Contents

1 Introduction

1.1 Objectives

The objective of this study is to investigate how various implementations of IEEE-754 floating point numbers affect speed of execution, size of implementation, power requirements, and accuracy. 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 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. Analyze 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: A Table Showing the 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 has changed drastically. As more data is required to be processed, the way in which it need 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 compatibility has meant that hardware changes have not come as quickly as perhaps thought initially possible. The x86 architecture, for example, discussed briefly in this paper, has been the underlying architecture of most processors since 1985, and has imposed limitations set by needed backwards-compatibility.

It can be argued that edge computing, particularly with the rise of "big data", will become vital in day-to-day life. Driverless 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.

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. 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 that has been started 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 utilize 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
- Parameterize 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 (Vivado 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 +, -, *, /) 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.

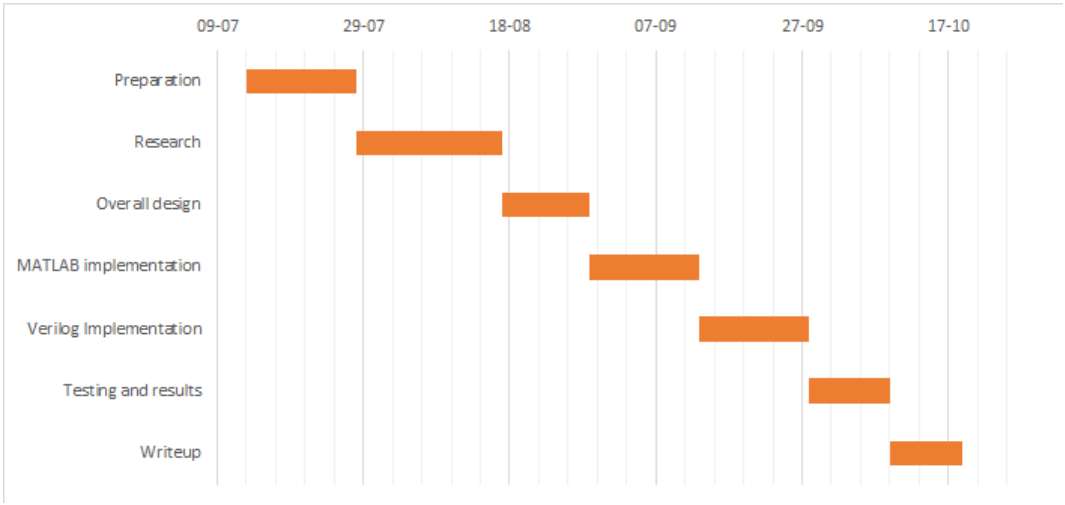


Figure 1.1: A Gantt chart showing the expected time to be spent on each aspect of the project

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 a previous paper that investigated the relationship between bit width, speed, and resource use.

2.1 Word Size

A "word" in computing has various meanings, which can depend on context. Generally, word size refers to the size of the general purpose registers used by the CPU. **That being said, when an architecture is extended to increase the size of the general purpose registers, the word size for that processor remains at it's original size.** For example, AMD's Opteron processor, the first 64-bit architecture, developed in 2003 [3], while being a 64-bit architecture, can still 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]

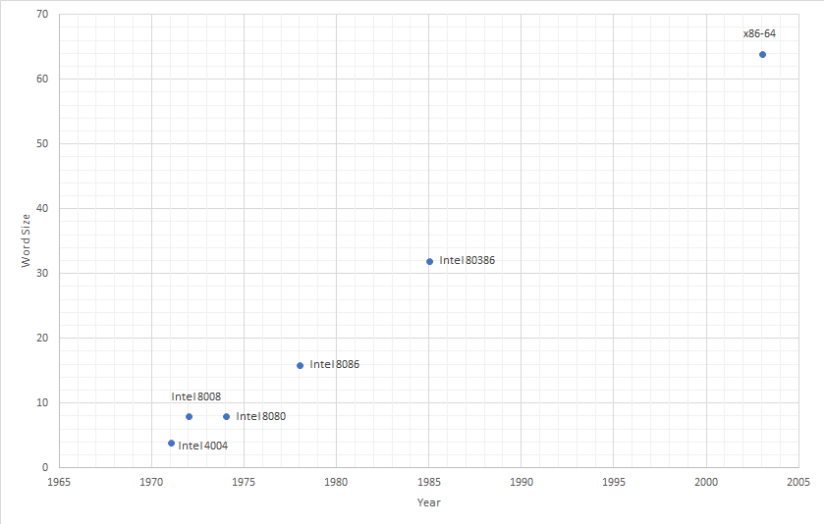


Figure 2.1: A graph showing the word size of commercially available processors over the years [10]

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 one time in parallel hardware. [13] Another advantage of longer instructions is the ability to handle more precision representations of data in less 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 word size

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. More time and complexity is required to design the hardware. As an example, buses 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 value, 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 to be 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: A table comparing the three representations for three numbers

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 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]

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

$$-2^n \leq x \leq +2^n - \frac{1}{2}^m \tag{3}$$

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 $2^{-8} = \frac{1}{256} = 3.90625x10^{-3}$. 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. While greater accuracy and range can be increased by In order to address this issue and provide higher accuracy possible, floating point representation is used.

2.2.3 Floating Point and the IEEE-754 Representation

For higher 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 (the bias is a constant added to the exponent to ensure positive only representations), 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:

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

The standard specifies a few levels of precision, for example:

- 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

Normalization

In addition to the exponent being biased, there is one other caveat: the significand must be normalized. This means that the digit to the left of the decimal point in the significand must be 1. For example, $0.00101_2 * 2^4$ normalized would be expressed as $1.01_2 * 2^2$. Because all numbers must be normalized, 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 LSD 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 DFP arithmetic, but not for BFP 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 (non 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 denormalized, 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 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 zero by zero or subtracting infinity from 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.

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, however, includes normalization and rounding.

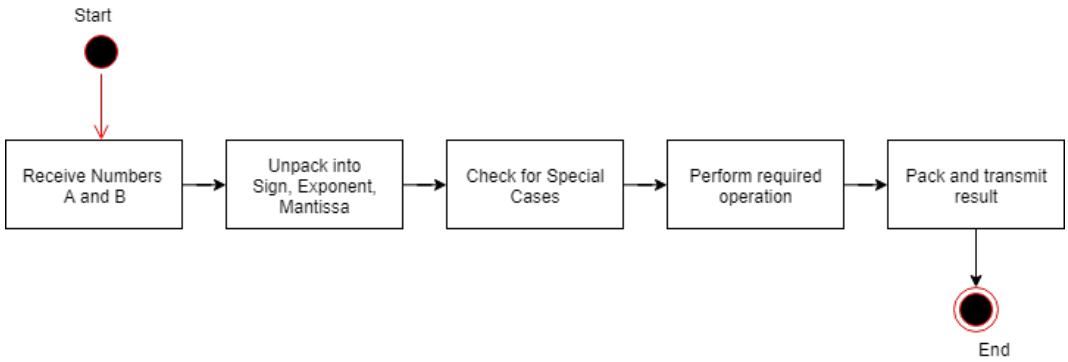


Figure 2.3: The stages involved when using IEEE-754 Floating point implementation

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]

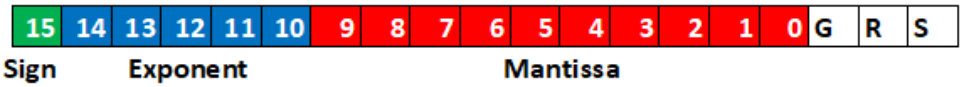


Figure 2.4: Showing how guard (G), round (R) and sticky (S) bits are appended to the floating point representation during calculation

2.2.6 Conclusion

Floating point representations require more resources than fixed point, primarily due to pre- and post-normalization 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 this, floating point implementations are required for greater dynamic range which may be critical to the task at hand. Thus optimization of floating-point bit-width important. Optimizing word size enables more parallelism, as there are more FPGA resources to be

utilized, or, if the degree of parallelization 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 generate more and more powerful systems capable of higher throughput, in recent years there has been a focus on a metric known as SWAP - size, weight and power.

There is no absolute optimal selection between these metrics - the ones to optimize 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 optimize 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 more and more 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 minimizing 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, 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 occupy the system. In terms of power, Summit requires 13 megawatts of power. [26] [27]

While these metrics are extremes, the point is to show that SWAP should be decided on a per-case basis, rather than attempting to optimize a metric before considering implications in the 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. Size does not only refer to physical size, but may also refer to process size, or, as it does in this investigation, the size of the implemented design on an FPGA, done 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 means of measurement for power include a clamp meter, USB meter, or a specialized 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.

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 5.2.

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, relating 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. [29] 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 program the FPGA.

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". [29] 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. [30] Amazon Web Services has had FPGAs available in EC2 F1 instances since 2017. [31] 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* [32].

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 [32]

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 2.5 shows the interaction of these components.

