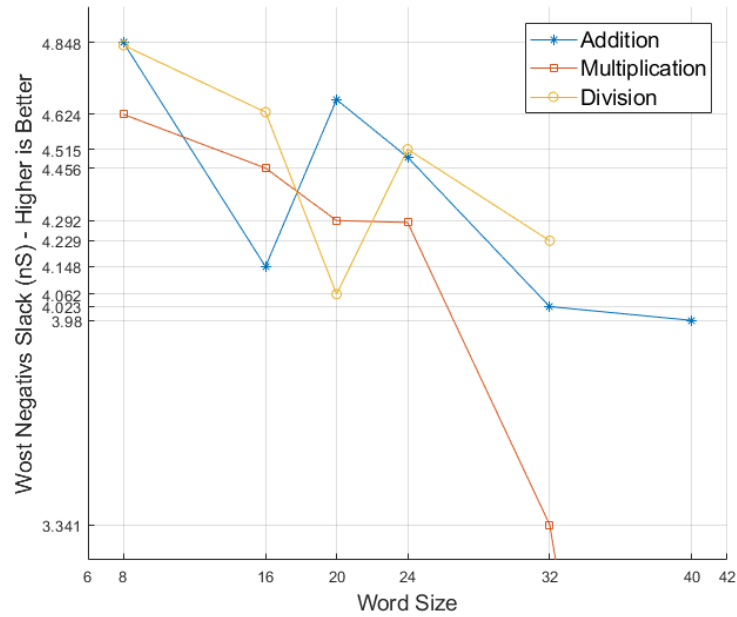


Figure 6.11: Change in WNS as bit width increases. Higher is better.

6.4.6 Timing - Speed of Execution

Clock cycles to produce results for 16 and 32 bit implementations were also recorded. Less clock cycles means a result is produced faster. These results are displayed in Table XXVI.

Table XXVI: Clock cycles taken for operations at 16 and 32 bits

Operation	Addition		Multiplication		Division	
	16 bit	32 bit	16 bit	32 bit	16 bit	32 bit
1	13	23	12	12	60	112
2	21	40	13	13	61	113
3	4	4	4	4	4	4
4	20	21	13	23	61	113
5	4	4	4	4	4	3
6	4	4	4	4	4	3

For the operations where speed up occurs, the results produced by the 16 bit and 32 bit calculation are recorded in Table XXVII below.

Table XXVII: Results produced by executing operations at different precisions resulting in speed-up

	Addition		Multiplication		Division	
Operation	16 bit	32 bit	16 bit	32 bit	16 bit	32 bit
1	7.1	7.1000004	-64.9	-64.8375	-2.354	-2.352381
2	590	589.9	471	471.2	736.5	736.25
4	81.94	81.91	-7.383	-7.38	-911	-911.1111

The speed up offered by moving to a smaller bit width is shown in Table XXVIII below, using Equation 8:

Table XXVIII: Speedup available by moving from 32 to 16 bit implementations

Operation	Addition	Multiplication	Division
1	1.769	1	1.867
2	1.905	1	1.852
4	1.05	1.769	1.852
Average Speedup	1.575	1.256	1.857

6.4.7 Accuracy and Precision

There is a loss in precision when moving to lower bit widths. Table XXIX below records the loss in precision ratio for the operations where speedup occurs, as defined by Equation 9. Recall that a result closer to 0 means that the 32-bit representation is closer to the value of the 64-bit representation, whereas a result of 2 means that the 16-bit representation is closer to the value of the 64-bit representation. It can be seen that 32-bit implementations are more accurate to the 64-bit representations than the 16-bit representations. For addition, precision is lost when moving to 32-bit. For multiplication and division, precision loss is negligible.

Table XXIX: Comparison of precision for calculations involving speedup

Operation	Addition	Multiplication	Division
1	2	0	0.00002941263128
2	0.5	0	0
4	0	0	0.000099999999907
Average	0.8333333333	0	0.00004313754345

When comparing 32 and 16-bit implementations, the difference between the values is compared. These results are recorded in Table XXX.

Table XXX: The difference between 16 and 32 bit values

Operation	Addition	Multiplication	Division
1	0.0000004	0.0625	0.00162
2	0.1	0.2	0.25
4	0.03	0.003	0.111
Average	0.0433	0.089	0.121

6.5 Comparison to Xilinx IP

This section compares the execution of the parameterized modules to that of the Xilinx IP available in Vivado. All experiments were run using a word size of 16 bits. The experiment was set up to be as quick to run as possible. As a result, the components used in all experiments are Xilinx BRAM, and the mathematical operator. The test bench used to obtain the data for the IP can be found in Appendix C.3.2, while the test bench used for the parameterized implementation can be found in Appendix C.3.3. All results used in this section can be found tabulated in Appendix G.

6.5.1 Comparison of Results

Table XXXI shows the comparison of the results produced by the modules. The operands used for each operation are the same as previous experiments. 0xfe00 and 0x7e00 are both representations of NaN.

Table XXXI: Comparison of results produced by the Vivado IP and the parameterized modules

Operation	Addition		Multiplication		Division	
	Xilinx IP	Parameterized	Xilinx IP	Parameterized	Xilinx IP	Parameterized
1	471a	471a	d40e	d40e	c0b5	c0b5
2	609c	609c	5f5c	5f5c	61c1	61c1
3	705f	705f	0000	0000	0000	0000
4	551f	551f	c762	c762	e31e	e31e
5	7e00	fe00	7e00	fe00	7e00	fe00
6	7c00	7c00	7c00	7c00	0000	0000

6.5.2 Resource Use

Figure 6.12 shows the resource use and requirements for the Xilinx IP versus the parameterized implementations. Figures 6.15 through 6.13 show the breakup of these resources.

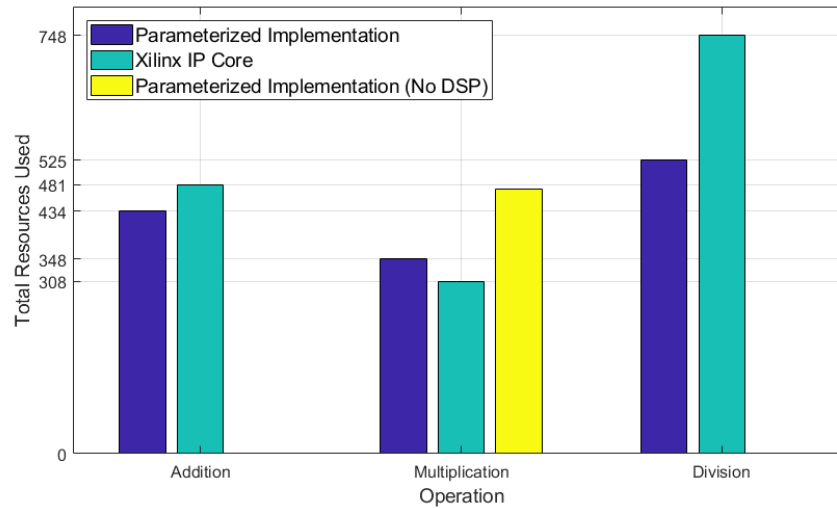


Figure 6.12: Total resources used across implementations

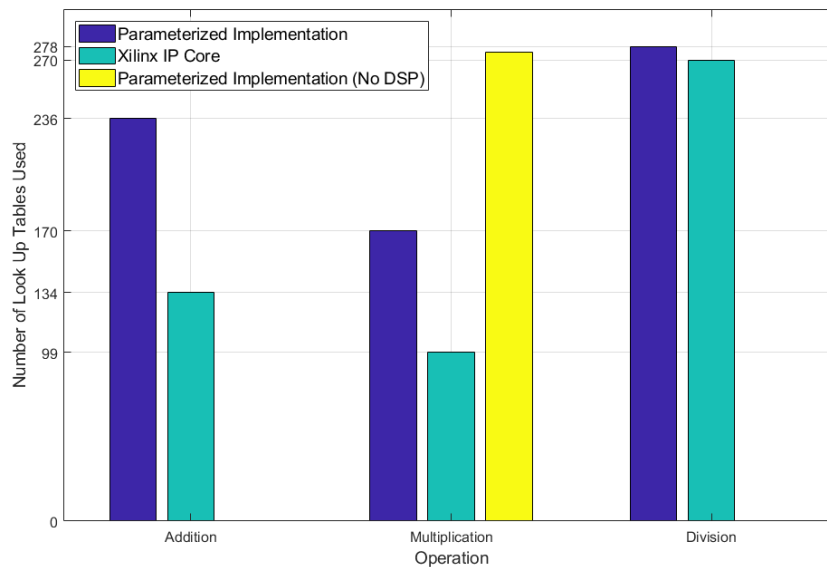


Figure 6.13: Look up tables used across implementations

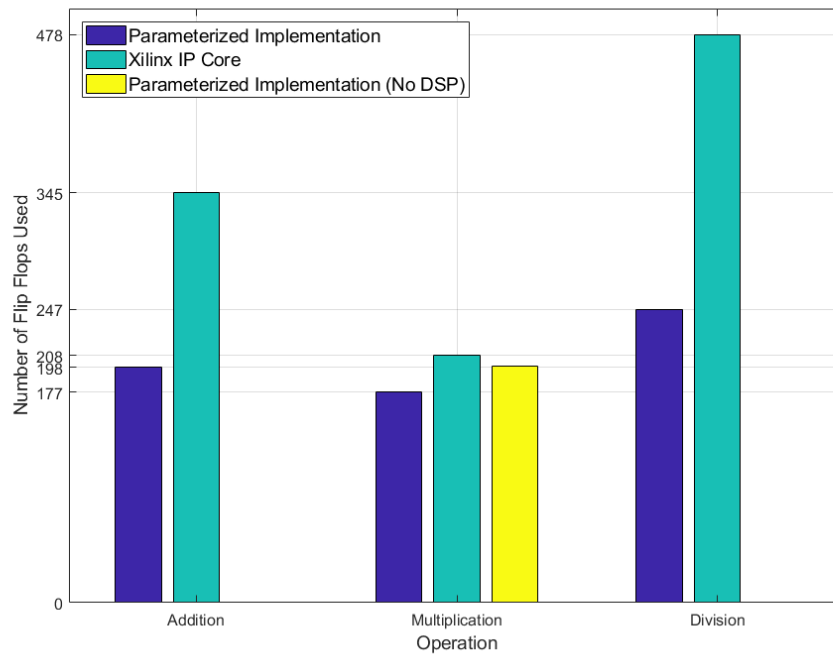


Figure 6.14: Flip flops used across implementations

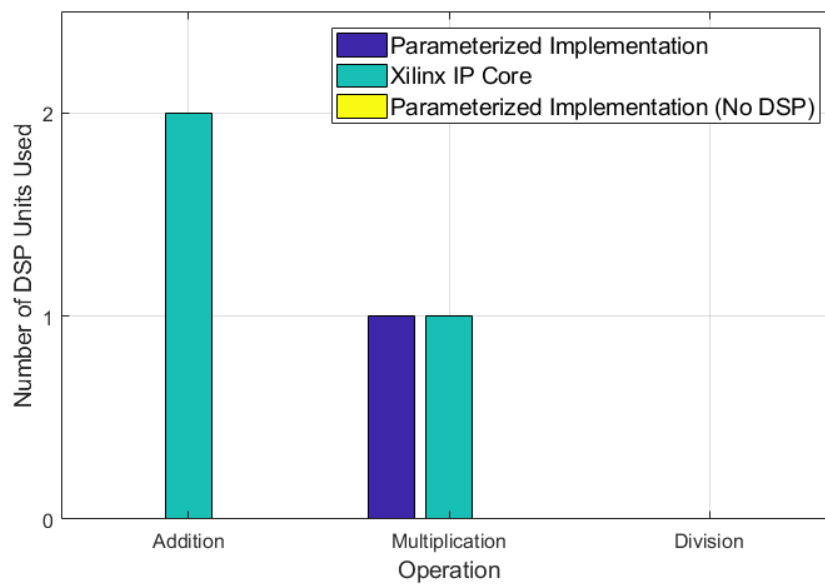


Figure 6.15: DSP units used across implementations

6.5.3 Power Usage

Figures 6.16 through 6.18 show how dynamic power requirements compare between the parameterized module and the Xilinx IP for addition, multiplication and division respectively. Figure 6.19 shows how the dynamic power (as a percentage of total power) increases when changing between the parameterized module and the Xilinx IP. Recall that dynamic power is the power required by the implementation of the module, and static power (the power consumed by the Nexys 4 DDR) remains constant at 0.097W.

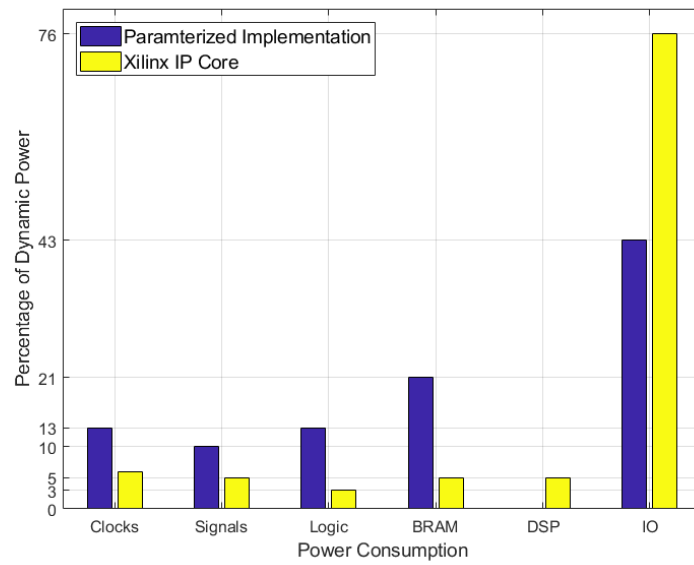


Figure 6.16: Change in dynamic power per signal type for addition

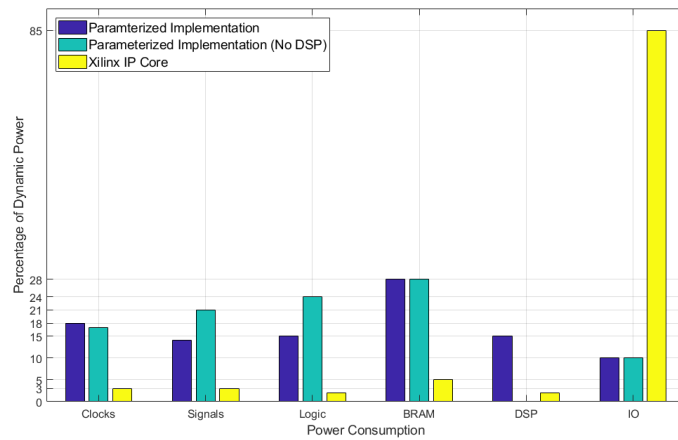


Figure 6.17: Change in dynamic power per signal type for multiplication

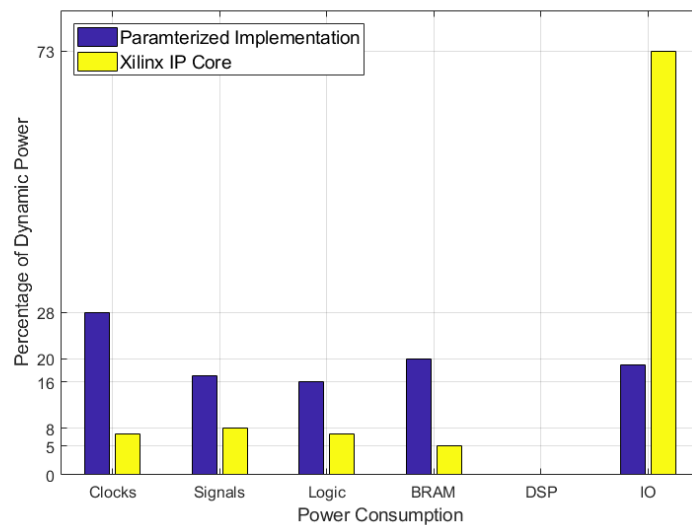


Figure 6.18: Change in dynamic power per signal type for division

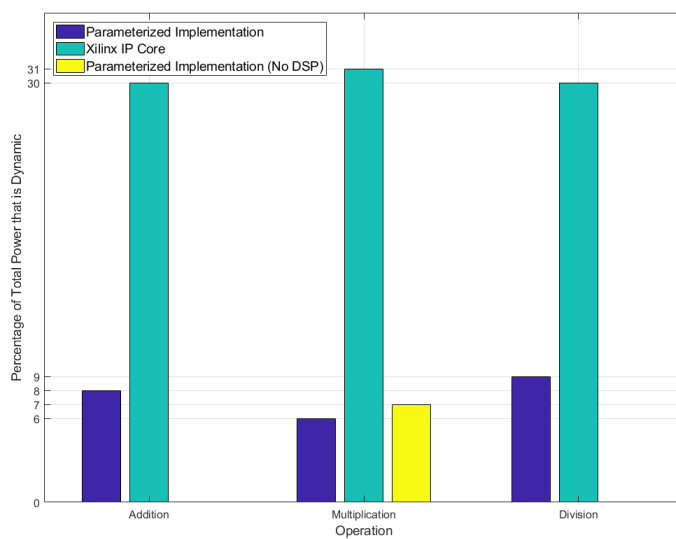


Figure 6.19: Change in total dynamic power between parameterized and Xilinx IP implementations

6.5.4 Timing

For timing, WNS and time to first and last result (in wall clock time) in nS was recorded. Both implementations ran at the same clock speed of 100MHz. The results for WNS are recorded in Table XXXII and the times to first/last result are shown in Table XXXIII below.