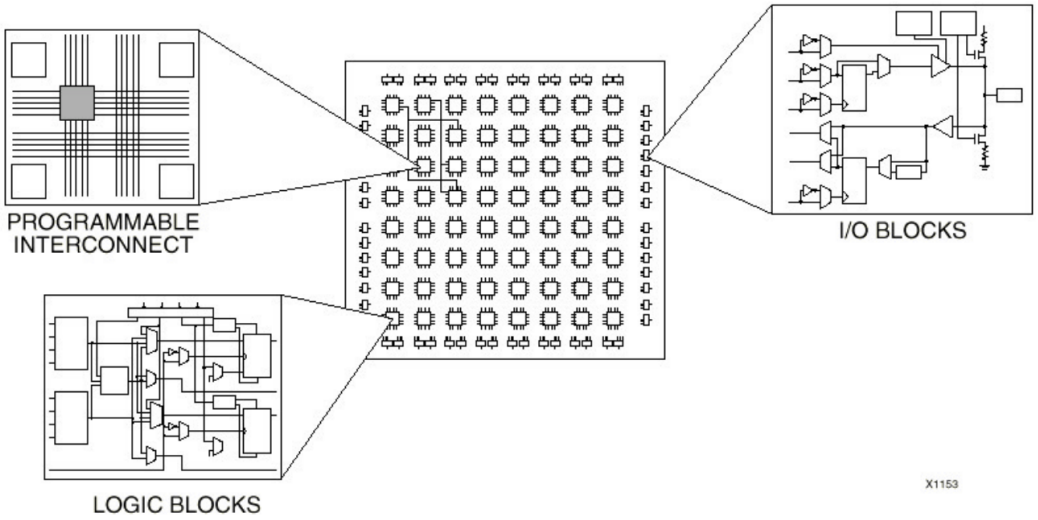


Figure 2.5: Simplified FPGA architecture

The Xilinx Artix 7 also contains DSP slices, specifically DSP48E1. There are 48-bit DSP units with the structure as seen in Figure 2.6. They are able of supporting the following functionality: multiply, multiply accumulate, multiply add, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bitwise 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. DSP units can be used explicitly or inferred. [33]

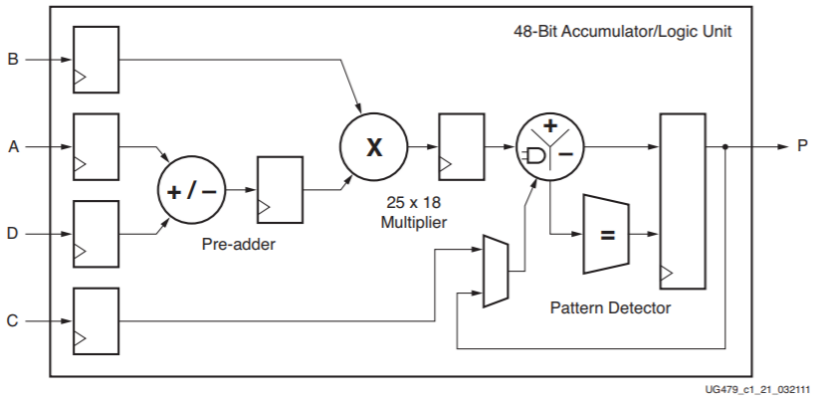


Figure 2.6: The architecture of a DSP slice [33]

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.

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. They can be summed up as shown in Table V below. Figure 2.7 compares flexibility and efficiency of the various platforms.

Table V: Advantages and Disadvantages of various Computing Platforms [34]

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		

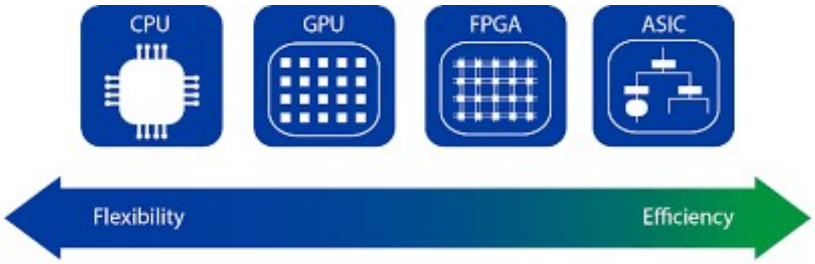


Figure 2.7: Comparison of various technologies in terms of flexibility and efficiency [35]

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 2.8 shows the speed of a specific type of hardware in comparison to how flexible the system is. In short, FPGAs and reconfigurable processors provide high flexibility and high MOPS (millions of operations per second) for the given power consumption.

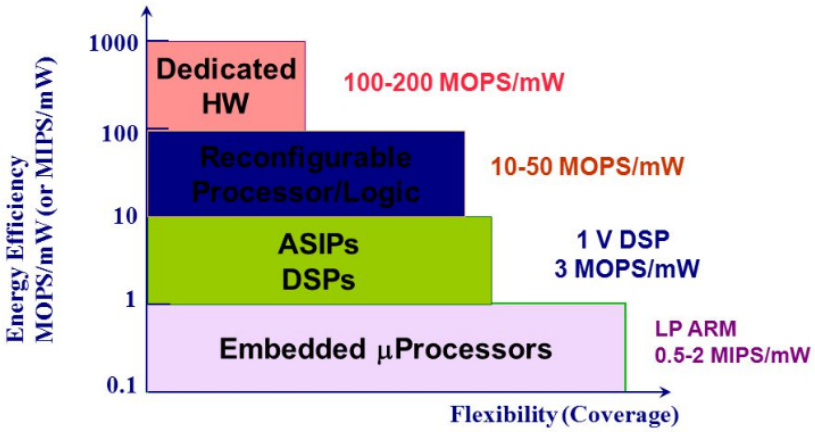


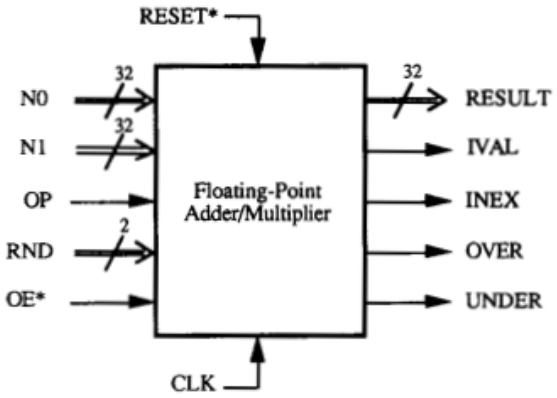
Figure 2.8: Comparison of energy efficiency to flexibility of the system [36]

2.5 Past Efforts at Implementing IEEE-754 on an FPGA

This section covers past attempts at optimizing 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. [37] 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 was then selected using a particular op-code. The circuit with inputs and outputs can be seen in Figure 2.9.



INPUTS:
- N0, N1 32-bit IEEE standard floating-point operands
- OP 1-bit operation code (0=add, 1=mult)
- RND: 2-bit control signal indicating rounding mode
- OE*: output enable for RESULT, see below)
- RESET*: system initialization signal
- CLK: system clock

OUTPUTS:
- RESULT: result of OP applied to N0, N1
- IVAL: indicates invalid operation
- INEX: indicates inexact operation
- OVER: indicates overflow
- UNDER: indicates underflow

Figure 2.9: High-level view of the Add/Multiply circuit implemented by Fagin et al [37]

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 adding the rounding stage of the IEEE-754 standard to the design required changes to the exponent and normalization 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 synthesized 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, but that 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 the design decision. 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 [38] did a quantitative analysis on FPGA custom computing machines (CCMs) focussing 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 reult 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 optimize 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 dision circuit had an added bottleneck in the recipricator (in this implemetnation, 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. [38]

	Adder/Subtractor	Multiplier	Divider
FG Function Generators	26%	36%	38%
Flip Flops	13%	13%	32%
Stages	3	3	5
Speed	9.3MHz	6.0MHz	5.9MHz

2.5.3 Louca, et al. (1996)

In 1996, Louca et al. [39] 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 Generator	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 above papers:

- 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 Optimizing Bit-Width in Custom Hardware Designs

One of the primary goals when designing hardware to to optimize 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 optimized by hand, the work and effort required to optimize larger designs can be difficult and time-consuming. As a result, optimizing bit widths is a commonly researched problem. This subsection presents two methods of bit width optimization and the results of the experimentation produced by the team.

2.6.1 MiniBit

MiniBit, developed by Lee, et al [40] 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 amd 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 then 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 optimization process. Precsion 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 (implementeed on a Virtex 4) by 20% and reduce latency by 12%.

2.6.2 The BitSize Tool

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

BitSize is a more advanced tool, providing details on parameters such as accuracy, dynamic range, area and speed. BitSize can also report on which representation (fixed or floating point) is more suited to the application given a set of constraints. 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 requires that a sample data set is used as input (dynamic analysis). 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 can result in speed increases of up to 12% and decrease resource usage considerably, allowing for features to be included in the design.

2.7 Previous Work Using Reduced Precision to Increase Throughput

This subsection covers a research paper written by Duben, et al [41].

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 computational results with silicon area, power, and operating frequency, something not available on commercial CPUs. The results were used in a comparitive 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 the 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
- 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 is still much more complicated and consumes significantly more time than programming CPUs or even GPUs
-

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

This section has covered literature relevant to the project at hand. The chapter started with discussion of how processing architecture 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.