Table XXXII: The Worst Negative Slack for the Given Implementations

Module	Implementation	WNS (ns)
Addition	Parameterized	4.632
	Vivado IP	6
Multiply	Parameterized	4.116
	No DSP	1.713
	Vivado IP	6.43
Division	Parameterized	4.509
	Vivado IP	5.803

Table XXXIII: Difference in Execution Speeds for the Given Implementations

Module	Implementation	Time to First Result	Time to Last Result
Addition	Parameterized	35	183
	Vivado IP	29	181
Multiply	Parameterized	33	149
	No DSP	33	149
	Vivado IP	19	121
Division	Parameterized	129	437
	Vivado IP	37	229

6.6 Comprehensive Test: Heterodyning in MATLAB

This subsection details the results of the MATLAB implementation of the heterodyne experiment, as described in Section 5.5.

6.6.1 Justification of Using MATLAB as a Comparative Test Case

In order to ensure that MATLAB can be used as a valid means of comparison, the operations completed in Experiments 1, 3 and 4 (testing of the Verilog framework and comparison to Xilinx IP) were run in MATLAB. The code used to generate these results is available in Appendix F.1. Table XXXIV shows the results generated by the MATLAB script versus the results generated by the online IEEE-754 calculator available at [55] for 16-bit precisions. It was decided that the fp8 and fp16 casts provided by [59] were suitable to use as a comparison.

Table XXXIV: Comparison of fp16 Cast in MATLAB to Results Generated by 16-bit Calculator

Operands		Addition		Multiplication		Division	
A	B	fp16 Cast	Online Calculator	fp16 Cast	Online Calculator	fp16 Cast	Online Calculator
12.35	-5.25	7.1	7.1	-64.875	-64.9	-2.3535	-2.354
589	0.8	590	590	471	471	736.5	736.5
0	8950	8952	8950	0	0	0	0
82	-0.09	81.9	81.94	-7.3828	-7.383	-911	-911

6.6.2 Designs and Outputs of the Golden Measure

The "golden measure" (the heterodyne implemented at-64 bit floating point precision) returned the waveforms as shown in Figure 6.20:

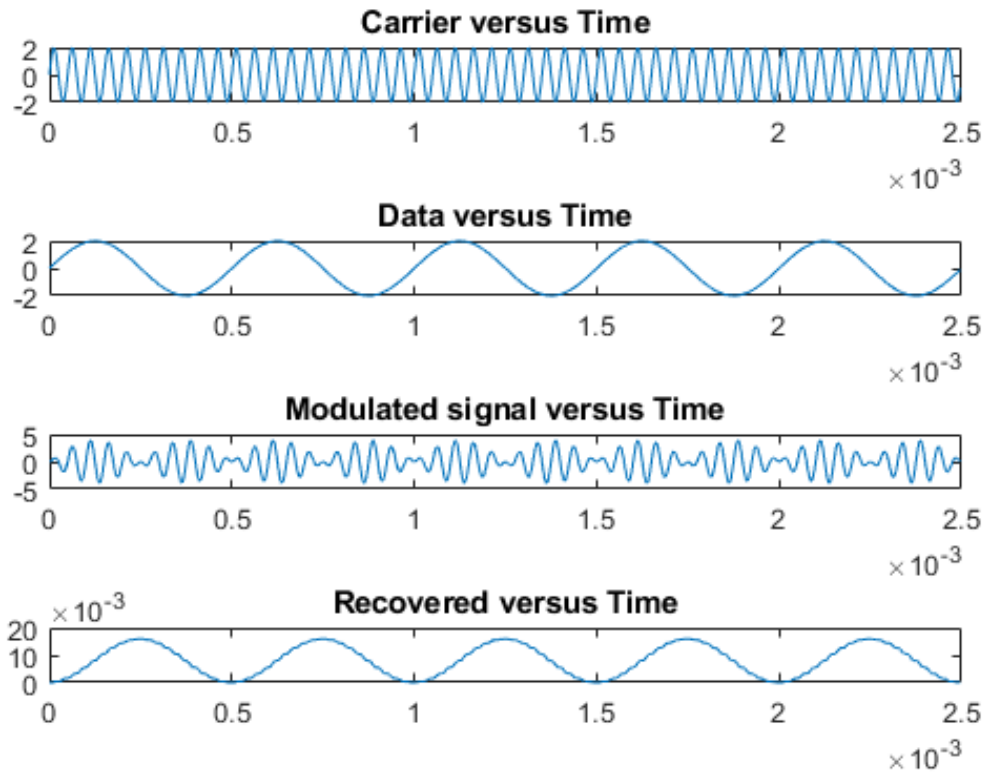


Figure 6.20: The resultant waveforms at IEEE-754 Double Precision (64-bits)

The comparison between the original and recovered signal is shown in Figure 6.21.

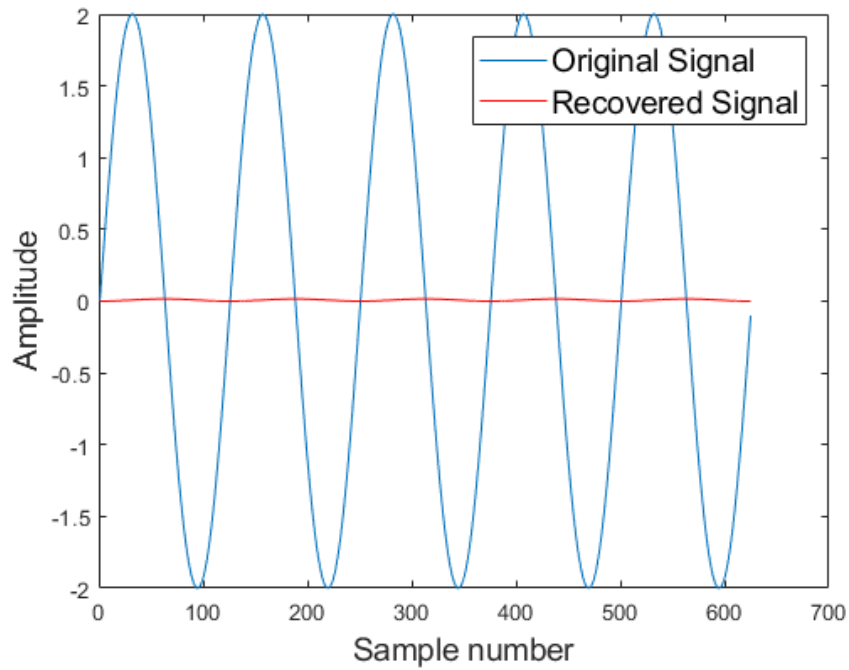


Figure 6.21: Comparison of original and recovered waveforms

Figure 6.22 below shows the original signal and recovered signal, when the recovered signal is shifted and scaled.

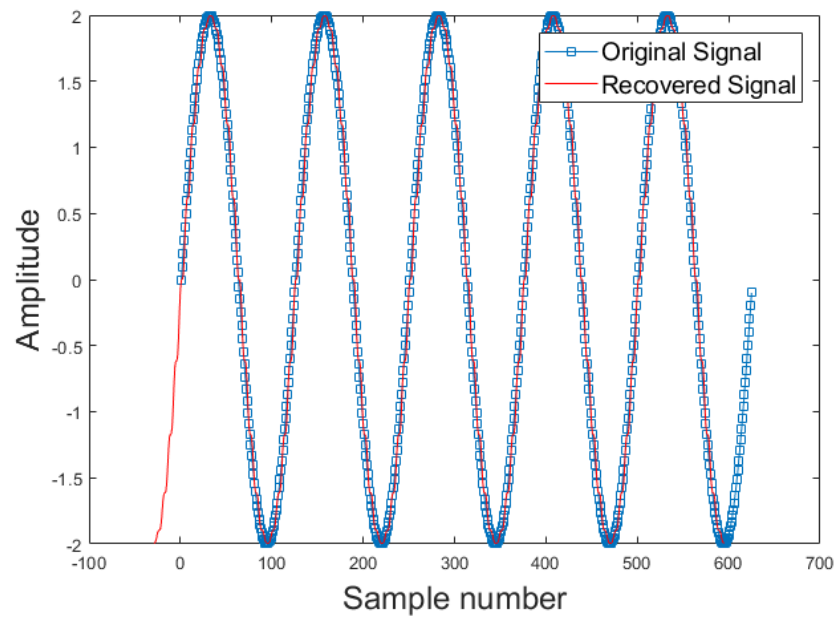


Figure 6.22: Scaled and shifted recovered waveform compared to original information signal

At this scale the signals are visually similar, however by zooming in (as in Figure 6.23) it can be seen that they do differ slightly.

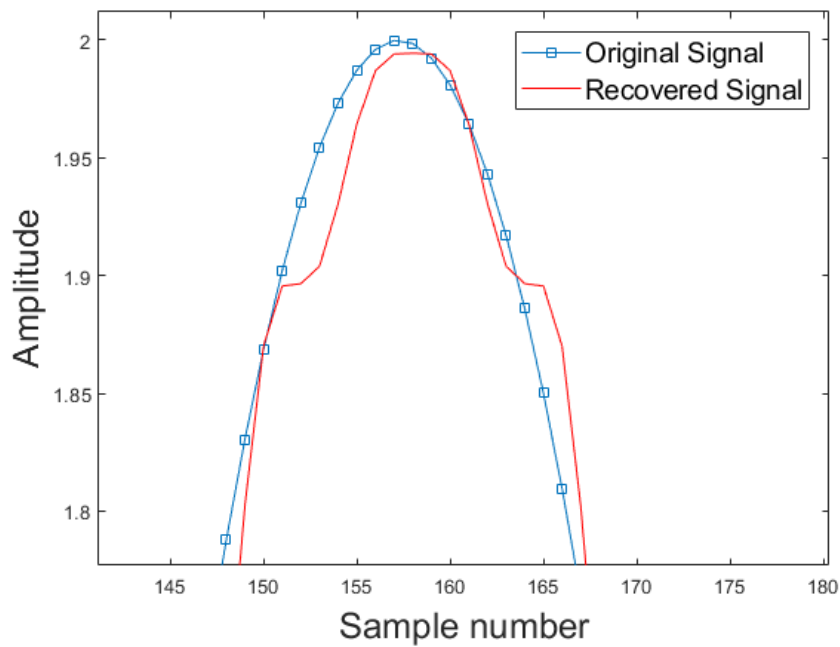


Figure 6.23: A zoomed scale-and-shift to highlight the differences between original waveform and recovered waveform

6.6.3 Comparing Outputs Across Bit Widths

This section compares the output from the filter when the simulation uses a single precision throughout the process. That is, four separate simulations were run, one at 8-bits, one at 16, one at 32 and one at 64-bits. Correlation was run between the full (double) precision output, and the output of the reduced precision waves. The results can be seen in Table XXXV below. It can be shown that the waves are sufficiently similar across all implementations.

Table XXXV: Correlation between double implementation and other bit-widths

Waveform	Correlation to Double (64-bit) implementation
fp8	0.999995261613458
fp16	0.999997620178028
Single (32-bit)	1
Double (64-bit)	1

Figure 6.24 shows these four waves as they are received. Although they appear visually similar, the waves do differ slightly in the peaks and troughs. In order to emphasise the difference between the waves, the scaling and shifting operation has been applied to all four waveforms, and the differences recorded in Table XXXVI.

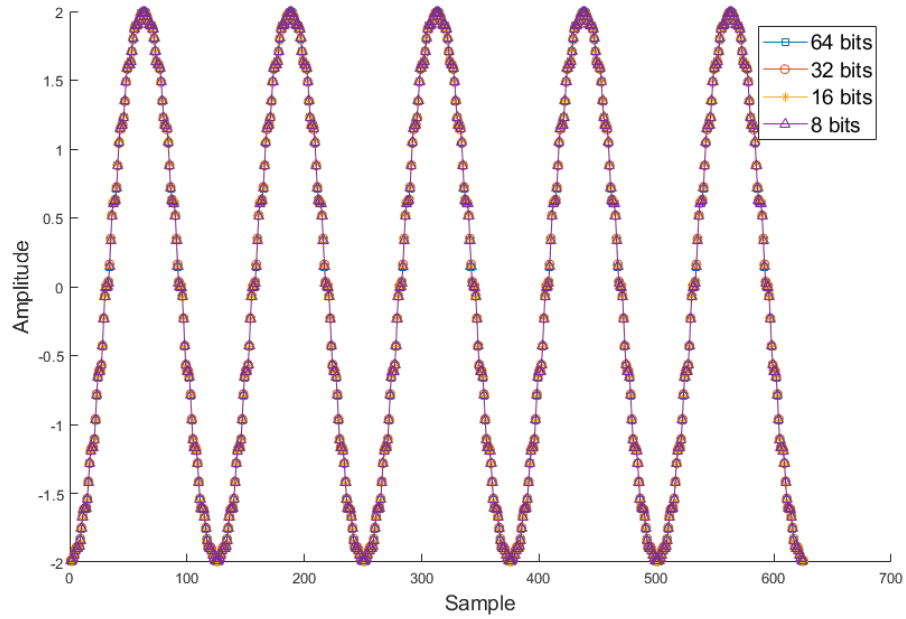


Figure 6.24: A graph showing the reconstructed information signal when using 8, 16, 32 and 64 bits throughout the process

Zooming in on the graph in Figure 6.24 produces the graph shown in Figure 6.25. While it appears that 16 and 8-bit implementations seem to offer similar shape and form to the 64-bit implementation, special note must be made of the difference between 32-bit and 64-bit implementations. The likely reason for the similarity between 8,16 and 64 bit implementations is that calculations for 8 and 16-bit are performed using 64-bit calculations. Using correct 8 and 16 bit calculations will likely yield results different to the ones presented in this experiment.

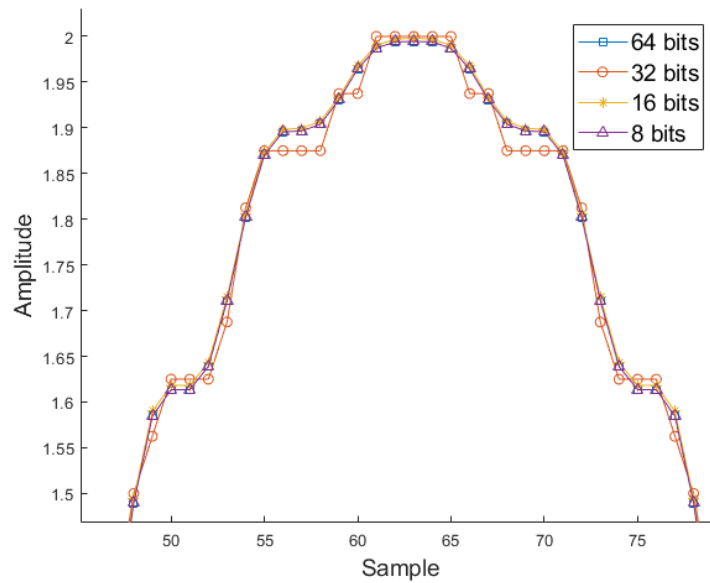


Figure 6.25: A graph showing the reconstructed information signal when using 8, 16, 32 and 64 bits throughout the process

Table XXXVI: Table showing differences in peak values of the scaled and shifted recovered signals

Bit width	Peak Amplitude	% Difference
8	1.9887	0.2858002407
16	1.9979	-0.1754913759
32	1.9944	0
64	1.9944	0

As can be seen, the peak amplitude for 8 and 16 bits varies, but by no more than a third of a percent. This could be considered insignificant in most applications.

Running a greater amplitude for the information carrying signal, however, produces a different result. Doubling the amplitude of the information carrying signal produces the following results, as recorded in Table XXXVII:

Table XXXVII: Table showing differences in peak values of the scaled and shifted recovered signals for double amplitude information-bearing signal

Bit width	Peak Amplitude	% Difference
8	-inf	-
16	3.9958	-0.1754913759
32	3.9888	0
64	3.9888	0

By increasing amplitude, it can be determined when various precisions break down. As shown in Table XXXVIII, fp8 breaks down when the modulated signal breaks past a value of 8, fp16 at 16, and single precision when the modulated signal exceeds $3.4E38$.

Table XXXVIII: The effect of continually increased amplitude on the output signals

Amplitude		Precision			
Carrier	x(t)	fp8	fp16	single	double
2	2^1	1.9887	1.9979	1.9944	1.9944
2	2^2	-inf	3.9958	3.9888	3.9888
2	2^3	0.9012	-inf	7.9777	7.9777
2	2^4	0.6318	1.7798	15.9554	15.9554
2	2^5	0.1375	0.8529	31.9108	31.9108
2	2^9	0	0	510.5726	510.5726
2	2^{127}	-inf	-inf	-inf	1.70E+38
2	2^{1023}	-inf	-inf	-inf	-inf

6.7 Achievable Speed Up of Use Case

From the experiments conducted in Section 5.6, the following results were obtained. It was shown that the Pi, with the correct compiler flag set, did support fp16. Table XXXIX below shows the type used in the C++ code, as well as the size of bytes required for representation, given by running `sizeof()` on a variable of the given type.

Table XXXIX: Types Supported by the Raspberry Pi

Type	sizeof()
<code>_fp16</code>	2
<code>float</code>	4
<code>double</code>	8

6.7.1 Speed Up Results

Table XL shows the averaged time (in milliseconds) of 5 runs for implementations at varying precisions using different FPU flags. These results are compared in Table XLI, where the "no FPU flag" is used as the baseline comparison for each bit width. Finally, table XLII shows the fastest and slowest implementations for each bit-width.

Table XL: Time Taken (in ms) for Multiplication Using Various Compiler Options

	none specified	vfpv3	vfpv3-fp16	fpv4	neon-fp-armv8	neon-fp16	vfpv3xd	vfpv3xd-fp16
fp16	0.123	0.1554	0.0602	0.0462	0.0774	0.047	0.1514	0.0474
fp32	0.0378	0.0734	0.037	0.0418	0.0422	0.045	0.0396	0.0724
fp64	0.0448	0.0666	0.0398	0.0402	0.1104	0.0764	0.1152	0.0846

Table XLI: Speed up obtained for Various Implementations

	none specified	vfpv3	vfpv3-fp16	fpv4	neon-fp-armv8	neon-fp16	vfpv3xd	vfpv3xd-fp16
fp16	1	0.792	2.043	2.662	1.589	2.617	0.812	2.595
fp32	1	0.515	1.022	0.904	0.896	0.84	0.955	0.522
fp64	1	0.673	1.126	1.114	0.406	0.586	0.389	0.53

Table XLII: Summarized Raspberry Pi Results

	Fastest Implementation	Time (ms)	Slowest Implementation	Time (ms)
fp16	fpv4	0.0462	vfpv3	0.1554
fp32	vfpv3-fp16	0.037	vfpv3	0.0734
fp64	vfpv3-fp16	0.0398	vfpv3xd	0.1152

The average times for fp16, fp32 and fp64 are 0.089, 0.049, and 0.072 milliseconds respectively. This puts fp16 as the slowest execution speed in this implementation. The fastest execution across all bit widths was achieved with the *-mfpu=fpv4* flag, whereas the slowest execution was with the *-mfpu=vfpv3xd* flag. As a brief reminder, the fpv4 flag uses the latest available version (on the Raspberry Pi 3B) of the hardware FPU accelerator, whereas vfpv3xd uses the third version of the floating point library, with added support for single precision.

7 Discussion and Conclusion

The aim of this investigation was to perform a cost benefit analysis on reducing word size for IEEE-754 floating point mathematical operations. This chapter summarizes the findings of the experiments recorded in Section 6.

7.1 Size of Implemented Design

Smaller word sized resulted in smaller designs. Usage of DSP resources on the FPGA reduced the number of logic gates required to implement the designs, allowing for those resources to be used elsewhere in the design. Use of the Xilinx IP required more resources than the standalone module implementations for the same word size.

7.2 Time for Implementing Design

While the recorded times for bitstream generation did vary slightly, the difference is negligible in terms of word clock time. The differences observed are likely due to background tasks running on the computer. The difference will likely be greater for larger implementations. This can be somewhat mitigated however by developing an IP core and moving the modules into out-of-context synthesis, requiring synthesis for the parameterised modules to be run only once.

7.3 Power Use

Greater word sizes required more power, but the effects in the implemented designs were negligible in comparison to the static power required by the FPGA. Using Xilinx IP had a considerable increase in power consumption - three to four times higher dynamic power than in comparison to the standalone combinatorial logic circuitry. The increase in execution speed may not be worth the increased power consumption, but further investigation in larger designs is needed.

7.4 Precision and Accuracy

While the range of the values used for calculations needs to be further expended on in future study, Sections 6.2 and 6.6 suggest that smaller bit widths can be used in implementations without too great a loss in precision, given that the range of the inputs is suitably scaled to produce an output in the range that can be accurately represented by a given bit size. A key takeaway from results obtained in 6.6 is that the disadvantage of smaller precisions is a

smaller dynamic range. Using lower precisions also results in higher frequency components (see Figure 6.25, comparing 64 and 32-bit implementations), but, given suitable filtering on the receiver end, this may be able to be mitigated.

7.5 Speed of Execution

Smaller bit widths were shown to have faster execution times under normal use cases. Due to the nature of the implementation, special values of the IEEE754 implementation such as Inf and NaN were returned as results in the same amount of clock cycles across implementations.

Xilinx IP was generally faster in producing the chain of results in 16-bit implementations. Addition showed no considerable improvement in speed, whereas multiplication finished slightly sooner and division using the Xilinx IP took half the time of the parameterized module.

Removal of DSP units had no effect on the speed of the multiplication operation implemented in the Verilog modules.

7.6 Comparison of Implemented Designs and Xilinx Developed IP

Xilinx offers implemented floating point modules. These were implemented at 16-bit (half precision) and compared to the parameterized modules at 16 bits. The IP used considerably more power, but only marginally more resources (with the exception of division). The worst negative slack is higher, which means Xilinx IP is better suited to faster designs.

7.7 Implementation on an Embedded System at Reduced Precision

The results recorded in Section 6.7 were not expected. Half size (fp16) calculations took the longest to execute, which was not only contrary to what was expected, but also contrary to the results produced in Section 6.4 and in particular Tables XXVII and XXVIII. It does go to show, however, that there is more pressure than ever on developers of high-performance systems to gain deeper insight into what may work best for an application (in this case, knowing which FPU provides the best performance for the multiplication of floating point numbers at a given precision).

7.8 Concluding remarks

The systems implemented in the experiments described in this paper suggest that using reduced precision is a viable option for increasing speed of execution while simultaneously

reducing power requirements and the size of the implemented design. The objectives and requirements presented in Section 1.1 were met in the following manner:

1. A framework was developed in Verilog to test IEEE754 calculations at arbitrary precisions
2. That framework was used to investigate the relationships between speed, size, and power for various bit widths
3. The variances in accuracy and precision when converting between a set of word sizes was measured and reported on
4. A real-world example that makes use of various bit-widths, and report on the performance and cost analysis was developed and reported on
5. Results were analysed and conclusions drawn about the speed of execution, size of implementation, power consumption and accuracy for varying word-sizes
6. Future work required to further investigate the questions raised in this paper was suggested.

In addition to meeting the above requirements, the project presented Verilog modules for parameterized implementations of floating point operations, and a python library to convert between arbitrary-sized bit-widths. These can be used to conduct further experiments on adjusting word size in larger systems.

It is clear from this investigation that further insights into calculations using arbitrary precisions is required. While lower precisions do offer an increase in speed, emphasis is placed on the person implementing the system to make use of these implementations. With the use of better compilers or a more streamlined design process (such as inclusion of MiniBit or BitSize), better speed up can be achieved.

As technology advances, we will likely see more unique architectures being developed, and more heterogeneous computing systems being developed, with an increase in the number of co-processors and edge-processors. While this has already been occurring continuously (see the development of dedicated hardware units, and co-processors such as graphics cards), there are limitations dictated by software compatibilities (see literature review on *History of Wordsizes*, and the reason for staying with the x86 architecture). We are in an exciting time for computing, as we are able to process more data, and as we do so more problems to solve become apparent to us. In order to progress, increase execution speed and more efficiently process data, frameworks need to be implemented in order to allow for more flexible, task-appropriate computing. FPGAs will likely become more prevalent in every day devices (they are already becoming available in most data centers) as a means of optimising particular execution. A scientist, for example, may one day have an FPGA embedded into a laptop

which reroutes whenever particular simulations are to be performed. It is an exciting time for hardware architects, software developers, and particularly end-users, with new technologies once thought impossible now being implemented (see Nvidia RTX and real-time ray tracing implementations). There is, however, onus on the developer to use these hardware features. For example, AMD's Vega Architecture offers hardware-level fp16 implementations, yet very few developers use this feature. This is not an uncommon case, though, as when better frameworks are released, the rate of adoption can be quite slow due to technical debt, or toolchains that make the move to new technologies or frameworks expensive.

8 Future Work

This paper records the investigation into varying precision for floating point implementations in hardware by implementing algorithms in HDL as well as examining a real world case scenario. It is believed that future work on arbitrarily-sized floating point implementations is warranted. The suggested future work is broken down as follows:

1. HDL Implementations

- (a) Comparison of various algorithms at different precisions as a means to compare speed up, and where particular algorithms may be most effective.
- (b) Enable pipelining to try and optimize one result per cycle. If this can be done in a parameterized implementation, it can be converted into an IP Block for use in other projects.
- (c) Power measurement should be better investigated. Preferably, a design should be instantiated to an FPGA, and the power measurement for each implementation be measured precisely.
- (d) Faster FPGAs should be tested, to determine the limit of clock speed at which the parameterized implementations can run.
- (e) Run designs through bit-width optimization programs in an attempt to further decrease size of implementation, and see what effect is had on SWAP, precision and speed.
- (f) Large designs using both Xilinx IP and the parameterized modules should be compared for speed and power comparisons.

2. MATLAB Experiments

- (a) A fully-implemented IEEE-754 compatible library for arbitrary bit widths should be implemented. The calculations should occur in the defined format, rather than being converted to double and converted back to the required format. This will allow for testing the efficacy of smaller bit-widths in terms of speed up.

3. Simulations and Experiments

- (a) Run more simulations for operations using a greater range of values. A good use case might be applying operations to a sine wave and exponential, as this will cover a large range of values.
- (b) The MATLAB use-case scenario tested in this investigation should be extended to include the entire heterodyning chain, not simply the MAC operations

- (c) Run the use-case scenario on hardware at incorrect precisions to determine how well dedicated hardware affects speed up of execution. For example, does running 16-bit implementations on 16-bit dedicated hardware offer any changes in speed in comparison to running it on 32-bit dedicated hardware?
- (d) Run more implementations at varying precisions. For example, it would be beneficial to investigate how machine learning applications may benefit from smaller word sizes.
- (e) Investigate dedicated hardware options. For example, AMD's Vega Architecture and the Rapid Packed Math implementations.
- (f) Compare speedup to fixed point implementations in a use-case scenario.

4. Embedded System Experiments

The unexpected result of the experiment on the Raspberry Pi leaves many questions. Some things to investigate further include:

- (a) Comparing the output of the assembly code generated to see which (if any) floating point instructions are used.
- (b) Using a 64-bit operation system on the Raspberry Pi as opposed to the 32-bit version.
- (c) Use a different compiler (there may be optimization issues)
- (d) Compile the code on the Raspberry Pi and compare results
- (e) Recompile Raspbian for ARMv7 (as opposed to ARMv6) and investigate any possible differences

This concludes suggestions for further work.

References

- [1] T. S. Perry, “David patterson says it’s time for new computer architectures and software languages,” Sep 2018. [Online]. Available: <https://spectrum.ieee.org/view-from-the-valley/computing/hardware/david-patterson-says-its-time-for-new-computer-architectures-and-software-languages>
- [2] J. Collins, “Testing the numerical precisions required to execute real world programs,” *International Journal of Software Engineering and Applications (IJSEA)*, vol. 8, p. 101, March 2017.
- [3] “AMD Opteron 246 - OSA246CEP5AL (OSA246BOX),” Date Accessed: 01/07/2018. [Online]. Available: [http://www.cpu-world.com/CPU%20s/K8/AMD-Opteron%20246%20-%20OSA246CEP5AL%20\(OSA246BOX\).html](http://www.cpu-world.com/CPU%20s/K8/AMD-Opteron%20246%20-%20OSA246CEP5AL%20(OSA246BOX).html)
- [4] P. A. Clayton, “Memory granule terms,” Date Accessed: 26/06/2018. [Online]. Available: <https://sites.google.com/site/paulclaytonplace/andy-glew-s-comparch-wiki/memory-granule-terms>
- [5] M. Satran, “Windows data types.” [Online]. Available: <https://docs.microsoft.com/en-za/windows/desktop/WinProg/windows-data-types>
- [6] “The story of the intel 4004,” Date Accessed: 28/06/2018. [Online]. Available: <https://www.intel.co.uk/content/www/uk/en/history/museum-story-of-intel-4004.html>
- [7] J. Brandon, “The cloud beyond x86: How old architectures are making a comeback,” Apr 2015, Date Accessed: 28/06/2018. [Online]. Available: <http://www.businesscloudnews.com/2015/04/15/a-rack-of-their-own-the-cloud-beyond-x86/>
- [8] *Intel 64 and IA-32 Architectures Software Developers Manual*, Intel, Jan. 2009.
- [9] “AMD Discloses New Technologies At Microprocessor Forum,” May 1999, Date Accessed: 28/06/2018. [Online]. Available: https://web.archive.org/web/20120308030806/http://www.amd.com/us/press-releases/Pages/Press_Release_751.aspx
- [10] Wikipedia contributors, “Word (computer architecture),” date Accessed: 28/06/2018. [Online]. Available: [https://en.wikipedia.org/wiki/Word_\(computer_architecture\)#Table_of_word_sizes](https://en.wikipedia.org/wiki/Word_(computer_architecture)#Table_of_word_sizes)
- [11] A. Kulkarni, “PAE - Physical Address Extension,” Oct 2017, Date Accessed: 01/07/2018. [Online]. Available: <https://medium.com/@aratik711/pae-physical-address-extension-dcec39257269>
- [12] J. Labrousse and G. A. Siavenburg, “A 500 mhz microprocessor with a very long instruction word architecture,” pp. 44,45, 1990.

- [13] J. A. Fisher, “Very long instruction word architectures and the eli-512,” 1983. [Online]. Available: <https://courses.cs.washington.edu/courses/cse548/13wi/Fisher-VLIW.pdf>
- [14] B. Casteilano, “How Does Word Length Affect the Performance and Operation of a CPU?” Oct 2012, Date Accessed: 01/07/2018. [Online]. Available: <https://bcastell.com/posts/word-length-vs-performance/>
- [15] T. Rauber and G. Runger, *Parallel Programming: for Multicore and Cluster Systems*. Springer Berlin Heidelberg, 2010.
- [16] G. Shobaki, N. Rmaileh, and J. Jamal, “Studying the impact of bit switching on cpu energy,” pp. 173,179, 20160523.
- [17] C Inacio and D Ombres, “The DSP decision: fixed point or floating?” *Spectrum, IEEE*, vol. 33, no. 9, pp. 72,74, 1996.
- [18] *AXD and armsd Debuggers Guide*, 3rd ed., ARM Limited, 2006.
- [19] R. Yates, *Fixed-Point Arithmetic: An Introduction*, Jan 2013.
- [20] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, pp. 1–70, Aug 2008.
- [21] M. Mano, C. R. Kime, and T. Martin, *Logic and Computer Design Fundamentals, 5th Ed.* Pearson Prentice Hall Upper Saddle River, NJ, 2016.
- [22] M. Taher, M. Aboulwafa, A. Abdelwahab, and E. Saad, “High-speed, area-efficient fpga-based floating-point arithmetic modules,” *Radio Science Conference, 2007. NRSC 2007. National*, pp. 1,8, 2007-03.
- [23] Karen Miller, “Lecture Notes - Floating Point Representation,” University of Wisconsin, Wisconsin, U.S.A.
- [24] “Unifying bit-width optimisation for fixed-point and floating-point designs.” USA: IEEE, 2004, pp. 79,88.
- [25] Patrick Schmid, “AMD FX: Energy Efficiency Compared To Eight Other CPUs,” *Tom’s Hardware*, October 2011. [Online]. Available: <https://www.tomshardware.com/reviews/fx-power-consumption-efficiency,3060-14.html>
- [26] “Summit supercomputer ranked fastest computer in the world,” *States News Service*, pp. States News Service, June 25, 2018, 2018-06-25.
- [27] “Top500 supercomputers.” [Online]. Available: <https://www.top500.org/>
- [28] “INA1x9 Data Sheet,” Texas Instruments, Dallas, Texas, U.S.A.