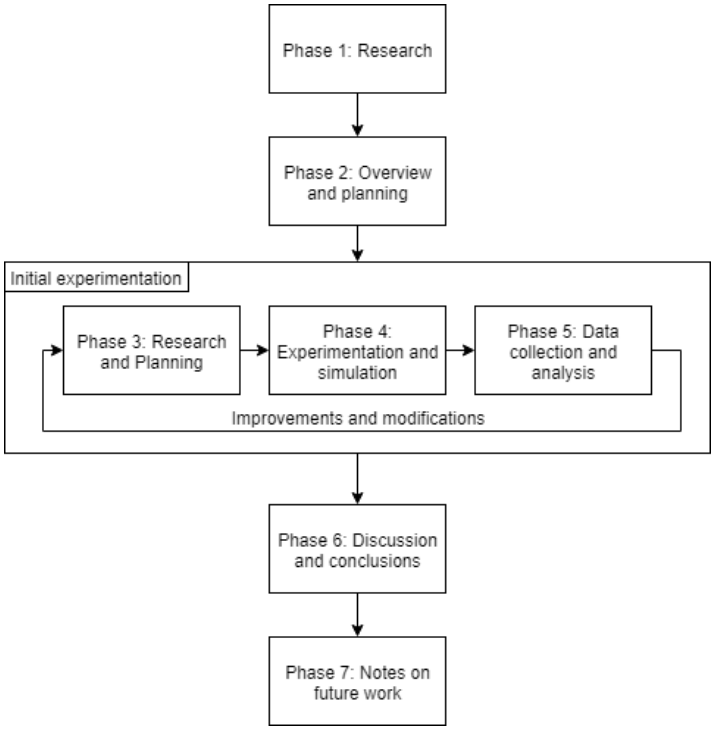


Figure 3.1: Phases of the project

3.1 Phase 1: Initial Research and Literature Review

In order to ensure meaningful experiments were conducted, reviews 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 optimization was investigated. Methods of reporting on size, weight and power of systems as well as precision and range of representations was also investigated. All of these reviews of literature are 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 section 4 and 5. This

initial set of experimentation involved running simulations, as well as 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 phases used in research are described in this section. The overall design process follows a Spiral Model, while each module to be designed and experimented followed the V-Model. The spiral model, initially developed by Barry Boehm in 1986. The traditional model follows the process as shown in Figure 3.2. The spiral model offers the following benefits[1]:

- Due to it's iterative nature, features can be developed in a systematic way
- The system can continuously be improved and iterated upon
- More functionality can be added at a later stage
- The spiral model aids in mitigating risk
- features are added in a systematic way
- Feedback is continual and it is easy to pivot/change if need be

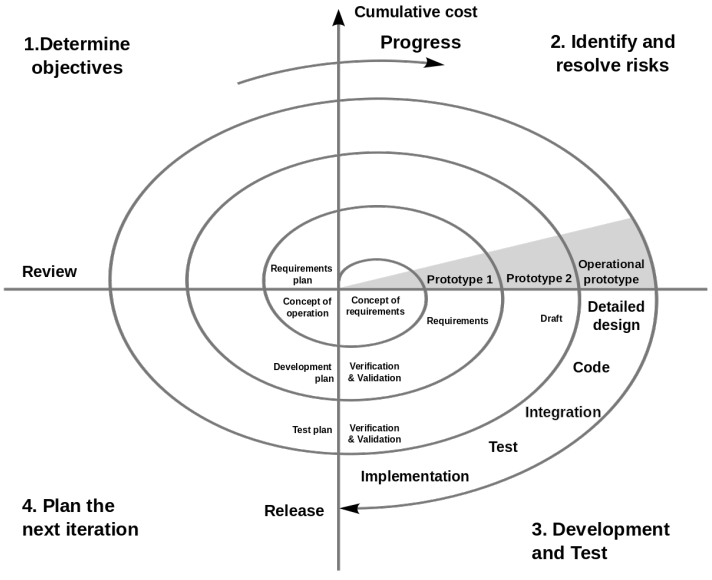


Figure 3.2: The Spiral Model [42]

[1]See https://www.guru99.com/what-is-spiral-model-when-to-use-advantages-disadvantages.html

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 for projects the following reasons:

- The V-Model allows for a sequential design flow where iterative processes are not possible or can be too time-consuming.
- The V-Model emphasizes testing and verification for the smaller modules and ensures completion and verification of a module before being used in a larger system. This helps reduce risk at a later stage in the project
- With the V-Model, components can be designed/implemented in parallel if need be, as each module is seen as it's own component, rather than new components being additions to a larger, pre-existing system.

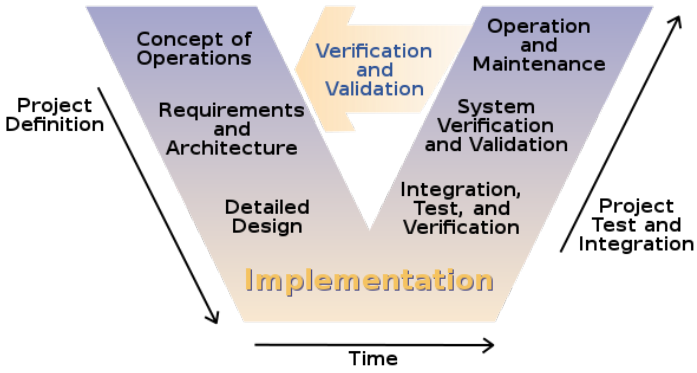


Figure 3.3: The V-Model [43]

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 parameterized 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 synthesized for varying bit widths. Size, weight and power metrics were recorded. Python scripts were used to ensure the correct results were being recorded. 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 synthesized 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 5.4, with results in Section 6.2.

Experiment 3 - Parameterization of Modules

The modules in Experiment 1 should be parameterized 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 Section 5.1, and results in Section 6.3 and 6.4 .

Experiment 4 - Comparison of Implementations

The parameterized 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 5.3, 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. If time permits, the use case must be developed to run on the FPGA using the operations developed in Experiments 1 and 3 and be run at varied word sizes. 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.

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 done. 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 experiemnts 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 requiremments

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. Finally, the section concludes by describing the exact ways in which the experiments will be conducted.

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.

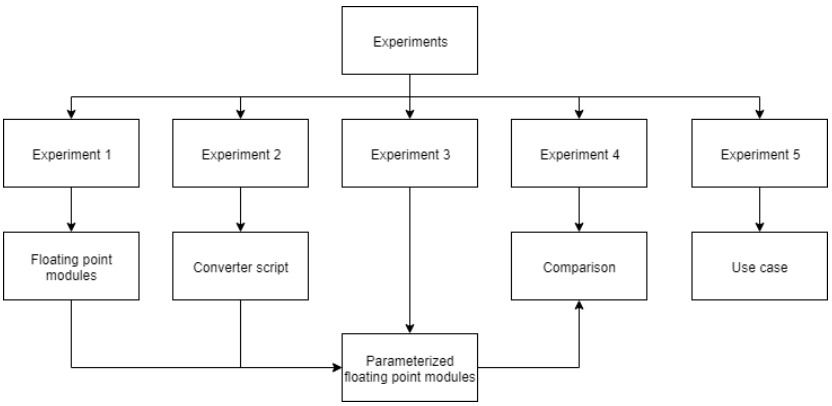


Figure 4.1: Break down of experiments

The requirements of each "experiment block" are as follows:

- Floating point modules
  The floating modules should implement the IEEE-754 standard. Careful consideration must be made when choosing implementations as they must be parameterizable (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.
- Parameterization and testing
  The modules as used in the first experiment should be parameterized. 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
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 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, more verbose than Verilog.
- Verilog
  Verilog is weakly typed, and follows a C-style syntax, making the language more familiar to more 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 the 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. [44]

Once the language is chosen, there are various options as a development environment. Xilinx Vivado Design Suite [45] is free to use (for students) and offers a lot of information relating

to synthesized designs.

4.2.4 SW02 and SWO3

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 [46], Octave [47], and Python [48] (with specific note of libraries SciPy [49] and NumPy [50]).

Table XI: 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 develop in. MATLAB is considered an important industry tool, which 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 inbetween these two options. It is similar to MATLAB in function, but is not as well optimized or supported. It is also free and open source, as is Python.

4.2.5 SW04 - Embedded System Use Case Development

The programming language chosen largely depends on embedded system chosen. C/C++ are quite popular, so this will be used in the comparison below. IEEE also lists Python as the most popular embedded programming language of 2018 [51], and as a result that will also be added to the comparison.

Python's execution speed is relatively slower to C, but it is better understood by most people will limited experience in the programming field, and it is easier to write Python. C, above and beyond it's faster execution speed, has a wealth of resources and a significant legacy in embedded systems development.

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 were made:

Table XII: 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
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 ma thematic 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 [52]. The algorithms are "optimized for space" [2] and are not the fastest methods for performing these operations.

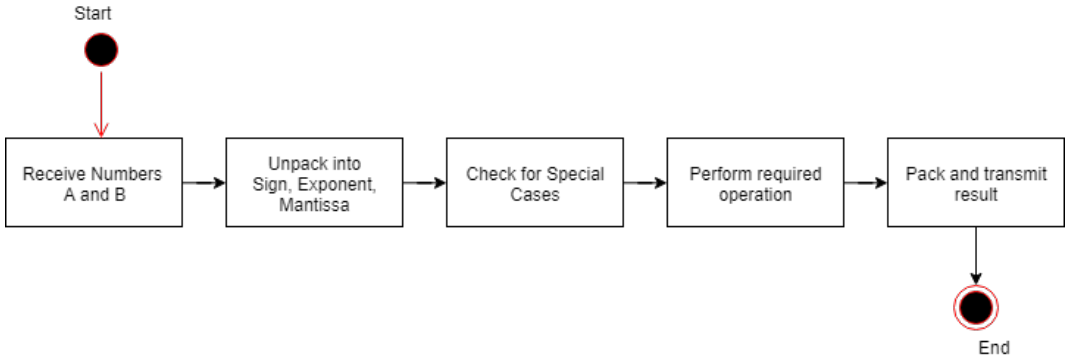


Figure 5.1: A high-level view for processing IEEE-754 operations

In all the basic operations, the process followed is akin to what is shown in Figure 5.1 above. In order to parameterize each module, the primary parameters were added. 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 mantissa in bits (starting at zero). The reson 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 XIII shows the values of these parameters for given word sizes.