- [29] S. Anthony and UTC, “Usb-c power meter could save your devices from dodgy cables and chargers,” Jan 2017. [Online]. Available: <https://arstechnica.com/gadgets/2017/01/usb-c-power-meter/>
- [30] D. R. Martinez, M. M. Vai, and R. A. Bond, *High performance embedded computing handbook: A systems perspective*. CRC Press, 2008.
- [31] J. Novet, “Microsoft is luring a.i. developers to its cloud by offering them faster chips,” May 2018. [Online]. Available: <https://www.cnbc.com/2018/05/07/microsoft-is-luring-a-i-developer-by-offering-them-faster-chips.html>
- [32] “Amazon ec2 f1 instances.” [Online]. Available: <https://aws.amazon.com/ec2/instance-types/f1/>
- [33] Xilinx, “Series 7 FPGAs Configurable Logic Block User Guide,” *Xilinx, San Jose, CA*, vol. 1.8, 2016.
- [34] Xilinx, “7 Series DSP48E1 Slice User Guide,” *Xilinx, San Jose, CA*, vol. 1.10, 2018.
- [35] S. Winberg, “Parallel computing fundamentals,base core equivalents.” [Online]. Available: <http://www.rrsg.uct.ac.za/courses/EEE4084F/Resources.html>
- [36] A. O. Agerholm, “In the age of fpga,” Jan 2018. [Online]. Available: <https://www.napatech.com/age-of-fpga/>
- [37] J. M. Rabaey, “Low-power silicon architecture for wireless communications: embedded tutorial,” in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*. ACM, 2000, pp. 377–380.
- [38] B. Fagin and C. Renard, “Field programmable gate arrays and floating point arithmetic,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 3, pp. 365,367, 1994-09.
- [39] N. Shirazi, A. Walters, and P. Athanas, “Quantitative analysis of floating point arithmetic on fpga based custom computing machines,” *IEEE Symposium on FPGAs for Custom Computing Machines, Proceedings*, pp. 155,162, 1995.
- [40] L. Louca, T. A. Cook, and W. H. Johnson, “Implementation of ieee single precision floating point addition and multiplication on fpgas,” *IEEE Symposium on FPGAs for Custom Computing Machines, Proceedings*, pp. 107,116, 1996.
- [41] D.-U. Lee, A. Gaffar, O. Mencer, and W. Luk, “Minibit: Bit-width optimization via affine arithmetic,” 2005, pp. 837,840.

- [42] P. D. Düben, F. P. Russell, X. Niu, W. Luk, and T. Palmer, “On the use of programmable hardware and reduced numerical precision in earth-system modeling,” *Journal of advances in modeling earth systems*, vol. 7, no. 3, pp. 1393–1408, 2015.
- [43] Wikipedia contributors, “Spiral model - Wikipedia, The Free Encyclopedia,” 2018, Date Accessed: 01/07/2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Spiral_model&oldid=859478809
- [44] —, “V-Model - Wikipedia, The Free Encyclopedia,” <https://en.wikipedia.org/w/index.php?title=V-Model&oldid=852259910>, 2018, Date Accessed: 01/07/2018.
- [45] H. Foster, “Part 10: The 2016 wilson research group functional verification study,” Oct 2016. [Online]. Available: <https://blogs.mentor.com/verificationhorizons/blog/2016/10/31/part-10-the-2016-wilson-research-group-functional-verification-study/>
- [46] “Vivado design suite,” Date Accessed: 03/10/2018. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html>
- [47] “MATLAB Website.” [Online]. Available: <https://www.mathworks.com/products/matlab.html>
- [48] “SciPy Website.” [Online]. Available: <https://www.gnu.org/software/octave/>
- [49] “Python Website.” [Online]. Available: <https://www.python.org/>
- [50] “SciPy Website.” [Online]. Available: <https://www.scipy.org/>
- [51] “NumPy Website.” [Online]. Available: <https://www.numpy.org/>
- [52] S. Cass, “The 2018 top programming languages,” Jul 2018, Date Accessed: 03/10/2018. [Online]. Available: <https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages>
- [53] J. Dawson, “Dawsonjon fpu,” <https://github.com/dawsonjon/fpu>, 2017.
- [54] dawsonjon, “Dawsonjon fpu,” <https://github.com/dawsonjon/fpu/issues/12>, 2017.
- [55] E. Weitz, “IEEE 754 Calculator,” 2016. [Online]. Available: <http://weitz.de/ieee/>
- [56] Gribouillis, “Convert between various ieee754 floating formats.” Jan 2012, Date Accessed: 03/10/2018. [Online]. Available: <https://www.daniweb.com/programming/software-development/code/422746/convert-between-various-ieee754-floating-formats>
- [57] “Vivado Timing - Where can I find the Fmax of a clock signal in the timing report?” Date Accessed: 24/9/2018. [Online]. Available: <https://www.xilinx.com/support/answers/57304.html>

- [58] A. Wilkinson, “FDM and Heterodyning,” June 15 2017.
- [59] C. Moler, “Introducing cleve’s laboratory,” 2016. [Online]. Available: <https://www.mathworks.com/company/newsletters/articles/introducing-cleves-laboratory.html>
- [60] “LTC1606 Data Sheet,” Linear Technologies, California, U.S.A.
- [61] S. W. Smith *et al.*, “The scientist and engineer’s guide to digital signal processing,” 1997.
- [62] “Windows toolchain for raspberry/pi.” [Online]. Available: <http://gnutoolchains.com/raspberry/>
- [63] *ARM Compiler armclang Reference Guide*, 6th ed., ARM Limited, 2016.

Appendix

A Link To Resources

All code, parameters, screen shots and recorded data is available on GitHub (<https://github.com/kcranky/EEE4022S>), under the Creative Commons BY-NC-SA 3.0 licence. The versions on GitHub are guaranteed to be the latest implementations, and no such guarantee is made for correctness or completion of the files presented in the remainder of this Appendix.

B Testbenches

B.1 32-bit Precision Test Bench

```
'timescale 1ns / 1ps

module divTB;
    parameter WORD_MSB = 31;

    // Inputs
    reg [WORD_MSB:0] A_INPUT = 0;
    reg [WORD_MSB:0] B_INPUT = 0;
    reg A_STB = 1;
    reg B_STB = 1;
    reg Z_ACK = 0;
    reg CLK = 1;

    // Outputs
    wire A_ACK;
    wire B_ACK;

    wire Z_STB;
    wire [WORD_MSB:0] RESULT;

    adder addtest (
        //multiplier multtest (
        //divider divtest (
            .input_a(A_INPUT),
            .input_b(B_INPUT),
            .input_a_stb(A_STB),
            .input_b_stb(B_STB),
            .output_z_ack(Z_ACK),
            .clk(CLK),
            .rst(),
            .output_z(RESULT),
            .output_z_stb(Z_STB),
            .input_a_ack(A_ACK),
            .input_b_ack(B_ACK)
        );

    //Define the task
    task domaths;
    input [WORD_MSB:0] valA;
    input [WORD_MSB:0] valB;
    begin
        A_STB = 0;
        B_STB = 0;
        A_INPUT = valA;
        B_INPUT = valB;
```

```

A_STB = 1;
B_STB = 1;
wait(A_ACK == 1);
wait(B_ACK == 1);
Z_ACK = 0;
wait(Z_STB == 1);
Z_ACK = 1;
end
endtask

initial begin
    domaths(32'h4145999A, 32'hC0A80000); //0x40E33334 0xC281ACCD 0xC0168D69
    domaths(32'h44134000, 32'h3F4CCCCD); // A: 0x44137333 M: 0x43EB999A D: 0x44381000
    domaths(32'h0, 32'h460BD800); // 0x460BD800 0x00000000 0x00000000
    domaths(32'h42A40000, 32'hBDB851EC); // 0x42A3D1EC 0xC0EC28F6 0xC463C71C
    domaths(32'hFFFFFFF, 32'h3DCCCCCD); // 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF
    domaths(32'h4145999A, 32'h7F800000); //0x7F800000 0x7F800000 0x0
    $finish;
end

//Control the clock
always begin
    #1 CLK = ~CLK;
end

endmodule

```

B.1.1 32-bit Waveforms

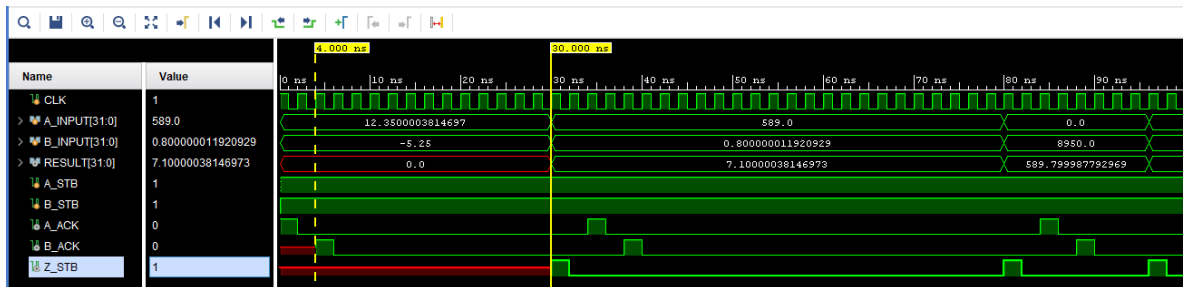


Figure B.1: Addition Waveform

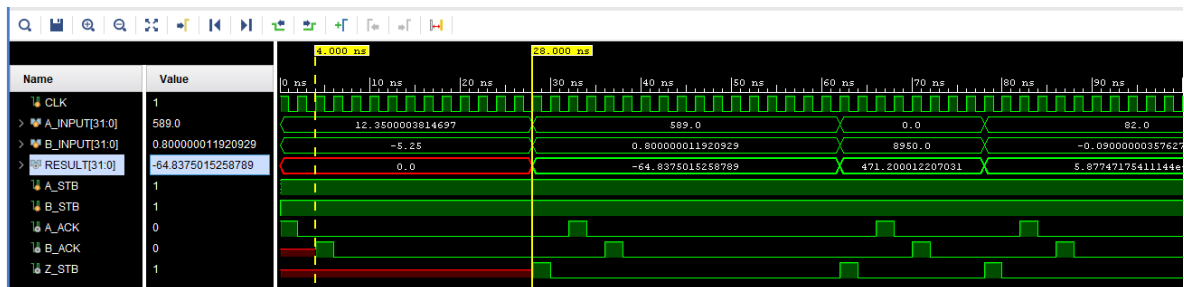


Figure B.2: Multiplication Waveform

B.1.2 16-bit Waveforms

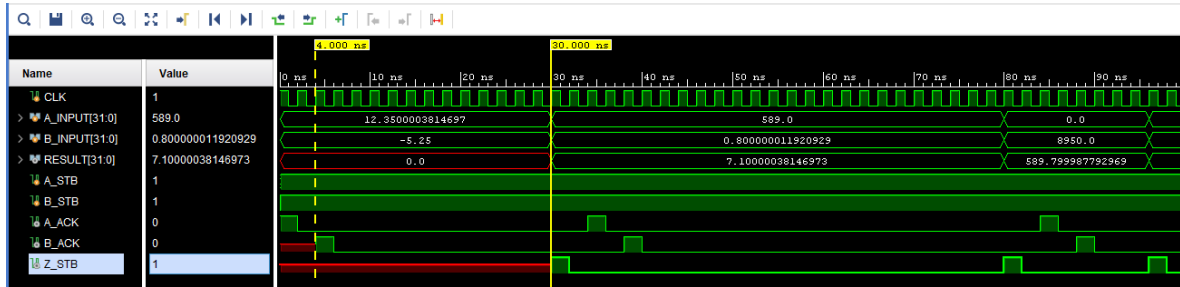


Figure B.3: Addition Waveform - 16 bit floating point

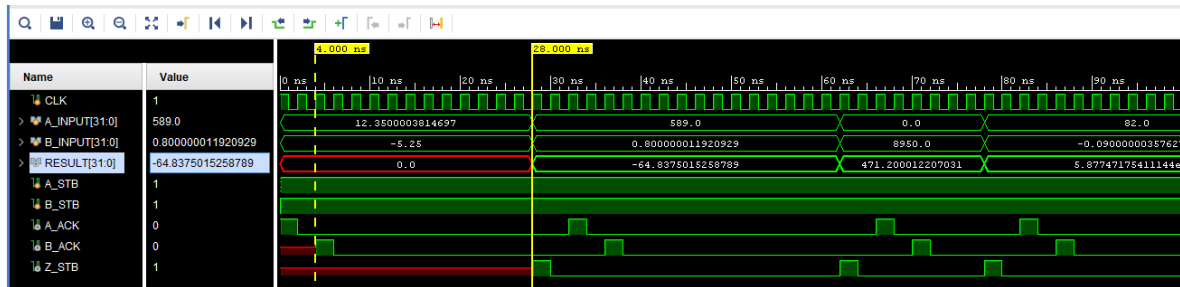


Figure B.4: Multiplication Waveform - 16 bit floating point

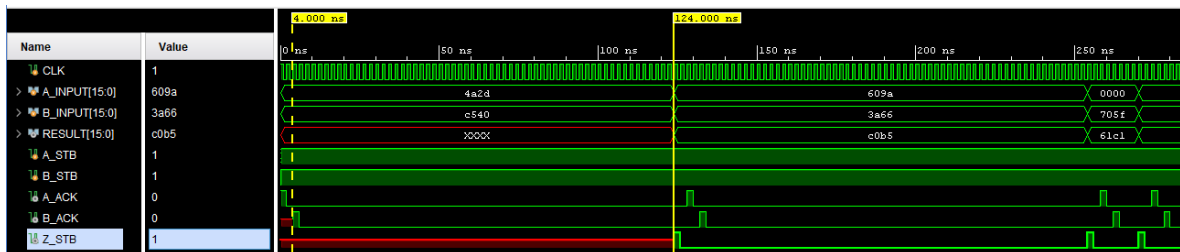


Figure B.5: Division Waveform - 16 bit floating point

C Verilog Code

C.1 Individual Parameterized Modules

C.1.1 Add

```
//IEEE Floating Point Adder (Single Precision)
//Copyright (C) Jonathan P Dawson 2013
//2013-12-12
//Parameterised by Keegan Crankshaw, September 2018

module adder #(parameter WORD_MSB=31, parameter EXPONENT_MSB=7, MANTISSA_MSB=22) (
```

```

    input_a,
    input_b,
    input_a_stb,
    input_b_stb,
    output_z_ack,
    clk,
    rst,
    output_z,
    output_z_stb,
    input_a_ack,
    input_b_ack);

parameter EXPONENT_BIAS = (2**(EXPONENT_MSB+1))/2-1;
parameter MANTISSA_MAX = 2**(MANTISSA_MSB+2)-1;

input      clk;
input      rst;

input      [WORD_MSB:0] input_a;
input      input_a_stb;
output     input_a_ack;

input      [WORD_MSB:0] input_b;
input      input_b_stb;
output     input_b_ack;

output     [WORD_MSB:0] output_z;
output     output_z_stb;
input      output_z_ack;

reg        s_output_z_stb;
reg        [WORD_MSB:0] s_output_z;
reg        s_input_a_ack;
reg        s_input_b_ack;

// Store states for FSM
reg        [3:0] state = 0;
parameter get_a = 4'd0,
          get_b   = 4'd1,
          unpack   = 4'd2,
          special_cases = 4'd3,
          align    = 4'd4,
          add_0    = 4'd5,
          add_1    = 4'd6,
          normalise_1 = 4'd7,
          normalise_2 = 4'd8,
          round    = 4'd9,
          pack     = 4'd10,
          put_z    = 4'd11;

reg        [WORD_MSB:0] a, b, z;
reg        [MANTISSA_MSB+4:0] a_m, b_m;
reg        [MANTISSA_MSB+1:0] z_m;
reg        [EXPONENT_MSB+1:0] a_e, b_e, z_e;
reg        a_s, b_s, z_s;
reg        guard, round_bit, sticky;
reg        [MANTISSA_MSB+5:0] sum;

always @(posedge clk)
begin
    case(state)

        get_a:
        begin
            s_input_a_ack <= 1;
            if (s_input_a_ack && input_a_stb) begin
                a <= input_a;
                s_input_a_ack <= 0;
                state <= get_b;
            end
        end

        get_b:
        begin

```

```

s_input_b_ack <= 1;
if (s_input_b_ack && input_b_stb) begin
    b <= input_b;
    s_input_b_ack <= 0;
    state <= unpack;
end
end

unpack:
begin
    a_m <= {a[MANTISSA_MSB : 0], 3'd0};
    b_m <= {b[MANTISSA_MSB : 0], 3'd0};
    a_e <= a[WORD_MSB-1 : MANTISSA_MSB+1] - EXPONENT_BIAS;
    b_e <= b[WORD_MSB-1 : MANTISSA_MSB+1] - EXPONENT_BIAS;
    a_s <= a[WORD_MSB];
    b_s <= b[WORD_MSB];
    state <= special_cases;
end

special_cases:
begin
    //if a is NaN or b is NaN return NaN
    if ((a_e == (EXPONENT_BIAS+1) && a_m != 0) || (b_e == (EXPONENT_BIAS+1) && b_m != 0)) begin
        z[WORD_MSB] <= 1;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1)-1);
        z[MANTISSA_MSB] <= 1;
        z[MANTISSA_MSB-1:0] <= 0;
        state <= put_z;
    end
    //if a is inf return inf
    end else if (a_e == (EXPONENT_BIAS+1)) begin
        z[WORD_MSB] <= a_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1)-1);
        z[MANTISSA_MSB:0] <= 0;
    end
    //if a is inf and signs don't match return nan
    if ((b_e == (EXPONENT_BIAS+1)) && (a_s != b_s)) begin
        z[WORD_MSB] <= b_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1)-1);
        z[MANTISSA_MSB] <= 1;
        z[MANTISSA_MSB-1:0] <= 0;
    end
    state <= put_z;
    //if b is inf return inf
    end else if (b_e == (EXPONENT_BIAS+1)) begin
        z[WORD_MSB] <= b_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1)-1);
        z[MANTISSA_MSB:0] <= 0;
        state <= put_z;
    end
    //if a is zero return b
    end else if (((signed(a_e) == -EXPONENT_BIAS) && (a_m == 0)) && (($signed(b_e) == -EXPONENT_BIAS) && (b_m == 0))) begin
        z[WORD_MSB] <= a_s & b_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= b_e[EXPONENT_MSB:0] + EXPONENT_BIAS;
        z[MANTISSA_MSB:0] <= b_m[MANTISSA_MSB+4:3];
        state <= put_z;
    end
    //if a is zero return b
    end else if (($signed(a_e) == -EXPONENT_BIAS) && (a_m == 0)) begin
        z[WORD_MSB] <= b_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= b_e[EXPONENT_MSB:0] + EXPONENT_BIAS;
        z[MANTISSA_MSB:0] <= b_m[MANTISSA_MSB+4:3];
        state <= put_z;
    end
    //if b is zero return a
    end else if (($signed(b_e) == -EXPONENT_BIAS) && (b_m == 0)) begin
        z[WORD_MSB] <= a_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= a_e[EXPONENT_MSB:0] + EXPONENT_BIAS;
        z[MANTISSA_MSB:0] <= a_m[MANTISSA_MSB+4:3];
        state <= put_z;
    end
    end else begin
        //Denormalised Number
        if ($signed(a_e) == -EXPONENT_BIAS) begin
            a_e <= -(EXPONENT_BIAS-1);
        end else begin
            a_m[MANTISSA_MSB+4] <= 1;
        end
        //Denormalised Number
        if ($signed(b_e) == -EXPONENT_BIAS) begin

```

```

        b_e <= -(EXPONENT_BIAS-1);
    end else begin
        b_m[MANTISSA_MSB+4] <= 1;
    end
    state <= align;
end

end

align:
begin
    if ($signed(a_e) > $signed(b_e)) begin
        b_e <= b_e + 1;
        b_m <= b_m >> 1;
        b_m[0] <= b_m[0] | b_m[1];
    end else if ($signed(a_e) < $signed(b_e)) begin
        a_e <= a_e + 1;
        a_m <= a_m >> 1;
        a_m[0] <= a_m[0] | a_m[1];
    end else begin
        state <= add_0;
    end
end

end

add_0:
begin
    z_e <= a_e;
    if (a_s == b_s) begin
        sum <= a_m + b_m;
        z_s <= a_s;
    end else begin
        if (a_m >= b_m) begin
            sum <= a_m - b_m;
            z_s <= a_s;
        end else begin
            sum <= b_m - a_m;
            z_s <= b_s;
        end
    end
    state <= add_1;
end

end

add_1:
begin
    if (sum[MANTISSA_MSB+5]) begin
        z_m <= sum[MANTISSA_MSB+5:4];
        guard <= sum[3];
        round_bit <= sum[2];
        sticky <= sum[1] | sum[0];
        z_e <= z_e + 1;
    end else begin
        z_m <= sum[MANTISSA_MSB+4:3];
        guard <= sum[2];
        round_bit <= sum[1];
        sticky <= sum[0];
    end
    state <= normalise_1;
end

normalise_1:
begin
    if (z_m[MANTISSA_MSB+1] == 0 && $signed(z_e) > -(EXPONENT_BIAS-1)) begin
        z_e <= z_e - 1;
        z_m <= z_m << 1;
        z_m[0] <= guard;
        guard <= round_bit;
        round_bit <= 0;
    end else begin
        state <= normalise_2;
    end
end

end

normalise_2:
begin
    if ($signed(z_e) < -(EXPONENT_BIAS-1)) begin
        z_e <= z_e + 1;
    end
end

```

```

        z_m <= z_m >> 1;
        guard <= z_m[0];
        round_bit <= guard;
        sticky <= sticky | round_bit;
    end else begin
        state <= round;
    end
end

round:
begin
    if (guard && (round_bit | sticky | z_m[0])) begin
        z_m <= z_m + 1;
        if (z_m == MANTISSA_MAX) begin
            z_e <= z_e + 1;
        end
    end
    end
    state <= pack;
end

pack:
begin
    z[MANTISSA_MSB : 0] <= z_m[MANTISSA_MSB:0];
    z[WORD_MSB-1 : MANTISSA_MSB+1] <= z_e[EXPONENT_MSB:0] + EXPONENT_BIAS;
    z[WORD_MSB] <= z_s;
    if ($signed(z_e) == -(EXPONENT_BIAS-1) && z_m[MANTISSA_MSB+1] == 0) begin
        z[WORD_MSB-1 : MANTISSA_MSB+1] <= 0;
    end
    //if overflow occurs, return inf
    if ($signed(z_e) > EXPONENT_BIAS) begin
        z[MANTISSA_MSB : 0] <= 0;
        z[WORD_MSB-1 : MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;
        z[WORD_MSB] <= z_s;
    end
    state <= put_z;
end

put_z:
begin
    s_output_z_stb <= 1;
    s_output_z <= z;
    if (s_output_z_stb && output_z_ack) begin
        s_output_z_stb <= 0;
        state <= get_a;
    end
end
endcase

if (rst == 1) begin
    state <= get_a;
    s_input_a_ack <= 0;
    s_input_b_ack <= 0;
    s_output_z_stb <= 0;
end

end
assign input_a_ack = s_input_a_ack;
assign input_b_ack = s_input_b_ack;
assign output_z_stb = s_output_z_stb;
assign output_z = s_output_z;
endmodule

```

C.1.2 Multiply

```

`timescale 1ns / 1ps
//IEEE Floating Point Multiplier (Single Precision)
//Copyright (C) Jonathan P Dawson 2013
//2013-12-12
//Parameterised by Keegan Crankshaw, September 2018
module multiplier #(parameter WORD_MSB=31, parameter EXPONENT_MSB=7, MANTISSA_MSB=22)(

```



```

    input_a,
    input_b,
    input_a_stb,
    input_b_stb,
    output_z_ack,
    clk,
    rst,
    output_z,
    output_z_stb,
    input_a_ack,
    input_b_ack);

parameter EXPONENT_BIAS = (2**(EXPONENT_MSB+1))/2-1;
parameter MANTISSA_MAX = 2**(MANTISSA_MSB+2)-1;
parameter PRODUCT_MSB = MANTISSA_MSB*2 +5; //squared + 2 + 3

input      clk;
input      rst;

input      [WORD_MSB:0] input_a;
input      input_a_stb;
output     input_a_ack;

input      [WORD_MSB:0] input_b;
input      input_b_stb;
output     input_b_ack;

output     [WORD_MSB:0] output_z;
output     output_z_stb;
input      output_z_ack;

reg        s_output_z_stb;
reg        [WORD_MSB:0] s_output_z;
reg        s_input_a_ack;
reg        s_input_b_ack;

reg        [3:0] state=0;
parameter get_a      = 4'd0,
          get_b      = 4'd1,
          unpack     = 4'd2,
          special_cases = 4'd3,
          normalise_a = 4'd4,
          normalise_b = 4'd5,
          multiply_0  = 4'd6,
          multiply_1  = 4'd7,
          normalise_1 = 4'd8,
          normalise_2 = 4'd9,
          round       = 4'd10,
          pack        = 4'd11,
          put_z       = 4'd12;

reg        [WORD_MSB:0] a, b, z;
reg        [MANTISSA_MSB+1:0] a_m, b_m, z_m;
reg        [EXPONENT_MSB+1:0] a_e, b_e, z_e;
reg        a_s, b_s, z_s;
reg        guard, round_bit, sticky;
reg        [PRODUCT_MSB:0] product;

always @(posedge clk)
begin
    case(state)

        get_a:
        begin
            s_input_a_ack <= 1;
            if (s_input_a_ack && input_a_stb) begin
                a <= input_a;
                s_input_a_ack <= 0;
                state <= get_b;
            end
        end

        get_b:
        begin

```

```

s_input_b_ack <= 1;
if (s_input_b_ack && input_b_stb) begin
    b <= input_b;
    s_input_b_ack <= 0;
    state <= unpack;
end
end

unpack:
begin
    a_m <= a[MANTISSA_MSB : 0];
    b_m <= b[MANTISSA_MSB : 0];
    a_e <= a[WORD_MSB-1 : MANTISSA_MSB+1] - EXPONENT_BIAS;
    b_e <= b[WORD_MSB-1 : MANTISSA_MSB+1] - EXPONENT_BIAS;
    a_s <= a[WORD_MSB];
    b_s <= b[WORD_MSB];
    state <= special_cases;
end

special_cases:
begin
    //if a is NaN or b is NaN return NaN
    if ((a_e == (EXPONENT_BIAS+1) && a_m != 0) || (b_e == (EXPONENT_BIAS+1) && b_m != 0)) begin
        z[WORD_MSB] <= 1;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1) -1);
        z[MANTISSA_MSB] <= 1;
        z[MANTISSA_MSB-1:0] <= 0;
        state <= put_z;
    //if a is inf return inf
    end else if (a_e == EXPONENT_BIAS+1) begin
        z[WORD_MSB] <= a_s ^ b_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1)-1);
        z[MANTISSA_MSB:0] <= 0;
        //if b is zero return NaN
        if (($signed(b_e) == -EXPONENT_BIAS) && (b_m == 0)) begin
            z[WORD_MSB] <= 1;
            z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1) -1);
            z[MANTISSA_MSB] <= 1;
            z[MANTISSA_MSB-1:0] <= 0;
        end
        state <= put_z;
    //if b is inf return inf
    end else if (b_e == EXPONENT_BIAS+1) begin
        z[WORD_MSB] <= a_s ^ b_s;
        z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1)-1);
        z[MANTISSA_MSB:0] <= 0;
        //if a is zero return NaN
        if (($signed(a_e) == -EXPONENT_BIAS) && (a_m == 0)) begin
            z[WORD_MSB] <= 1;
            z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**((EXPONENT_MSB+1) -1);
            z[MANTISSA_MSB] <= 1;
            z[MANTISSA_MSB-1:0] <= 0;
        end
        state <= put_z;
    //if a is zero return zero
    end else if (($signed(a_e) == -EXPONENT_BIAS) && (a_m == 0)) begin
        z<=0;
        state <= put_z;
    //if b is zero return zero
    end else if (($signed(b_e) == -EXPONENT_BIAS) && (b_m == 0)) begin
        z<=0;
        state <= put_z;
    end else begin
        //Denormalised Number
        if (($signed(a_e) == -EXPONENT_BIAS) begin
            a_e <= -(EXPONENT_BIAS-1);
        end else begin
            a_m[MANTISSA_MSB+1] <= 1;
        end
        //Denormalised Number
        if (($signed(b_e) == -EXPONENT_BIAS) begin
            b_e <= -(EXPONENT_BIAS-1);
        end else begin
            b_m[MANTISSA_MSB+1] <= 1;
        end
    end
end

```

```

        end
        state <= normalise_a;
    end
end

normalise_a:
begin
    if (a_m[MANTISSA_MSB+1]) begin
        state <= normalise_b;
    end else begin
        a_m <= a_m << 1;
        a_e <= a_e - 1;
    end
end

normalise_b:
begin
    if (b_m[MANTISSA_MSB+1]) begin
        state <= multiply_0;
    end else begin
        b_m <= b_m << 1;
        b_e <= b_e - 1;
    end
end

multiply_0:
begin
    z_s <= a_s ^ b_s;
    z_e <= a_e + b_e + 1;
    product <= a_m * b_m * 4;
    state <= multiply_1;
end

multiply_1:
begin
    //26 = PRODUCT_MSB-MANTISSA_MSB-1
    z_m <= product[PRODUCT_MSB:PRODUCT_MSB-MANTISSA_MSB-1];
    guard <= product[PRODUCT_MSB-MANTISSA_MSB-2];
    round_bit <= product[PRODUCT_MSB-MANTISSA_MSB-3];
    sticky <= (product[PRODUCT_MSB-MANTISSA_MSB-4:0] != 0);
    state <= normalise_1;
end

normalise_1:
begin
    if (z_m[MANTISSA_MSB+1] == 0) begin
        z_e <= z_e - 1;
        z_m <= z_m << 1;
        z_m[0] <= guard;
        guard <= round_bit;
        round_bit <= 0;
    end else begin
        state <= normalise_2;
    end
end

normalise_2:
begin
    if ($signed(z_e) < -(EXPONENT_BIAS-1)) begin
        z_e <= z_e + 1;
        z_m <= z_m >> 1;
        guard <= z_m[0];
        round_bit <= guard;
        sticky <= sticky | round_bit;
    end else begin
        state <= round;
    end
end

round:
begin
    if (guard && (round_bit | sticky | z_m[0])) begin
        z_m <= z_m + 1;
        if (z_m == MANTISSA_MAX) begin
            z_e <= z_e + 1;

```

```

        end
    end
    state <= pack;
end

pack:
begin
    z[MANTISSA_MSB : 0] <= z_m[MANTISSA_MSB:0];
    z[WORD_MSB-1 : MANTISSA_MSB+1] <= z_e[EXPONENT_MSB:0] + EXPONENT_BIAS;
    z[WORD_MSB] <= z_s;
    if ($signed(z_e) == -(EXPONENT_BIAS-1) && z_m[MANTISSA_MSB+1] == 0) begin
        z[WORD_MSB-1 : MANTISSA_MSB+1] <= 0;
    end
    //if overflow occurs, return inf
    if ($signed(z_e) > EXPONENT_BIAS) begin
        z[MANTISSA_MSB : 0] <= 0;
        z[WORD_MSB-1 : MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;
        z[WORD_MSB] <= z_s;
    end
    state <= put_z;
end

put_z:
begin
    s_output_z_stb <= 1;
    s_output_z <= z;
    if (s_output_z_stb && output_z_ack) begin
        s_output_z_stb <= 0;
        state <= get_a;
    end
end

endcase

if (rst == 1) begin
    state <= get_a;
    s_input_a_ack <= 0;
    s_input_b_ack <= 0;
    s_output_z_stb <= 0;
end

end

assign input_a_ack = s_input_a_ack;
assign input_b_ack = s_input_b_ack;
assign output_z_stb = s_output_z_stb;
assign output_z = s_output_z;

endmodule

```

C.1.3 Divide

```

//IEEE Floating Point Divider (Single Precision)
//Copyright (C) Jonathan P Dawson 2013
//2013-12-12
//Parameterised by Keegan Crankshaw, September 2018
module divider #(parameter WORD_MSB=31, parameter EXPONENT_MSB=7, MANTISSA_MSB=22)(
    input_a,
    input_b,
    input_a_stb,
    input_b_stb,
    output_z_ack,
    clk,
    rst,
    output_z,
    output_z_stb,
    input_a_ack,
    input_b_ack);

    parameter EXPONENT_BIAS = (2**(EXPONENT_MSB+1))/2-1;
    parameter MANTISSA_MAX = 2**(MANTISSA_MSB+2)-1;

```

```

parameter DIV_MSB = (MANTISSA_MSB+1)*2+1+3;
//50 = (22+1)*2+1+3
input      clk;
input      rst;

input      [WORD_MSB:0] input_a;
input      input_a_stb;
output     input_a_ack;

input      [WORD_MSB:0] input_b;
input      input_b_stb;
output     input_b_ack;

output     [WORD_MSB:0] output_z;
output     output_z_stb;
input      output_z_ack;

reg        s_output_z_stb;
reg        [WORD_MSB:0] s_output_z;
reg        s_input_a_ack;
reg        s_input_b_ack;

reg        [3:0] state=0;
parameter get_a      = 4'd0,
          get_b      = 4'd1,
          unpack      = 4'd2,
          special_cases = 4'd3,
          normalise_a  = 4'd4,
          normalise_b  = 4'd5,
          divide_0     = 4'd6,
          divide_1     = 4'd7,
          divide_2     = 4'd8,
          divide_3     = 4'd9,
          normalise_1  = 4'd10,
          normalise_2  = 4'd11,
          round        = 4'd12,
          pack         = 4'd13,
          put_z        = 4'd14;

reg        [WORD_MSB:0] a, b, z;
reg        [MANTISSA_MSB+1:0] a_m, b_m, z_m;
reg        [EXPONENT_MSB+2:0] a_e, b_e, z_e;
reg        a_s, b_s, z_s;
reg        guard, round_bit, sticky;
reg        [DIV_MSB:0] quotient, divisor, dividend, remainder;
reg        [5:0] count;

always @(posedge clk)
begin

    case(state)

        get_a:
        begin
            s_input_a_ack <= 1;
            if (s_input_a_ack && input_a_stb) begin
                a <= input_a;
                s_input_a_ack <= 0;
                state <= get_b;
            end
        end

        get_b:
        begin
            s_input_b_ack <= 1;
            if (s_input_b_ack && input_b_stb) begin
                b <= input_b;
                s_input_b_ack <= 0;
                state <= unpack;
            end
        end

        unpack:
        begin
            a_m <= a[MANTISSA_MSB : 0];