[2] See https://github.com/dawsonjon/fpu/issues/12

Table XIII: 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 XIV along with a short description of the signal. Figure 5.2 shows a the block diagram view. The universality allows the same test bench to be used for each operator.

Table XIV: The signals used in the parameterized 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 Operation A
output	input_b_ack	Acknowledgement of Operation B

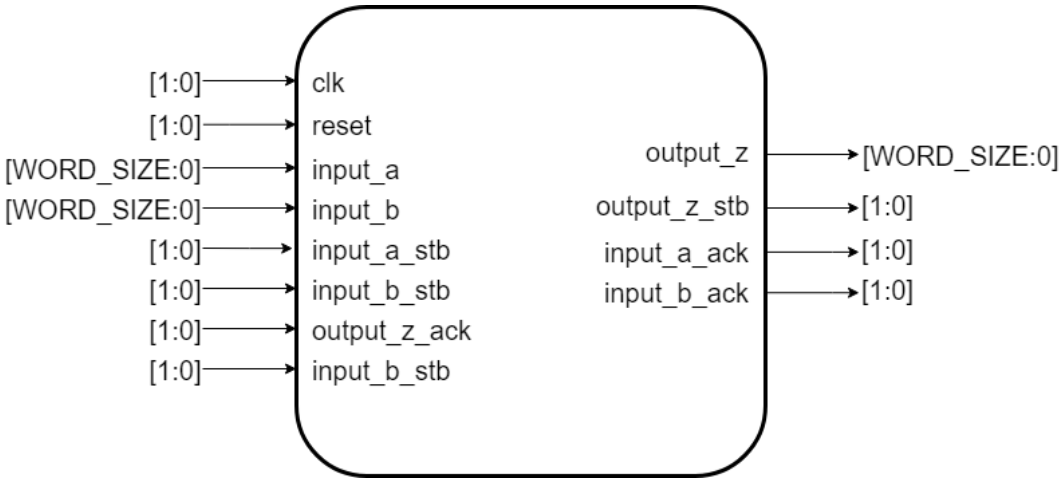


Figure 5.2: The IO signals in each mathematical operator

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

5.1.1 Addition

Addition was the first module that was to be parameterized. Due to the fact that IEEE 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 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.

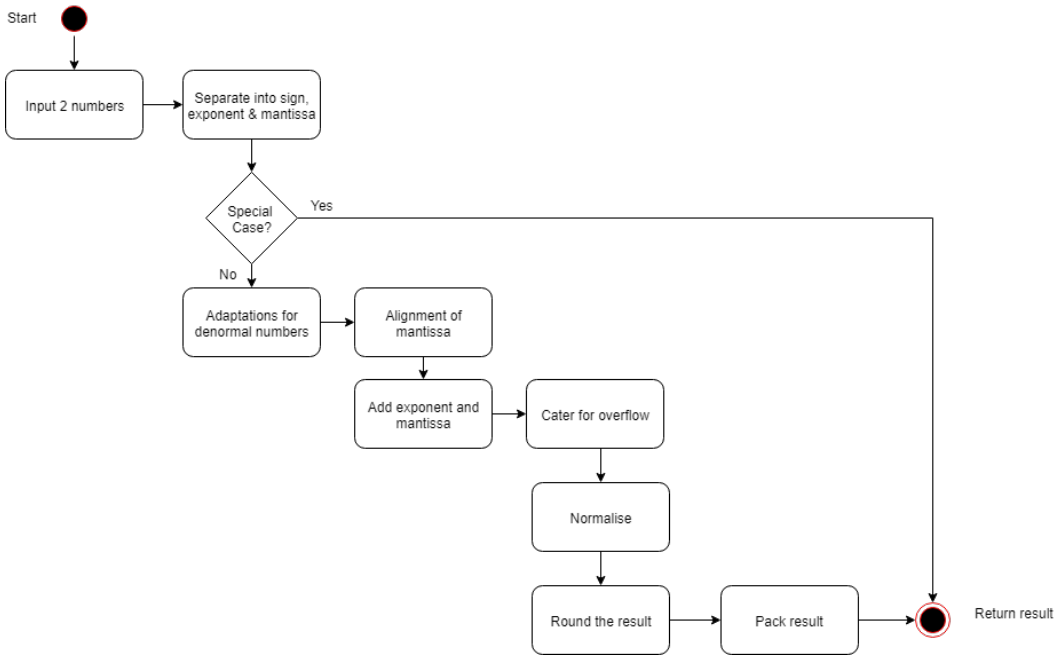


Figure 5.3: The states in Jon Dawson's FPU Addition module [52]

The final code used can be found in Appendix B.1.1.

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 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, put_z. Special cases considered in this module include multiplication of zero, nan and inf. The final code used can be found in Appendix B.1.2.

5.1.3 Division

The division algorithm used a restoring division algorithm. The 32-bit implementation in the original divider can be seen in Figure 5.4. 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. [53] Additional parameters defined included *DIV_MSB*, which was used to create a register big enough to shift *MANTISSA_MSB*,

as well as store overflow and guard, round and stick bits. The final code used can be found in Appendix B.1.3.

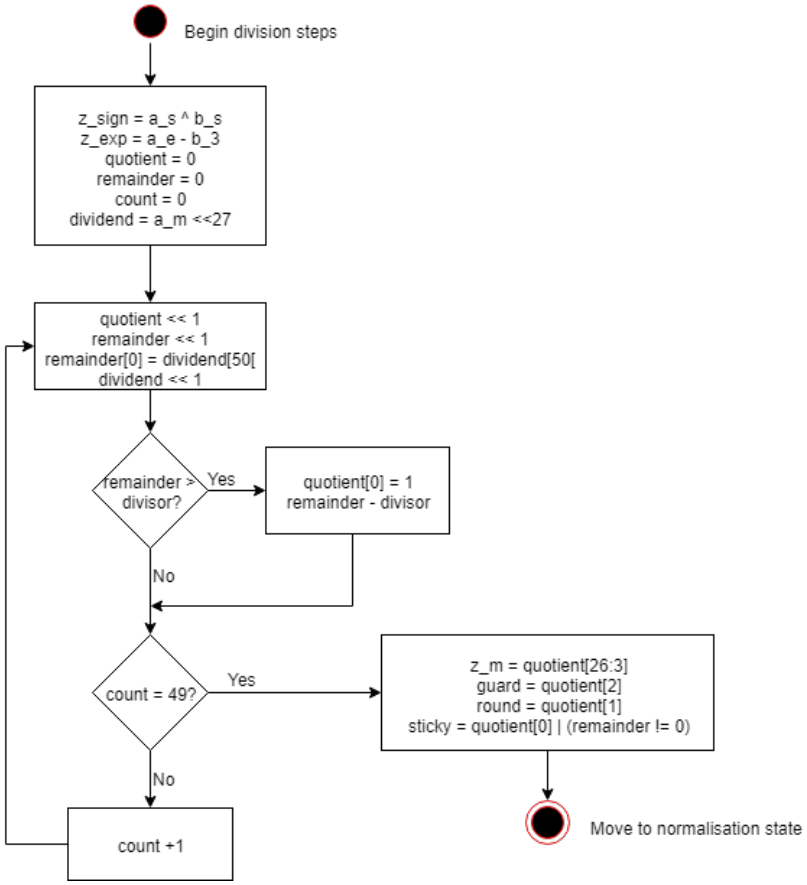


Figure 5.4: The restoring-division algorithm for 32-bit floating point numbers [52]

5.2 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

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, timing and power use for the given application.

5.2.1 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 (DRAM), Ultra RAM (URAM), and DSP units. These metrics will be recorded and compared for each of the experiments conducted.

5.2.2 Power Measurements

An example of a high-level power report as provided by Vivado is shown in Figure 5.5 below.

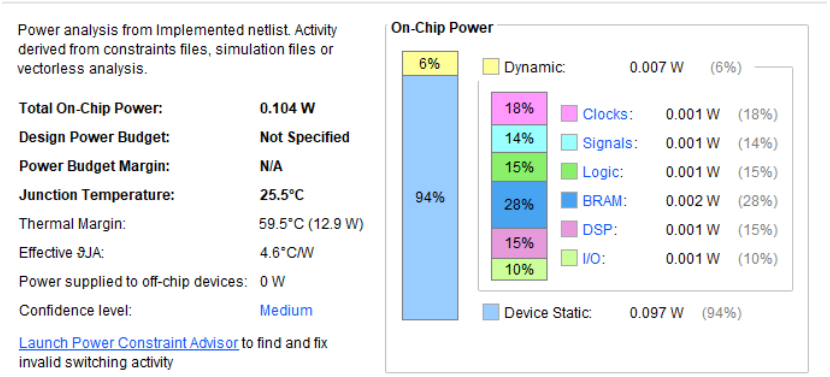


Figure 5.5: An example of the power report. The system shown is using the parameterized multiplier at 16 bits and BRAM

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.

5.2.3 Timing Measurements

All designs will be synthesized to run on the Nexys 4 DDR, which has a 100MHz clock. As a means of comparison of maximum timing, worst negative slack 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 that that 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 [54]:

$$Period_{min} = Period_{actual} - WNS \tag{5}$$

$$F_{max} = \frac{1}{Period_{min}} \tag{6}$$

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

$$F_{max} = \frac{1}{10^9 - WNS} \tag{7}$$

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

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

5.2.4 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 fairly easily without having to continuously wait for large projects to synthesize, which can take lots of time. 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.

5.2.5 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:

$$Speedup = \frac{T_{old}}{T_{new}} \tag{8}$$

5.2.6 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)
- Determine the Euclidian 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:

$$Change\ in\ Accuracy = \frac{Distance_{32}}{Distance_{16}} \tag{9}$$

5.3 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 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, only 16 bit calculations will be used. 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 coefficient file used for the BRAM is available in Appendix B.3.1. The testbenches used to test the Xilinx IP is available in Appendix B.3.2 and Appendix]B.3.3.

5.4 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 and Raspberry Pi implementations. This module also acts as a "sanity check" for data in and out of the implementations. Tools exist to verify half, single and double precision online, but none for the arbitrary sized implementations used in this project.

5.4.1 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 C.1.

A second, in-depth online search yielded the *anyfloat.py* module [55]. Some additions were made in order to streamline the process. The *anyfloat.py* module can be seen in Appendix C.2.

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 E.2.

An analogue receiver is shown in Figure 5.6.

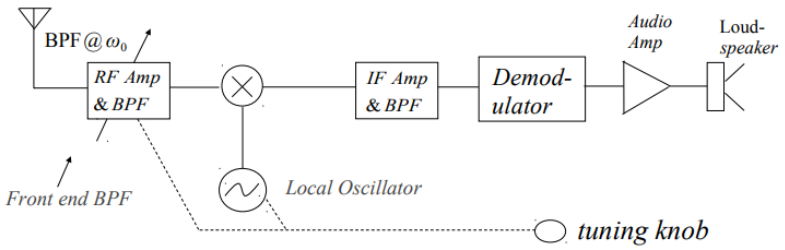


Figure 5.6: A heterodyne receiver [56]

In order to simply 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.

The tools available in MATLAB 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 [57], 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
```

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

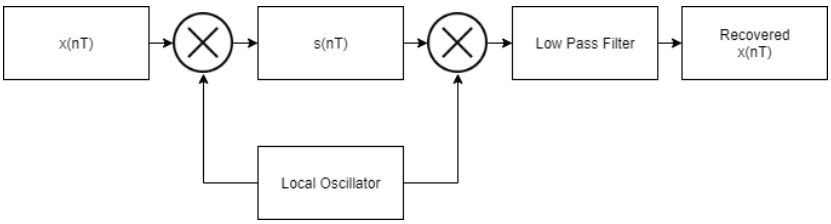


Figure 5.7: The implementation of heterodyning in this experiment

5.5.1 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 [58], which is a 16 bit 250ksps ADC.

```
% 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:

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

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

Table XV: 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.

5.5.2 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) and is considered easy to implement, hence the reason for this choice.

A moving average filter is defined as follows:

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

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);
```

5.5.3 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.

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

5.5.4 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. Estimate the required word size at different stages of the system
   For example, this is a multiplication operation. The bit width can be estimated by examining the amplitudes of the signals, determining the maximum, and seeing which floating point implementation is able to store the maximum. Or for precision, seeing what the varying levels of precision are for each implementation of the waves are, and choosing a floating point size accordingly.

3. Run simulations
   This stage simply involves gathering the data. In order to run all combinations, a total

of 16 experiments must be run. This is due to the following:

- Varying the precisions of the information signal (4 options)
- Varying the precisions of the resultant signal (4 options)

4. 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.

5.5.5 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.
- There is no way of effectively measuring the benefits of reduced precision in this example
  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.

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 [52] floating point operations were accurate, they were tested prior to parameterisation, and the results compare to an online tool [59]. The operations, their result, and the expected result are all documented in table XVI 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 protocol used for each mathematical operators is the same. The full testbench is available in Appendix A.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 A.1.1. Despite differences seen in Table XVI, the results produced by the modules when represented in binary and hexadecimal form are the same as those produced by the online calculator.