```

```

b_m <= b[MANTISSA_MSB : 0];
a_e <= a[WORD_MSB-1 : MANTISSA_MSB+1] - EXPONENT_BIAS;
b_e <= b[WORD_MSB-1 : MANTISSA_MSB+1] - EXPONENT_BIAS;
a_s <= a[WORD_MSB];
b_s <= b[WORD_MSB];
state <= special_cases;
end

special_cases:
begin
  //if a is NaN or b is NaN return NaN
  if ((a_e == EXPONENT_BIAS+1 && a_m != 0) || (b_e == EXPONENT_BIAS+1 && b_m != 0)) begin
    z[WORD_MSB] <= 1;
    z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
    z[MANTISSA_MSB] <= 1;
    z[MANTISSA_MSB-1:0] <= 0;
    state <= put_z;
    //if a is inf and b is inf return NaN
  end else if ((a_e == EXPONENT_BIAS+1) && (b_e == EXPONENT_BIAS+1)) begin
    z[WORD_MSB] <= 1;
    z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
    z[MANTISSA_MSB] <= 1;
    z[MANTISSA_MSB-1:0] <= 0;
    state <= put_z;
    //if a is inf return inf
  end else if (a_e == EXPONENT_BIAS+1) begin
    z[WORD_MSB] <= a_s ^ b_s;
    z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
    z[MANTISSA_MSB:0] <= 0;
    state <= put_z;
    //if b is zero return NaN
    if ($signed(b_e == -EXPONENT_BIAS) && (b_m == 0)) begin
      z[WORD_MSB] <= 1;
      z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
      z[MANTISSA_MSB] <= 1;
      z[MANTISSA_MSB-1:0] <= 0;
      state <= put_z;
    end
    //if b is inf return zero
  end else if (b_e == EXPONENT_BIAS+1) begin
    z[WORD_MSB] <= a_s ^ b_s;
    z[WORD_MSB-1:MANTISSA_MSB+1] <= 0;
    z[MANTISSA_MSB:0] <= 0;
    state <= put_z;
    //if a is zero return zero
  end else if (($signed(a_e) == -EXPONENT_BIAS) && (a_m == 0)) begin
    z[WORD_MSB] <= a_s ^ b_s;
    z[WORD_MSB-1:MANTISSA_MSB+1] <= 0;
    z[MANTISSA_MSB:0] <= 0;
    state <= put_z;
    //if b is zero return NaN
    if (($signed(b_e) == -EXPONENT_BIAS) && (b_m == 0)) begin
      z[WORD_MSB] <= 1;
      z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
      z[MANTISSA_MSB] <= 1;
      z[MANTISSA_MSB-1:0] <= 0;
      state <= put_z;
    end
    //if b is zero return inf
  end else if (($signed(b_e) == -EXPONENT_BIAS) && (b_m == 0)) begin
    z[WORD_MSB] <= a_s ^ b_s;
    z[WORD_MSB-1:MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
    z[MANTISSA_MSB:0] <= 0;
    state <= put_z;
  end else begin
    //Denormalised Number
    if ($signed(a_e) == -EXPONENT_BIAS) begin
      a_e <= -EXPONENT_BIAS+1;
    end else begin
      a_m[MANTISSA_MSB+1] <= 1;
    end
    //Denormalised Number
    if ($signed(b_e) == -EXPONENT_BIAS) begin
      b_e <= -EXPONENT_BIAS+1;
    end else begin

```

```

        b_m[MANTISSA_MSB+1] <= 1;
    end
    state <= normalise_a;
end
end

normalise_a:
begin
    if (a_m[MANTISSA_MSB+1]) begin
        state <= normalise_b;
    end else begin
        a_m <= a_m << 1;
        a_e <= a_e - 1;
    end
end

normalise_b:
begin
    if (b_m[MANTISSA_MSB+1]) begin
        state <= divide_0;
    end else begin
        b_m <= b_m << 1;
        b_e <= b_e - 1;
    end
end

divide_0:
begin
    z_s <= a_s ^ b_s;
    z_e <= a_e - b_e;
    quotient <= 0;
    remainder <= 0;
    count <= 0;
    //27 = 50-(22+1) = DIV_MSB-(MANTISSA_MSB+1)
    dividend <= a_m << (DIV_MSB-(MANTISSA_MSB+1));
    divisor <= b_m;
    state <= divide_1;
end

divide_1:
begin
    quotient <= quotient << 1;
    remainder <= remainder << 1;
    remainder[0] <= dividend[DIV_MSB];
    dividend <= dividend << 1;
    state <= divide_2;
end

divide_2:
begin
    if (remainder >= divisor) begin
        quotient[0] <= 1;
        remainder <= remainder - divisor;
    end
    if (count == DIV_MSB-1) begin
        state <= divide_3;
    end else begin
        count <= count + 1;
        state <= divide_1;
    end
end

divide_3:
begin
    z_m <= quotient[MANTISSA_MSB+4:3];
    guard <= quotient[2];
    round_bit <= quotient[1];
    sticky <= quotient[0] | (remainder != 0);
    state <= normalise_1;
end

normalise_1:
begin
    if (z_m[MANTISSA_MSB+1] == 0 && $signed(z_e) > -(EXPONENT_BIAS-1)) begin
        z_e <= z_e - 1;
    end
end

```

```

        z_m <= z_m << 1;
        z_m[0] <= guard;
        guard <= round_bit;
        round_bit <= 0;
    end else begin
        state <= normalise_2;
    end
end

normalise_2:
begin
    if ($signed(z_e) < -(EXPONENT_BIAS-1)) begin
        z_e <= z_e + 1;
        z_m <= z_m >> 1;
        guard <= z_m[0];
        round_bit <= guard;
        sticky <= sticky | round_bit;
    end else begin
        state <= round;
    end
end

round:
begin
    if (guard && (round_bit | sticky | z_m[0])) begin
        z_m <= z_m + 1;
        if (z_m == MANTISSA_MAX) begin
            z_e <= z_e + 1;
        end
    end
    state <= pack;
end

pack:
begin
    z[MANTISSA_MSB : 0] <= z_m[MANTISSA_MSB:0];
    z[WORD_MSB-1 : MANTISSA_MSB+1] <= z_e[EXPONENT_MSB:0] + EXPONENT_BIAS;
    z[WORD_MSB] <= z_s;
    if ($signed(z_e) == -(EXPONENT_BIAS-1) && z_m[MANTISSA_MSB+1] == 0) begin
        z[WORD_MSB-1 : MANTISSA_MSB+1] <= 0;
    end
    //if overflow occurs, return inf
    if ($signed(z_e) > EXPONENT_BIAS) begin
        z[MANTISSA_MSB : 0] <= 0;
        z[WORD_MSB-1 : MANTISSA_MSB+1] <= 2**(EXPONENT_MSB+1) -1;;
        z[WORD_MSB] <= z_s;
    end
    state <= put_z;
end

put_z:
begin
    s_output_z_stb <= 1;
    s_output_z <= z;
    if (s_output_z_stb && output_z_ack) begin
        s_output_z_stb <= 0;
        state <= get_a;
    end
end

endcase

if (rst == 1) begin
    state <= get_a;
    s_input_a_ack <= 0;
    s_input_b_ack <= 0;
    s_output_z_stb <= 0;
end

end
assign input_a_ack = s_input_a_ack;
assign input_b_ack = s_input_b_ack;
assign output_z_stb = s_output_z_stb;
assign output_z = s_output_z;

```



```
endmodule
```

C.2 Implementations for SWAP Measurements

```
'timescale 1ns / 1ps

module implementation(
    input CLK100MHZ,
    output [7:0] led
);

    reg [2:0] state = 0;
    parameter put = 4'd0,
               waitAck = 4'd1,
               waitRes = 4'd2,
               endex = 4'd3;

    // Inputs
    parameter WORD_MSB = 31;

    reg [WORD_MSB:0] A_INPUT = 8'b01101111; //15.5
    reg [WORD_MSB:0] B_INPUT = 8'b01101111; //15.5
    reg A_STB = 1;
    reg B_STB = 1;
    reg Z_ACK = 0;

    // Outputs
    wire A_ACK;
    wire B_ACK;

    wire Z_STB;
    wire [WORD_MSB:0] RESULT;
    assign led[7:0] = RESULT[7:0];

    divider #(.WORD_MSB(WORD_MSB), .EXPONENT_MSB(2), .MANTISSA_MSB(3)) multtest (
        .input_a(A_INPUT),
        .input_b(B_INPUT),
        .input_a_stb(A_STB),
        .input_b_stb(B_STB),
        .output_z_ack(Z_ACK),
        .clk(CLK100MHZ),
        .rst(0),
        .output_z(RESULT),
        .output_z_stb(Z_STB),
        .input_a_ack(A_ACK),
        .input_b_ack(B_ACK)
    );

    always @(posedge CLK100MHZ) begin
        case (state)

            put:
            begin
                A_STB = 1;
                B_STB = 1;
                state = waitAck;
            end

            waitAck:
            begin
                if (A_ACK == 1 && B_ACK == 1) begin
                    A_STB = 0;
                    B_STB = 0;
                    state = waitRes;
                end
            end

            waitRes:
            begin
                Z_ACK = 0;
            end
        endcase
    end
endmodule
```

```

        if (Z_STB == 1) begin
            Z_ACK = 1;
            state = endex;
        end
    end

    endex:
    begin
        // Do Nothing
    end

endcase

end;

endmodule

```

C.3 Test Benches

C.3.1 Coefficient File Used

```

memory_initialization_radix=16;
memory_initialization_vector=4A2D C540 609A 3A66 0000 705F 5520 ADC3 FFFF 2E66 4A2D 7C00;

```

C.3.2 IP Test bench

```

`timescale 1ns / 1ps

module tb(

);

// Memory IO
reg wea = 0;
reg web = 0;
reg [3:0] addra = 0;
reg [3:0] addrb = 1;
reg [15:0] dina=0; //We're not putting data in, so we can leave this unassigned
reg [15:0] dinb=0; //We're not putting data in, so we can leave this unassigned
wire [15:0] douta;
wire [15:0] doutb;

reg clk=0;

always begin
    #1 clk = ~clk;
end

// Maths IO
reg s_axis_a_tvalid = 0;           // input wire s_axis_a_tvalid
wire s_axis_a_tready;             // output wire s_axis_a_tready
reg [15:0] s_axis_a_tdata = 0;    // input wire [15 : 0] s_axis_a_tdata

reg s_axis_b_tvalid = 0;           // input wire s_axis_b_tvalid
wire s_axis_b_tready;             // output wire s_axis_b_tready
reg [15:0] s_axis_b_tdata = 0;    // input wire [15 : 0] s_axis_b_tdata

wire m_axis_result_tvalid;        // output wire m_axis_result_tvalid
reg m_axis_result_tready = 0;    // input wire m_axis_result_tready
wire [15:0] m_axis_result_tdata; // output wire [15 : 0] m_axis_result_tdata

reg result = 0;

reg [3:0] clkcnt = 0;

```

```

// State machine
reg [3:0] state = 0;
parameter fetch = 4'd0,
           put = 4'd1,
           getRes = 4'd2,
           endex = 4'd3,
           fetch2 = 4'd4;

always @(posedge clk) begin

    case (state)
    fetch:
    begin
        if(clkcnt == 2) begin
            s_axis_a_tdata <= douta;
            s_axis_b_tdata <= doutb;
            state <= put;
        end
        else
            clkcnt <= clkcnt+1;
        end

    put:
    begin
        if (s_axis_a_tready==1 && s_axis_b_tready==1) begin
            s_axis_a_tdata <= douta;
            s_axis_b_tdata <= doutb;

            s_axis_a_tvalid <= 1;
            s_axis_b_tvalid <= 1;

            addra <= addra + 2;
            addrb <= addrb + 2;
            m_axis_result_tready <= 1;
            state <= getRes;
        end
    end

    getRes: //Get the result!
    begin
        if (m_axis_result_tvalid==1 && result==0) begin
            s_axis_a_tvalid <= 0;
            s_axis_b_tvalid <= 0;
            result = 1;
            state <= endex;
        end
        else
            result <= m_axis_result_tvalid;
        end

    endex:
    begin
        if (addrb == 5'hf)
            $finish;
        else
            state <= put;
        end
    end

    endcase;
end

mem_16 memory (
    .clka(clk),    // input wire clka
    .wea(wea),     // input wire [0 : 0] wea
    .addra(addra), // input wire [4 : 0] addra
    .dina(dina),   // input wire [15 : 0] dina
    .douta(douta), // output wire [15 : 0] douta
    .clkb(clk),    // input wire clkb
    .web(web),     // input wire [0 : 0] web
    .addrb(addrb), // input wire [4 : 0] addrb
    .dinb(dinb),   // input wire [15 : 0] dinb
    .doutb(doutb)  // output wire [15 : 0] doutb

```

```

);

addition your_instance_name (
    .aclk(clk),                                // input wire aclk
    .s_axis_a_tvalid(s_axis_a_tvalid),         // input wire s_axis_a_tvalid
    .s_axis_a_tready(s_axis_a_tready),         // output wire s_axis_a_tready
    .s_axis_a_tdata(s_axis_a_tdata),           // input wire [15 : 0] s_axis_a_tdata
    .s_axis_b_tvalid(s_axis_b_tvalid),         // input wire s_axis_b_tvalid
    .s_axis_b_tready(s_axis_b_tready),         // output wire s_axis_b_tready
    .s_axis_b_tdata(s_axis_b_tdata),           // input wire [15 : 0] s_axis_b_tdata
    .m_axis_result_tvalid(m_axis_result_tvalid), // output wire m_axis_result_tvalid
    .m_axis_result_tready(m_axis_result_tready), // input wire m_axis_result_tready
    .m_axis_result_tdata(m_axis_result_tdata)   // output wire [15 : 0] m_axis_result_tdata
);

endmodule

```

C.3.3 IP Comparison Test bench

```

`timescale 1ns / 1ps

module implementation_tb(
);

reg clk=0;

always begin
    #1 clk = ~clk;
end

// Memory IO
reg wea = 0;
reg web = 0;
reg [5:0] addra = 0;
reg [5:0] addrb = 1;
reg [15:0] dina=0; //We're not putting data in, so we can leave this unassigned
reg [15:0] dinb=0; //We're not putting data in, so we can leave this unassigned
wire [15:0] douta;
wire [15:0] doutb;

// Maths IO
reg [15:0] mathina=0;
reg [15:0] mathinb=0;
reg A_STB = 0;
reg B_STB = 0;
reg Z_ACK = 0;
wire [15:0] result;
wire Z_STB;
wire A_ACK;
wire B_ACK;

// State machine
reg [3:0] state = 0;
parameter fetch = 4'd0,
            waitAck = 4'd1,
            getRes = 4'd2,
            endex = 4'd3,
            fetch2 = 4'd4;

reg [2:0] clkcnt = 0;

always @(posedge clk) begin
    clkcnt = clkcnt+1;

    case (state)

        fetch:
            begin
                if (clkcnt == 3) begin

```

```

        mathina <= douta;
        addra <= addra+2;

        mathinb <= doutb;
        addrb <= addrb+2;

        A_STB <= 1;
        B_STB <= 1;

        state <= waitAck;
    end
end

fetch2:
begin
    mathina = douta;
    addra = addra+2;

    mathinb = doutb;
    addrb = addrb+2;

    A_STB = 1;
    B_STB = 1;

    state = waitAck;
end

waitAck:
begin
    if (B_ACK == 1) begin
        state = getRes;
    end
end

getRes:
begin
    Z_ACK = 0;

    if (Z_STB == 1) begin
        Z_ACK = 1;
        state = endex;
    end
end

endex:
begin
    if (addrb == 5'd13)
        $finish;
    else
        state = fetch2;
    end
end

endcase;
end

blkmemgen memory (
    .clka(clk),    // input wire clka
    .wea(wea),     // input wire [0 : 0] wea
    .addra(addra), // input wire [4 : 0] addra
    .dina(dina),   // input wire [15 : 0] dina
    .douta(douta), // output wire [15 : 0] douta
    .clkb(clk),    // input wire clkb
    .web(web),     // input wire [0 : 0] web
    .addrb(addrb), // input wire [4 : 0] addrb
    .dinb(dinb),   // input wire [15 : 0] dinb
    .doutb(doutb)  // output wire [15 : 0] doutb
);

multiplier #(.WORD_MSB(15), .EXPONENT_MSB(4), .MANTISSA_MSB(9)) multiplier(
    .input_a(mathina),
    .input_b(mathinb),
    .input_a_stb(A_STB),
    .input_b_stb(B_STB),
    .output_z_ack(Z_ACK),
    .clk(clk),

```

```

        .rst(0),
        .output_z(result),
        .output_z_stb(Z_STB),
        .input_a_ack(A_ACK),
        .input_b_ack(B_ACK)
    );

endmodule

```

D Python Scripts

D.1 First Conversion Attempt

```

#Converter for arbitrary base floating points
from struct import *
from parameterDefinitions import Word40, Word32, Word24, Word20, Word16, Word8

def parse_bin_frac(s):
    return int(s[1:], 2) / 2.**(len(s) - 1)

def get_dec_value(binVal, struct):
    """Convert an-bit floating binary value to decimal"""
    sign = binVal[0]
    exponent = binVal[1:struct["EXPONENT_MSB"]+2]
    mantissa = binVal[struct["EXPONENT_MSB"]+2:struct["WORD_MSB"]+1]
    s = int(sign, 2)
    e = int(exponent, 2)
    m = parse_bin_frac("." + mantissa)
    result = pow(-1, s) * (1+m) * pow(2, e-struct["EXPONENT_BIAS"])
    print("{} converted to {}".format(binVal, result))
    return result

def get_bin_value(decVal, struct):
    """Convert a decimal number to n-bit floating point representation"""

    # Convert the integer input into one which the program understands
    decVal = str("{0:f}".format(float(decVal)))

    # Process the value
    integer, frac = decVal.split(".")
    frac=float(frac)/pow(10,len(frac))

    integer = integer.replace('-', '')
    i = bin(int(integer))[2:]
    f = ""

    #get the fractional part
    for j in range(struct["MANTISSA_MSB"]):
        frac = float(frac*2)
        t = ("{0:f}".format(float(frac))).split(".")
        f = str(f) + t[0]
        frac = float(t[1])/pow(10,len(t[1]))
        if(int(float(t[1]))==0):
            break

    # Normalise
    mantissa = i+f
    c = 0
    while mantissa[0] != "1":
        mantissa=mantissa[1:]
        c = c+1
        if len(mantissa)==0:
            break

    # Bias the exponent
    if (c != 0):
        exp = bin(c + struct["EXPONENT_BIAS"])