```
// 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
```

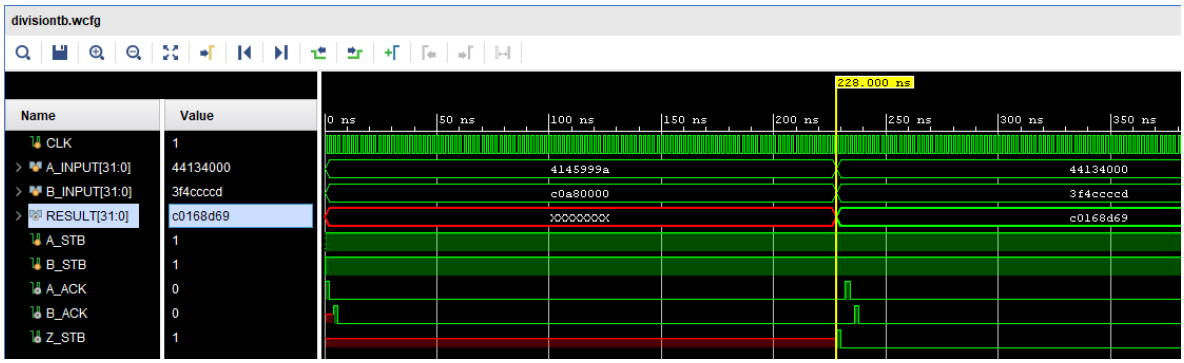


Figure 6.1: Division Waveform for Jon Dawson's 32-bit floating point divider

Table XVI: 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	5.87747175411144e-39	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 attempted designs and final 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 [59]. 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 XVII.

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

	16 bits		**32 bits**	
Decimal Representation	**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	ffffffff
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 XVIII.

Table XVIII: Hexidecimal 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	3dcccccccd
inf	70	7f800000	7f0000	7f80000000

6.2.3 Accuracy to Decimal Representations

IEEE-754, but it's nature, 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 XIX.

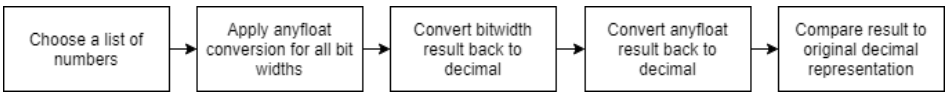


Figure 6.2: The process followed in this experiment

Table XIX: 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 XIX 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.

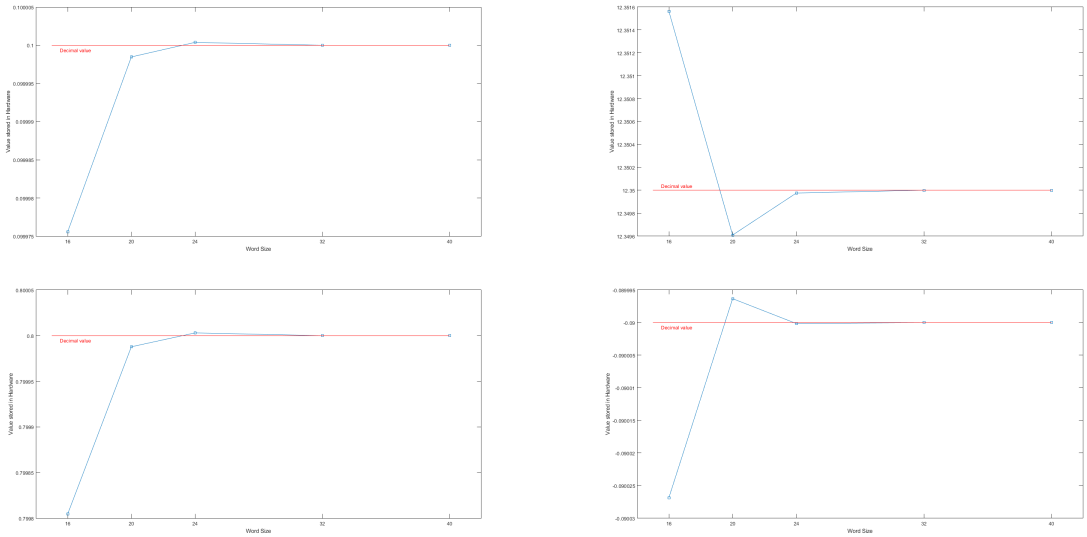


Figure 6.3: Changes in accuracy of representation in hardware for 0.1, 0.8, 12.35 and -0.09

6.3 Parameterization of the Verilog Modules

The next step was parameterizing the Verilog Modules in order to run comparitive tests. The modules were parameterized incrementally, using git as a means of version control. Once parameterized, 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, save the change in parameters. 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 (
//devider #(.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 [59] is recorded in Table XX. Waveforms are available in Appendix A.1.2.

Table XX: 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	0200	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 word sizes for the differing algorithms. The method of determining these values is given in Section 5.2. All results given in table form are recorded in Appendix G.1. 40-bit division did not synthesize, and as a consequence is excluded from the results.

6.4.1 Resource Use Per Module

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

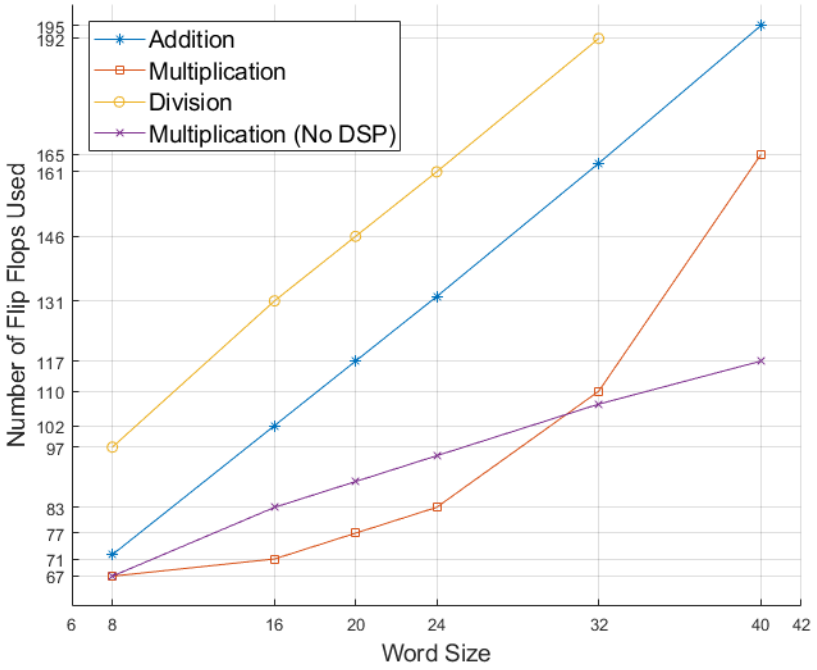


Figure 6.4: Flip flops required to implement various operations at varying precision

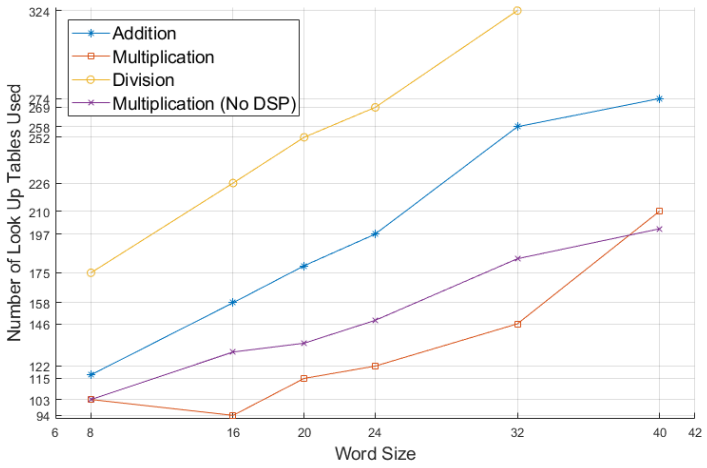


Figure 6.5: Look up tables required to implement various operations at varying precision

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.

Table XXI: Comparison of resources when using DSP

	WNS		LUT		FF	
Word Size	**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. 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 word size increases, and hence the overall requirements of the power consumption change.

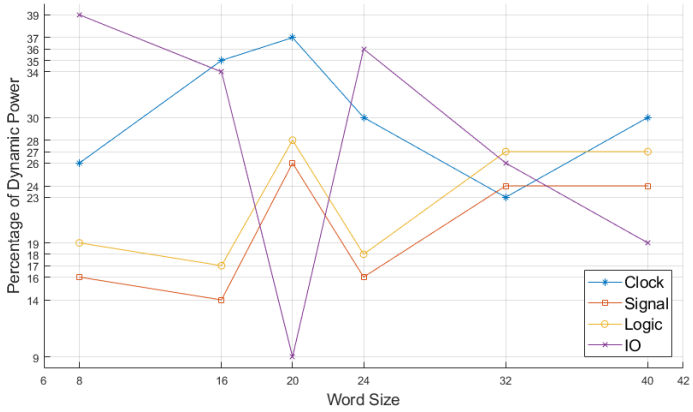


Figure 6.6: Change in dynamic power for Addition as word size increases

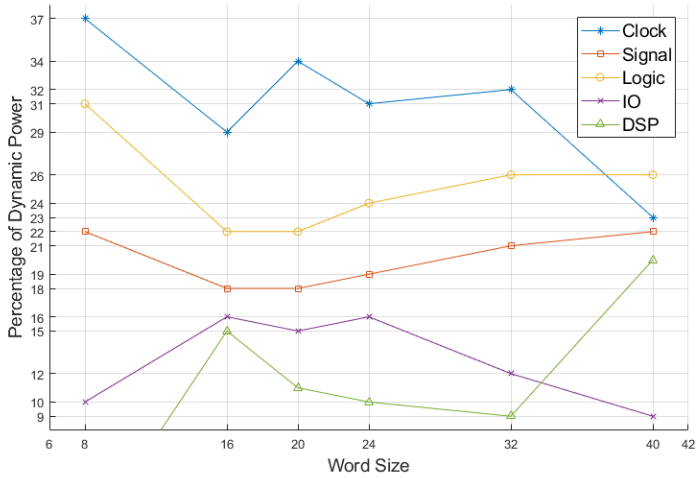


Figure 6.7: Change in dynamic power for multiplication as word size increases

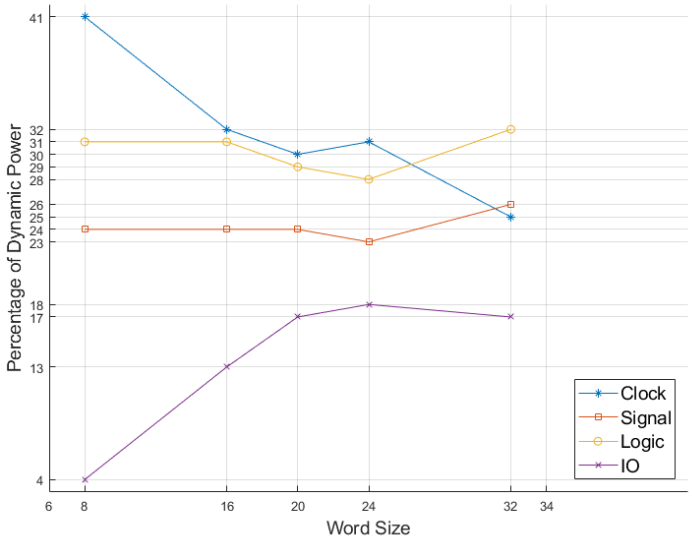


Figure 6.8: Change in dynamic power for division as word size increases

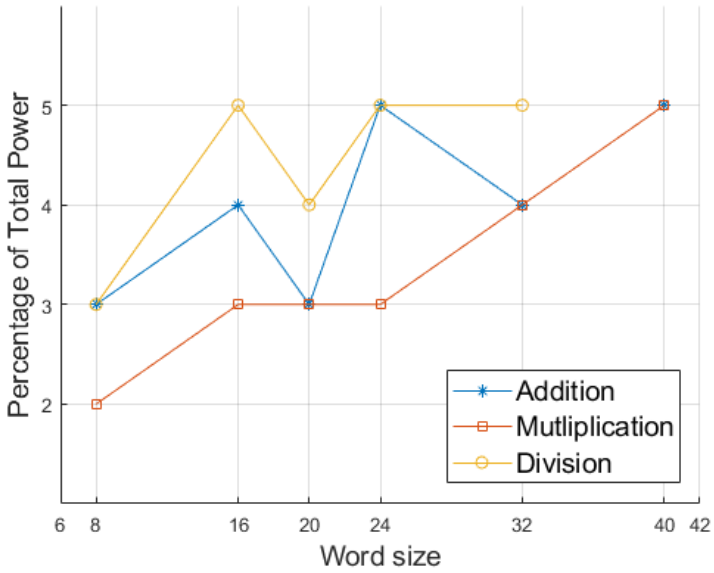


Figure 6.9: Change in dynamic power for the system as word size increases

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 B.2. The results are shown in Figure 6.10. The maximum time taken is 74 seconds, with the mimimum time being 112 seconds, a difference of 38 seconds.

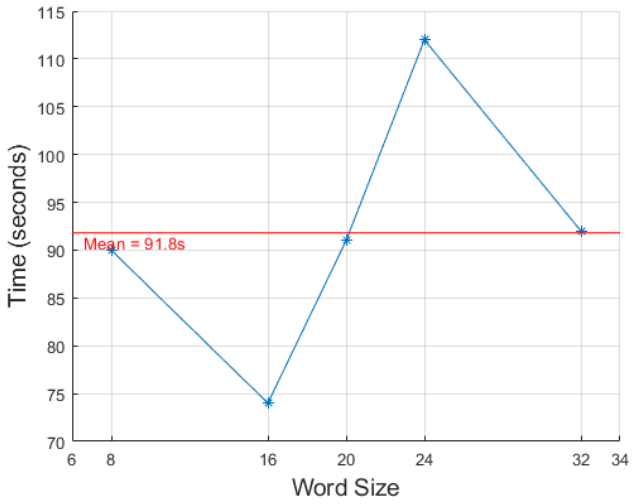


Figure 6.10: The time taken to generate a bitstream for various word sizes

6.4.5 Timing

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

Table XXII: Change in WNS for varying word sizes

Word Size	Addition	Multiplication	Division
8	4.848	4.624	4.84
16	4.148	4.456	4.631
20	4.67	4.292	4.062
24	4.49	4.286	4.515
32	4.023	3.341	4.229
40	3.98	0.714	NA

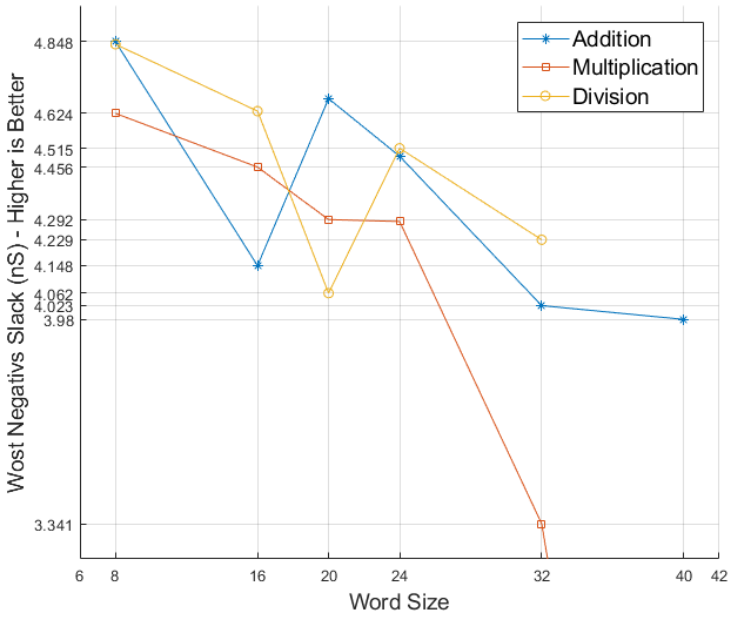


Figure 6.11: Change in WNS as word size increases. Higher is better.

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

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

	Addition		**Multiplication**		**Division**	
Operation	**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 XXIV below.

Table XXIV: 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 word size is shown in Table XXV below, using Equation 8:

Table XXV: 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**

6.4.6 Accuracy and Precision

There is a loss in precision when moving to lower bit widths. Table XXVI below records the loss in precision 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 XXVI: Comparison of precision for calculations involving speedup

Operation	**Addition**	**Multiplication**	**Division**
1	2	0	0.00002941263128
2	0.5	0	0
4	0	0	0.00009999999907
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 XXVII.

Table XXVII: 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 B.3.2, while the test bench used for the parameterized implementation can be found in Appendix B.3.3. All results used in this section can be found tabulated in Appendix F.

6.5.1 Comparison of Results

Table XXVIII shows the comparison of the results produced by the modules. The operands used for each operation are the same as previous experiments.

Table XXVIII: 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

Figures 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.

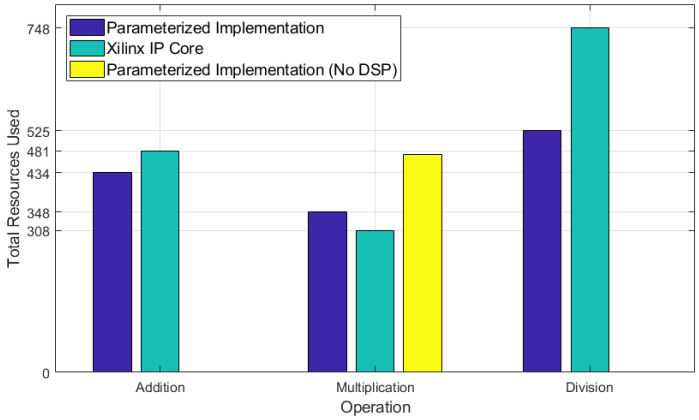