```

```

        exp = str(str(exp)[2:]).rjust(struct["EXPONENT_MSB"]+1,"0")
    else:
        exp = bin(len(i)-1+struct["EXPONENT_BIAS"])[2:]

    #Round to nearest, tie to even

    # Fill
    mantissa = str(str(mantissa)[1:]).ljust(struct["MANTISSA_MSB"]+1,"0")
    if (float(decVal) > 0):
        result = ("0"+exp+mantissa)[0:struct["WORD_MSB"]+1]
    else:
        result = ("1"+exp+mantissa)[0:struct["WORD_MSB"]+1]

    print("{}_converted_from_{}".format(result, decVal))
    print("Hex: {}".format(hex(int(result,2))))
    return result

if __name__ == "__main__":
    wordSize = Word16

    get_bin_value(12.35, wordSize)
    get_bin_value(-5.25, wordSize)
    get_bin_value(589, wordSize)
    get_bin_value(0.8, wordSize)
    get_bin_value(0, wordSize)
    get_bin_value(8950, wordSize)
    get_bin_value(0.1, wordSize)

    print("Finished")

```

D.2 AnyFloat Module

```

# ANYFLOAT MODULE
# https://www.daniweb.com/programming/software-development/code/422746/convert-between-various-ieee754-floating-formats

from __future__ import print_function
from collections import namedtuple
from fractions import Fraction
from math import isnan
import struct
import sys
from math import exp

version_info = (1, 1)
if sys.version_info < (2, 7):
    raise ImportError("Module anyfloat requires python 2.7 or newer.")
DEFAULT_SIZE = (11, 52)

def trunc_round(n, k):
    rshift = n.bit_length() - 1 - k
    if rshift >= 0:
        n >>= (rshift)
    else:
        n <<= (-rshift)
    return (n + 1) >> 1

def more_bin_digits(n, k):
    return bool(n >> k)

def unset_high_bit(n):
    assert n > 0
    return n ^ (1 << (n.bit_length() - 1))

def fbin(n, nbits):
    assert (0 <= n)
    assert not (n >> nbits)
    return "{val:0>{width}}".format(val = bin(n)[2:], width = nbits)
_anyfloat = namedtuple("anyfloat", "sign_exponent_significand")

```

```

class anyfloat(_anyfloat):
    __slots__ = ()
    _b32 = 1 << 32
    _b64 = 1 << 64

    def __new__(cls, sign, exponent, significand):
        assert sign in (0, 1)
        if significand:
            significand = significand//(significand & -significand)
        return _anyfloat.__new__(cls, sign, exponent, significand)

    @staticmethod
    def _encode(log2, mantissa, a, b):
        A = ~(~0 << a)
        AA = A >> 1
        if mantissa <= 0:
            return (A, 0) if (mantissa == -1) else (A, 1 << (b-1)) if mantissa else (0, 0)
        elif log2 <= -AA:
            nbits = b + log2 + AA
            rounded = trunc_round(mantissa, nbits) if (nbits >= 0) else 0
            return (1, 0) if more_bin_digits(rounded, b) else (0, rounded)
        elif log2 <= AA:
            rounded = trunc_round(mantissa, b + 1)
            return ((log2 + 1 + AA, 0) if (log2 < AA) else (A, 0) )
            if more_bin_digits(rounded, b+1) else (log2 + AA, unset_high_bit(rounded)) )
        else:
            return (A, 0)

    @staticmethod
    def _decode(exponent, significand, a, b):
        A = ~(~0 << a)
        AA = A >> 1
        assert 0 <= exponent <= A
        assert 0 <= significand < (1 << b)
        if exponent == A:
            return (0, -2 if significand else -1)
        elif exponent: # normal case
            return (exponent - AA, significand|(1 << b))
        else: # subnormal case
            if significand:
                return (significand.bit_length() - AA - b, significand)
            else:
                return (0, 0)

    def __float__(self):
        return self.int64_to_float(self.to_ieee())

    @classmethod
    def from_float(cls, x):
        """Create an anyfloat instance from a python float (64 bits double precision number)."""
        return cls.from_ieee(cls.float_to_int64(x))

    @classmethod
    def from_ieee(cls, n, size = DEFAULT_SIZE):
        """Create an anyfloat from an ieee754 integer.

        """
        """Create an anyfloat from an integer which binary representation is the ieee754
        format of a floating point number. The argument 'size' is a tuple (w,p)
        containing the width of the exponent part and the significand part in
        this ieee754 format."""
        w, p = size
        r = n >> p
        significand = (r << p) ^ n
        sign = int(r >> w)
        if not sign in (0, 1):
            raise ValueError(("Integer value out of range for ieee754 format", n, size))
        exponent = (sign << w) ^ r
        e, s = cls._decode(exponent, significand, w, p)
        if e == -2:
            sign = 0
        return cls(sign, e, s)

    def ieee_parts(self, size = DEFAULT_SIZE):
        w, p = size
        e, s = self._encode(self.exponent, self.significand, w, p)

```



```

        sign = 0 if (e + 1) >> w else self.sign
        return sign, e, s
    def to_ieee(self, size = DEFAULT_SIZE):
        """Convert to an IEEE754 integer.

        Convert self to an integer which binary representation is the IEEE754 format corresponding
        to the 'size' argument (read the documentation of from_ieee() for the meaning of the size
        argument).
        """
        sign, e, s = self.ieee_parts(size)
        return ((sign << size[0]) | e) << size[1] | s

    @classmethod
    def int64_to_float(cls, n):
        """Convert a 64 bits integer to a python float.

        This class method converts an integer representing a 64 bits floating point
        number in the IEEE754 double precision format to this floating point number."""

        if not (0 <= n < cls._b64):
            raise ValueError(("Integer value out of range for 64 bits IEEE754 format", n))
        u, v = divmod(n, cls._b32)
        return struct.unpack(">d", struct.pack(">LL", u, v))[0]

    @classmethod
    def float_to_int64(cls, x):
        """Convert a python float to a 64 bits integer.

        This class method converts a float to an integer representing this
        float in the 64 bits IEEE754 double precision format."""

        u, v = struct.unpack(">LL", struct.pack(">d", x))
        return (u << 32) | v

    def bin(self, size = DEFAULT_SIZE, sep=' '):
        """Return a binary representation of self.

        The returned string contains only the characters '0' and '1' and shows the
        IEEE754 representation of the real number corresponding to self with the given
        size (w, p).
        """
        if sep:
            sign, e, s = self.ieee_parts(size)
            return sep.join((fbin(sign, 1), fbin(e, size[0]), fbin(s, size[1])))
        else:
            return fbin(self.to_ieee(size), sum(size) + 1)

    def to_fraction(self):
        s = self.significant
        b = s.bit_length()
        k = self.exponent + 1 - b
        if k < 0:
            d = Fraction(s, 2 ** (-k))
        else:
            d = Fraction(s * 2**k, 1)
        return -d if self.sign else d

def main():
    val = exp(2)
    print ("exp(2)=", val)
    af = anyfloat.from_float(val)
    print (af)
    print (af.bin(), "(64 bits float)")
    print (" " * 2, af.bin(size = (8, 23)), "(32 bits)")
    print (" " * 7, af.bin(size = (3, 4)), "(8 bits)")
    print ("conversion to float works:", float(af) == val)

if __name__ == "__main__":
    #main()
    #exponent, mantissa

    af = anyfloat.from_float(0.1)

    s = (3, 4)
    print(af.bin(size = s))

```

E Vivado

E.1 Creating a project

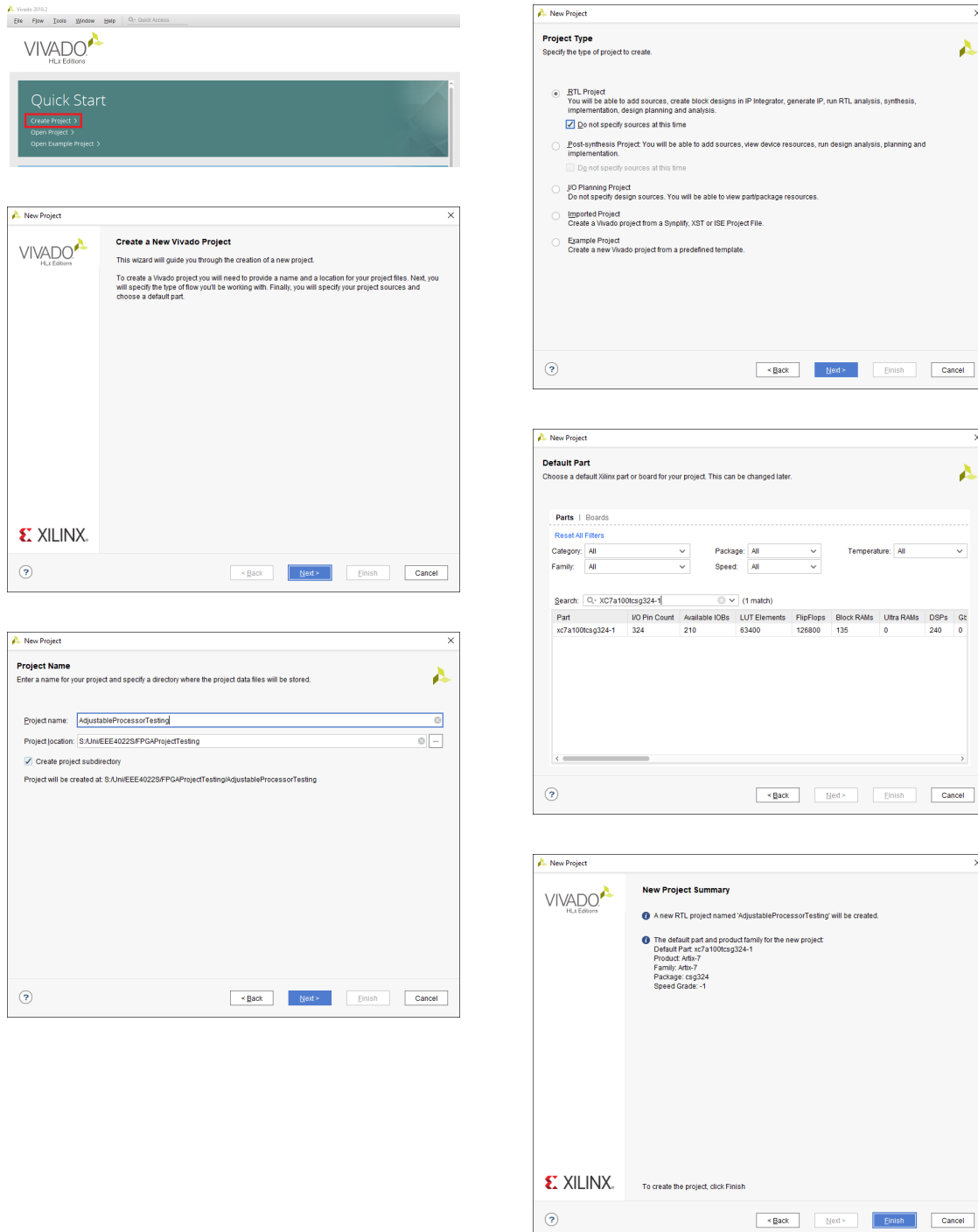


Figure E.1: The process for creating a new project in Vivado Design Suite

F MATLAB Code

F.1 Justification Experiment

```
num1 = [ 12.35 589 0 82];
num2 = [-5.25 0.8 8950 -0.09];

add = num1 + num2;
mul = num1 .* num2;
div = num1 ./ num2;

single(add)
single(mul)
single(div)
```

F.2 Heterodyne Experiment

```
clear all;
%close all;
clc;

% AD9162 DAC used for manipulating carrier
% http://www.analog.com/en/analog-dialogue/articles/new-rf-dac-broadens-sdr-horizon.html
% http://www.analog.com/media/en/technical-documentation/data-sheets/AD9161-9162.pdf

% Define parameters
Fs = 250E3;           % samples per second:
StopTime = 0.0025;    % seconds
Ac = 2^1;             % Amplitude of carrier
Fc = 20E3;            % Frequency of Carrier

Ax = 2^1022;          % Amplitude of x(t)
Fx = 2000;            % Frequency of x(t)

dt = 1/Fs;            % seconds per sample
t = (0:dt:StopTime-dt)'; % seconds

NumTaps = Fc/2;

% Generate x(t)
X = Ax*sin(2*pi*Fx*t);
X8 = fp8(Ax*sin(2*pi*Fx*t));
X16 = fp16(Ax*sin(2*pi*Fx*t));
X32 = single(Ax*sin(2*pi*Fx*t));

%% Generate LO
C = Ac*sin(2*pi*Fc*t);
C8 = fp8(Ac*sin(2*pi*Fc*t));
C16 = fp16(Ac*sin(2*pi*Fc*t));
C32 = single(Ac*sin(2*pi*Fc*t));

%% Multiply with LO
Y = X .* C;
Y8 = fp8(double(X8) .* double(C8));
Y16 = fp16(double(X16) .* double(C16));
Y32 = X32 .* C32;

%% Multiply with LO to get back to X(t)
O = Y .* C;
O8 = fp8(double(Y8) .* double(C8));
O16 = fp16(double(Y16) .* double(C16));
O32 = Y32 .* C32;

%% Pass through LPF
B = 1/NumTaps*ones(NumTaps,1);
out = filter(B,1,O);
```

```

out8 = filter(B,1,double(O8));
out16 = filter(B,1,double(O16));
out32 = filter(B,1,double(O32));
%% Plots for comparison - Carriers
% figure(1)
% subplot(3,1,1)
% plot(t, C)
% title('Carrier versus Time');

% subplot(3,1,2)
% plot(t, double(fp8(C)));
% title('Carrier @ 8 bit FP');

% subplot(3,1,3);
% plot(t, double(fp16(C)));
% title('Carrier @ 16 bit FP');

%% Plots for comparison - full precision
%figure(1)
% subplot(4,1,1)
% plot(t, C)
% title('Carrier versus Time');

% subplot(4,1,2)
% plot(t, X)
% title('Data versus Time');

% subplot(4,1,3);
% plot(t, Y)
% title('Modulated signal versus Time');

% subplot(4,1,4);
% plot(t, out)
% title('Recovered versus Time');

%% Visual comparison

%figure(2)
%plot(X) %Plot signal s1
%hold on
%plot(out, 'r'); %Delay signal s2 by delay in order to align them
%hold off
%corrcoef(X, [delay+1:length(out)+delay])
%legend("Original Signal", "Recovered Signal")

out = (out .* 250);
sub = max(out);
out = out - sub/2;

out8 = (out8 .* 250);
sub8 = max(out8);
out8 = out8 - sub8/2;

out16 = (out16 .* 250);
sub16 = max(out16);
out16 = out16 - sub16/2;

out32 = (out32 .* 250);
sub32 = max(out32);
out32 = out32 - sub32/2;

%corrcoef(X', out')
%X1=xcorr(X,out); %compute cross-correlation between vectors s1 and s2
%[m,d]=max(X1); %find value and index of maximum value of cross-correlation amplitude
%delay=d-max(length(X),length(out)); %shift index d, as length(X1)=2*N-1; where N is the length of the signals
%plot(X, '-s') %Plot signal s1
%hold on
%plot([delay+1:length(out)+delay],out, 'r'); %Delay signal s2 by delay in order to align them
%hold off
%corrcoef(X, [delay+1:length(out)+delay])
%legend("Original Signal", "Recovered Signal")

```

```

corrcoef(out, out32);
corrcoef(out, out8);
corrcoef(out, out16);

%figure(3)
%hold on
%plot(out, '-s')
%plot(out8, '-o')
%plot(out16, '-*')
%plot(out32, '-r')
%legend("64 bits", "32 bits", "16 bits", "8 bits")

maxxs = [double(max(X8)) double(max(X16)) double(max(X32)) double(max(X))]
maxys = [double(max(Y8)) double(max(Y16)) double(max(Y32)) max(Y)]
maxos = [double(max(O8)) double(max(O16)) double(max(O32)) max(O)]
maxouts = [max(out8) max(out16) max(out32) max(out)]

```

G IP Comparison Results

G.1 All Results Recorded

Table XLIII: The results for comparisons between IP and parameterized implementations

Module	Word Size	WNS (ns)	Power (W)	Dynamic										Static		LUT	FFs	BRAM	URAM	DSP	First result	Last result
				Clocks	Signals	Logic	BRAM	DSP	IO	Power	%											
Addition	16	4.632	0.106	0.001	13.00%	0.001	13.00%	0.002	21.00%	0	0.00%	0.004	43.00%	0.097	92.00%	236	198	0.5	0	0	35	183
	16 (IP)	6	0.138	0.002	6.00%	0.001	3.00%	0.002	5.00%	0.002	5.00%	0.031	76.00%	0.097	70.00%	134	345	0.5	0	2	29	181
Multiply	16	4.116	0.104	0.001	18.00%	0.001	15.00%	0.002	28.00%	0.001	15.00%	0.001	10.00%	0.097	94.00%	170	177	0.5	0	1	33	149
	16 (no DSP)	1.713	0.104	0.001	17.00%	0.001	21.00%	0.002	24.00%	0.002	28.00%	0	0.00%	0.097	93.00%	275	199	0.5	0	0	33	149
Division	16 (IP)	6.43	0.141	0.001	3	0.001	3	0.001	2	0.002	5	0.001	2	0.037	85	99	208	0.5	0	1	19	121
	16	4.509	0.107	0.003	28.00%	0.002	17.00%	0.002	16.00%	0.002	20.00%	0	0.00%	0.097	91.00%	278	247	0.5	0	0	129	437
	16 (IP)	5.803	0.139	0.003	7.00%	0.003	8.00%	0.003	7.00%	0.002	5.00%	0	0.00%	0.097	70.00%	270	478	0.5	0	0	37	229

H SWAP of Modules

H.1 Power and Timing

Table XLIV: Timing and Power Metrics for Jon Dawon's Veilog modules at Varying Precision

Module	Word Size	WNS (ns)	Total power (W)	Dynamic						Static					
				Clocks		Signals		Logic		DSP		IO		Power (W)	%
Addition	8	4.848	0.1	0.001	26.00%	<0.001	16.00%	0.001	19.00%	0	0	0.001	39.00%	0.097	97.00%
	16	4.148	0.101	0.001	35.00%	0.00%	14.00%	0.001	17.00%	0	0	0.001	34.00%	0.097	96.00%
	20	4.67	0.1	0.001	37.00%	0.001	26.00%	0.001	28.00%	0	0	<0.001	9.00%	0.097	97.00%
	24	4.49	0.102	0.001	30	0.001	16	0.001	18	0	0	0.002	36	0.097	95.00%
Multiply	32	4.023	0.102	0.001	23.00%	0.001	24.00%	0.001	27.00%	0	0	0.001	26.00%	0.097	96
	40	3.98	0.102	0.001	30.00%	0.001	24.00%	0.001	27.00%	0	0	0.001	19.00%	0.097	95.00%
	8	4.624	0.099	0.001	37.00%	<0.001	22.00%	0.001	31.00%	0	0	<0.001	10.00%	0.097	98.00%
	16	4.456	0.1	0.001	29.00%	0.001	18.00%	0.001	22.00%	<0.001	15	0.001	16.00%	0.097	97.00%
Division	20	4.292	0.1	0.001	34.00%	0.001	18.00%	0.001	22.00%	<0.001	11	<0.001	15.00%	0.097	97.00%
	24	4.286	0.1	0.001	31.00%	0.001	19.00%	0.001	24.00%	<0.001	10	0.001	16.00%	0.097	97.00%
	32	3.341	0.101	0.001	32.00%	0.001	21.00%	0.001	26.00%	<0.001	9	<0.001	12.00%	0.097	96.00%
	40	0.714	0.102	0.001	23.00%	0.001	22.00%	0.001	26.00%	0.001	20	0.001	9.00%	0.097	95.00%
Division	8	4.84	0.1	0.001	41.00%	0.001	24.00%	0.001	31.00%	0	0	<0.001	4.00%	0.097	97.00%
	16	4.631	0.102	0.002	32.00%	0.001	24.00%	0.001	31.00%	0	0	0.001	13.00%	0.097	95.00%
	20	4.062	0.102	0.001	30.00%	0.001	24.00%	0.001	29.00%	0	0	0.001	17.00%	0.097	96.00%
	24	4.515	0.102	0.002	31.00%	0.001	23.00%	0.001	28.00%	0	0	0.001	18.00%	0.097	95.00%
Division	32	4.229	0.102	0.001	25.00%	0.001	26.00%	0.002	32.00%	0	0	0.001	17.00%	0.097	95.00%
	40	N/A													

H.2 Resource Use

Table XLV: Physical Resource Use for Jon Dawon's Verilog modules at Varying Precision

Module	Word Size	LUT	FF	BRAM	URAM	DSP
Addition	8	117	72	0	0	0
	16	158	102	0	0	0
	20	179	117	0	0	0
	24	197	132	0	0	0
	32	258	163	0	0	0
	40	274	195	0	0	0
Multiply	8	103	67	0	0	0
	16	94	71	0	0	1
	20	115	77	0	0	1
	24	122	83	0	0	1
	32	146	110	0	0	2
	40	210	165	0	0	4
Division	8	175	97	0	0	0
	16	226	131	0	0	0
	20	252	146	0	0	0
	24	269	161	0	0	0
	32	324	192	0	0	0
	40	N/A				

I Embedded System Use Case

I.1 CPU Info

```

processor      : 1
model name    : ARMv7 Processor rev 4 (v7l)
BogoMIPS      : 38.40
Features      : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant    : 0x0
CPU part       : 0xd03
CPU revision   : 4

processor      : 2
model name    : ARMv7 Processor rev 4 (v7l)
BogoMIPS      : 38.40
Features      : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant    : 0x0
CPU part       : 0xd03
CPU revision   : 4

```

```

processor      : 3
model name    : ARMv7 Processor rev 4 (v7l)
BogoMIPS     : 38.40
Features      : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant   : 0x0
CPU part      : 0xd03
CPU revision  : 4

Hardware      : BCM2835
Revision     : a22082
Serial       : 00000000b9250425

```

I.2 Batch File for Cross-Compilation and Transfer

```

@echo off
rem see https://www.slideshare.net/tomoaki0705/cvim-half-precision-floating-point
rem see https://embeddedartistry.com/blog/2017/10/9/r1q7pksku2q3gww9rpqef0dnskphtc
rem This batch file compiles a c or cpp program for the raspberry pi on a windwos machine
rem RPi cross-compilation tools are required
rem -mfp16-format=ieee tells the compiler to use IEEE-754 representations for half-precision floating point
rem -mfpu=vfpv4 tells the compiler to use the FPU is available
rem vfpv4 adds support for half-precision

rem Hawdware floating point
rem -mfpu=vfpv4
rem vfpv3_fp16
rem neon-fp16

rem single precision
rem fpv4-sp

rem hw fp
rem vfpv3

rem Software FP
rem -fpu=softvfp

rem 16 bit floating point
rem vfpv3_fp16

rem another one?
rem -mfpu=neon-fp-armv8

rem set /p file="Enter file to compile: "
@echo on
arm-linux-gnueabi-g++ -g -gdb test.c Timer.cpp -o output -mfp16-format=ieee -mfpu=vfpv3xd-fp16
pscp -pw <password> output pi@10.3.141.1:

```

I.3 The C Code Used

```

@echo off
rem see https://www.slideshare.net/tomoaki0705/cvim-half-precision-floating-point
rem see https://embeddedartistry.com/blog/2017/10/9/r1q7pksku2q3gww9rpqef0dnskphtc
rem This batch file compiles a c or cpp program for the raspberry pi on a windwos machine
rem RPi cross-compilation tools are required
rem -mfp16-format=ieee tells the compiler to use IEEE-754 representations for half-precision floating point
rem -mfpu=vfpv4 tells the compiler to use the FPU is available
rem vfpv4 adds support for half-precision

rem Hawdware floating point
rem -mfpu=vfpv4
rem vfpv3_fp16
rem neon-fp16

```

```

rem single precision
rem fpv4-sp

rem hw fp
rem vfpv3

rem Software FP
rem -fpu=softvfp

rem 16 bit floating point
rem vfpv3_fp16

rem another one?
rem -mfpu=neon-fp-armv8

rem set /p file="Enter file to compile: "
@echo on
arm-linux-gnueabi-g++ -ggdb test.c Timer.cpp -o output -mfp16-format=ieee -mfpu=vfpv3xd-fp16
pscp -pw <password> output pi@10.3.141.1:

```

I.4 Timing Module

```

//=====
// Copyright (C) John-Philip Taylor
// tyljoh010@myuct.ac.za
//
// This file is part of the EEE4084F Course
//
// This file is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <http://www.gnu.org/licenses/>
//=====

#ifndef Timer_h
#define Timer_h
//-----

#include <stdio.h>
//-----

void tic(); // Start timer
double toc(); // Get time interval
//-----

#endif
//-----

```

```

//=====
// Copyright (C) John-Philip Taylor
// tyljoh010@myuct.ac.za
//
// This file is part of the EEE4084F Course
//
// This file is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

```

```

// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program. If not, see <http://www.gnu.org/licenses/>
//=====

#include "Timer.h"
//-----

#ifdef __WIN32__
#include <windows.h>
#else
#include <time.h>
#endif
//-----

static double Frequency;
static double Start;
//-----

void tic(){
    static bool First = true;

#ifdef __WIN32__
    LARGE_INTEGER Time;
    if(First){
        QueryPerformanceFrequency(&Time);
        Frequency = Time.QuadPart;
        printf("Clock resolution: %lg ns\n", 1e9/Frequency);
        First = false;
    }
    QueryPerformanceCounter(&Time);
    Start = (double)Time.QuadPart / Frequency;
#else
    // This is an alternative, using <sys/time.h>. It is not as accurate though.
    // timeval Time;
    // gettimeofday(&Time, 0);
    // Start = (double)(Time.tv_sec) + (double)(Time.tv_usec)*1e-6;

    timespec Time;
    if(First){
        clock_getres(CLOCK_MONOTONIC, &Time);
        Start = (double)(Time.tv_sec) + (double)(Time.tv_nsec)*1e-9;
        printf("Clock resolution: %lg ns\n", Start*1e9);
        First = false;
    }
    clock_gettime(CLOCK_MONOTONIC, &Time);
    Start = (double)(Time.tv_sec) + (double)(Time.tv_nsec)*1e-9;
#endif
}
//-----

double toc(){
    double End;

#ifdef __WIN32__
    LARGE_INTEGER Time;
    QueryPerformanceCounter(&Time);
    End = (double)Time.QuadPart / Frequency;
    return End - Start;
#else
    //timeval Time;
    //gettimeofday(&Time, 0);
    //End = (double)(Time.tv_sec) + (double)(Time.tv_usec)*1e-6;

    timespec Time;
    clock_gettime(CLOCK_MONOTONIC, &Time);
    End = (double)(Time.tv_sec) + (double)(Time.tv_nsec)*1e-9;
    return End - Start;
#endif
}
//-----

```

J Ethics Clearance

Project Title	Adjustable Precision and Performance Cost Analysis	08/13/2018
	by Keegan Crankshaw in EBE Electrical Submissions Undergraduate	id. 10909785
	crnkee002@myuct.ac.za	

Original submission	08/13/2018
----------------------------	------------

Cover Letter	The project is a cost-benefit analysis for changing the word size in particular hardware designs. There are no foreseen ethical issues.
Application Checklist	Read the EBE Ethics in Research Handbook before completing this application
Researcher(s)	Keegan Crankshaw
Department	Electrical Engineering
E-mail	crnkee002@myuct.ac.za
Status of Applicant	Student
Degree Being Studied (For Students Only)	Bsc
Name of Supervisor (For Students Only)	Simon Winberg
Review Track	Normal
Motivation for an Expedited Review	n/a
Signature	Ethics_Signature_form_CRNKEE002.pdf
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
--------------------------------------	-----------

Research Proposal

[Electrical_and_Computer_Engineering_BSc_Project_Topics.pdf](#)

Question 2.1: Discrimination	n/a
---------------------------------	-----

Question 2.2: Participation of socially or physically vulnerable people	n/a
----------------------------------------------------------------------------------	-----

Question 2.3: Informed consent	n/a
-----------------------------------	-----

Question 2.4: Confidentiality	n/a
----------------------------------	-----

Question 2.5: Anonymity	n/a
----------------------------	-----

Question 2.6: Risks of physical, psychological or social harm	n/a
------------------------------------------------------------------------	-----

Question 2.7: Payments and giving of gifts	n/a
--------------------------------------------------	-----

Interview Schedule	n/a
--------------------	-----

Consent Form	n/a
--------------	-----

Additional Comments	n/a
---------------------	-----

Question 3.1: Community participation	n/a
---------------------------------------------	-----

Question 3.2: Termination of economic or social support	n/a
------------------------------------------------------------------	-----

Question 3.3: n/a
Provision of sub
standard services

Additional Comments n/a

Question 4.1: n/a
Conflicts of interest

Question 4.2: n/a
Sharing of
information

Question 4.3: n/a
Conflict of interest
with other research

Additional Comments n/a

K ELO Tracking Form

EEE4022S/F Final Year Project
Supervision ELO Tracking Form 2018

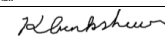
Student Name: Keegan Crankshaw

Student Number: CRNKEE002

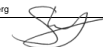
State whether a DP / DPR is awarded: **DP**

	Supervisor Comment	Supervisor Comment	Supervisor Comment
ELO 4: Investigations, experiments and data analysis Demonstrate competence to plan and conduct investigations and experiments. The balance of investigation and experiment should be appropriate to the discipline. Research methodology to be applied in research or investigation where the student engages with selected knowledge in the research literature of the discipline. Note: An investigation differs from a design in that the objective is to produce knowledge and understanding of a phenomenon and a recommended course of action rather than specifying how an artifact could be produced. ACTION REQUIRED FROM THE STUDENT: The Student must engage in structured research relating to the topic at hand in order to gain insight into the chosen field. Such research includes information on word sizes, the relation between word size, accuracy and speed of execution, HDL and comparison of hardware and software in terms of execution speed. This knowledge must be combined to form a literature review. Using knowledge obtained from the research aspect of the project, as well as knowledge and experience gained through the course of the undergraduate degree, the student must design and implement experiments to investigate the hypothesis.	The student did good planning of the methodology for this project, at a suitably early stage so as to direct the design and tests to be done. He has followed an effective approach. Thorough planning was done early on and the objectives and various processing operations and testing approaches were discussed in detail and effectively planned, culminating in a good sequence of test experiments to do with a variety of calculations for testing the efficiency and resource utilization of the designs posed.		
ELO 6: Professional and technical communication Demonstrate competence to communicate effectively, both orally and in writing, with engineering audiences and the community at large. This course evaluates the long report component of this outcome at exit level. Material to be communicated is in an academic or simulated professional context. Audiences range from engineering peers, management and lay persons, using appropriate academic or professional discourse. Written reports range from short (500-1000 word plus tables diagrams) to long (10 000 to 15 000 words plus tables, diagrams and appendices), covering material at exit-level. Methods of providing information include the conventional methods of the discipline, for example engineering drawings, as well as subject-specific methods. ACTION REQUIRED FROM THE STUDENT: The Student must produce a project report (or thesis) of high standard. Each aspect of the project must be well documented. The required diagrams, charts, etc must be included to support the body of text in the report. Professional language and correct tone must be used to convey all aspects of the project, in a manner that is not exclusive to audiences. The report should be correctly referenced using the IEEE referencing method. The primary aspects of the project must also be collated into a PowerPoint presentation as well as a poster as a means of presentation both to peers and wider communities at the department's Open Day.	The student has shown good professionalism, both in his written and oral communication. In terms of draft report chapters, he has a good understanding of what is expected and has made good progress thus far. The student has used a good choice of terms, wording and writing style for the draft documentation shown for inclusion into the report. Good understanding of use of visualization methods. Based on the progress so far, it looks like the student will finish the report on time.		
ELO 8: Individual, team and multidisciplinary working Demonstrate competence to work effectively as an individual, in teams and in multidisciplinary environments. This course evaluates the individual working component of this learning outcome at exit level. ACTION REQUIRED FROM THE STUDENT: The Student must be able to work individually on the project, reaching out to peers and supervisors if assistance is required. However, assistance given must not hinder the student's ability to complete the project individually, that is to say assistance given should not seek to complete aspects of the project, rather guide the student in the correct direction.	The project incorporates aspects of multidisciplinary in terms of drawing of the range of expertise in maths, standards and FPGA/HDL design skills which are key to this project. The student has refined the project objectives and methodology, and is maintaining a good focus on these. The student has participated adequately in works involving others.		
ELO 9: Independent learning ability Demonstrate competence to engage in independent learning through well developed learning skills. Operate independently in complex, ill-defined contexts requiring personal responsibility and initiative, accurately self-evaluate and take responsibility for learning requirements; be aware of social and ethical implications of applying knowledge in particular contexts. ACTION REQUIRED FROM THE STUDENT: The Student will execute the project in an ethically sound manner. The Student will spend the time required independently researching information as well as developing the skills required to implement the given aspects of the project.	The student shows good competence in working independently on complex tasks. The student has demonstrated ability to take responsibility for learning what is needed and to use the tools to develop the required system.		

Student Name Keegan Crankshaw
Student Signatures



Internal Examiner Name Simon Winberg
Internal Examiner Signatures



Designation Final Year Student of Engineering
Dates 28-Aug-18

Designation Internal Examiner
Dates 28-Aug-18