Figure 6.12: Total resources used across implementations

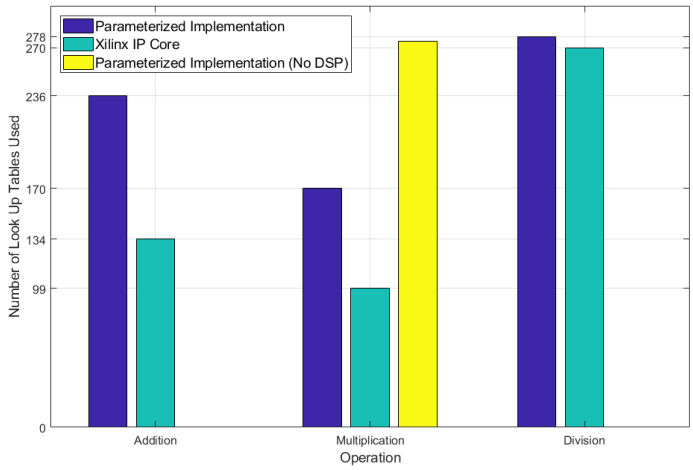


Figure 6.13: Look up tables used across implementations

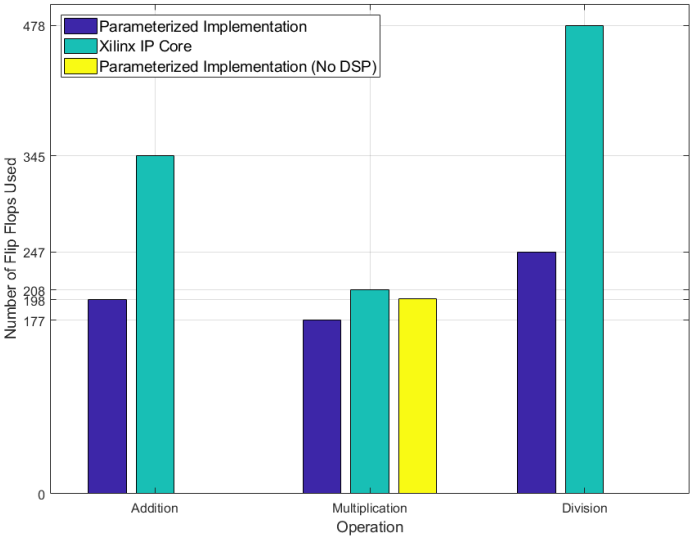


Figure 6.14: Flip flops used across implementations

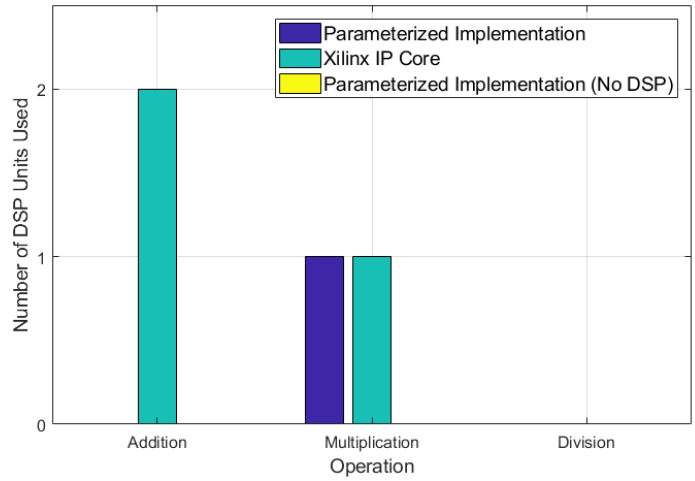


Figure 6.15: DSP units used across implementations

6.5.3 Power Usage

Figures 6.16 through 6.18 show how dynamic power requirements compare between the paramaterized 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.

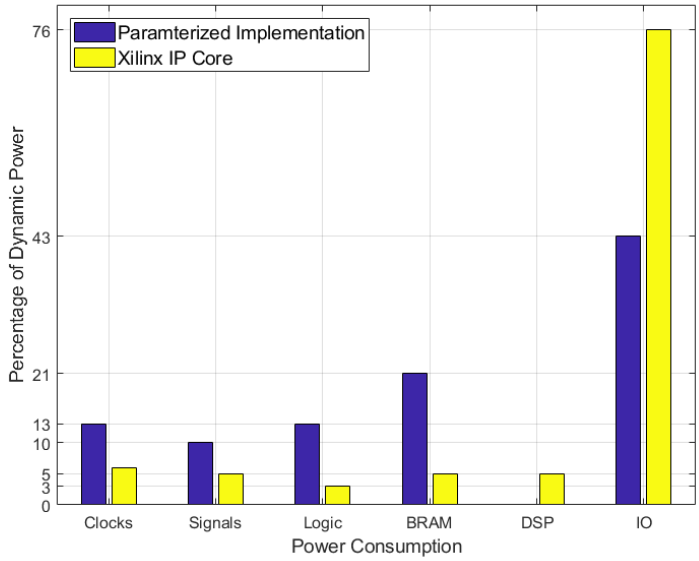


Figure 6.16: Change in dynamic power per signal type for addition

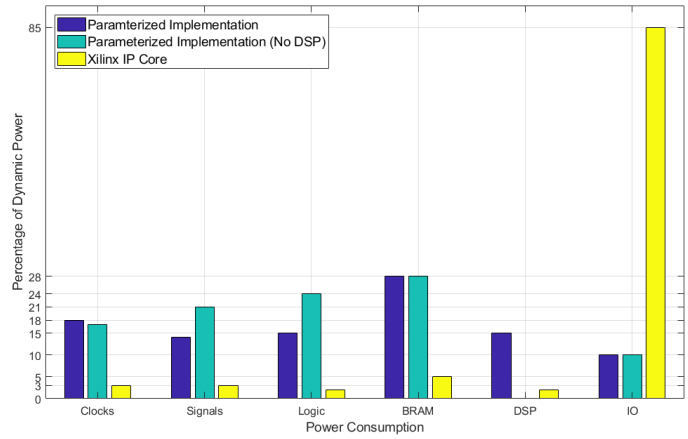


Figure 6.17: Change in dynamic power per signal type for multiplication

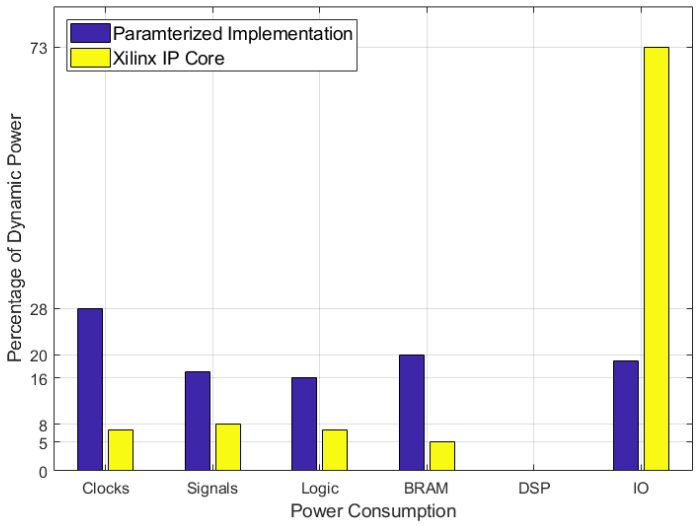


Figure 6.18: Change in dynamic power per signal type for division

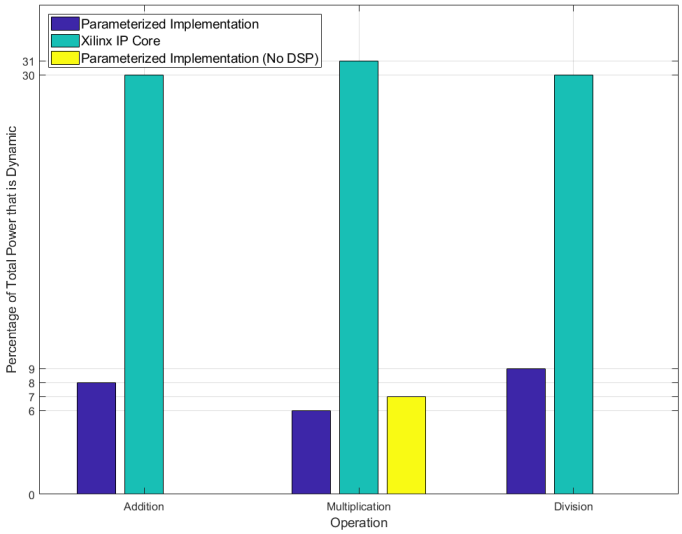


Figure 6.19: Change in total dynamic power between parameterized and Xilinx IP implementations

6.5.4 Timing

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

Table XXIX: 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 XXX: 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 (rather than the FPGA), 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 E.1. Table XXXI shows the results generated by the MATLAB script vs the results generated by the online IEEE-754 calculator available at [59] for 16 bit precisions. It was decided that the fp8 and fp16 casts provided by [57] were suitable to use as a comparison.

Table XXXI: 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:

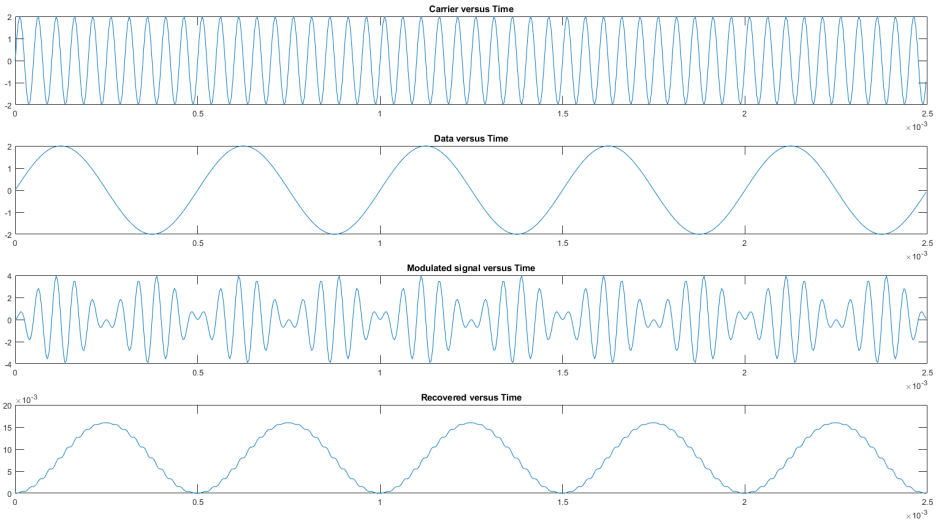


Figure 6.20: The resultant waveforms at IEEE-754 Double Precision (64 bits)

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

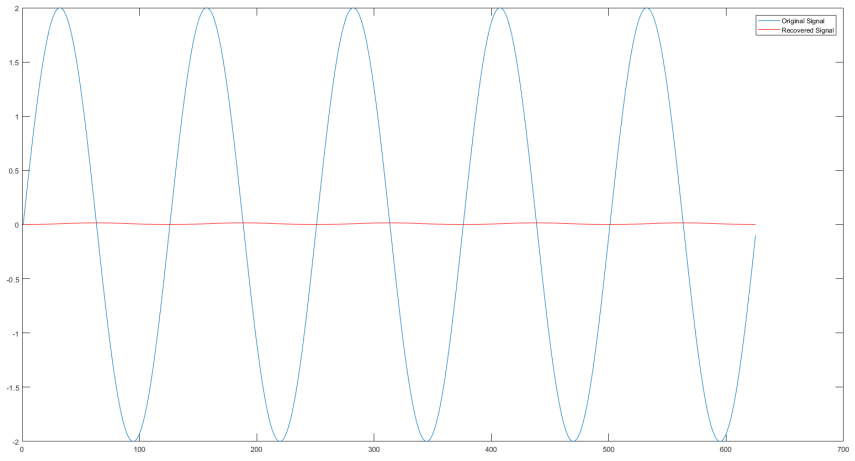


Figure 6.21: Comparison of original and recovered waveforms

Figure 6.22 below shows the original signal and recovered signal, when the recovered signal

is shifted and scaled.

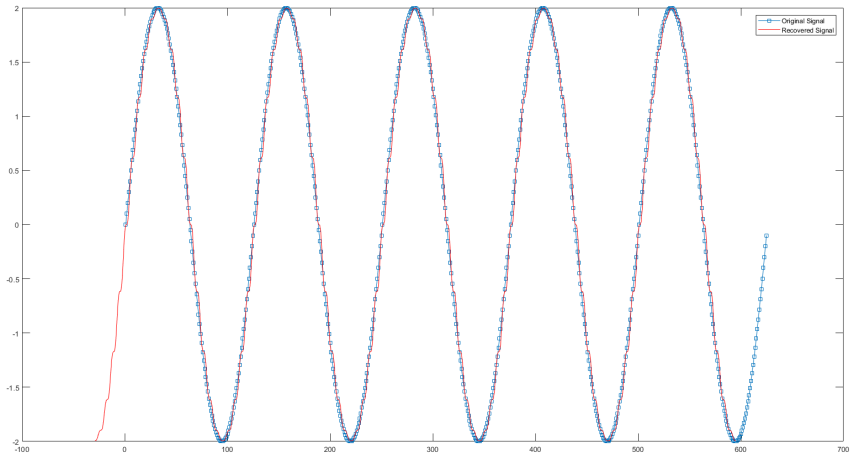


Figure 6.22: Scaled and shifted recovered waveform compared to original information signal

At this scale the signals are visually similar, however by zooming in it can be seen that they do differ.

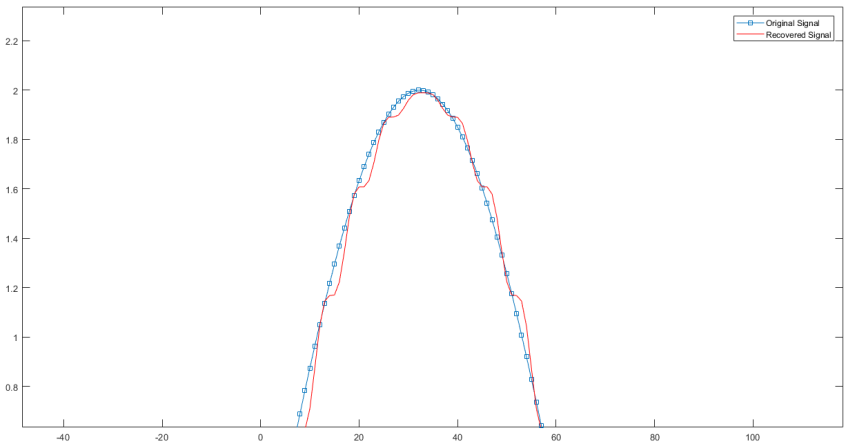


Figure 6.23: A zoomed scale-and-shift to highlight the differences between original waveform and recovered waveform

6.6.3 Comparing Output when Utilizing a Single Bit Width Throughout

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. 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 emphasize the difference between the waves, the scaling and shifting operation has been applied to all four waveforms, and the differences recorded in Table XXXII.

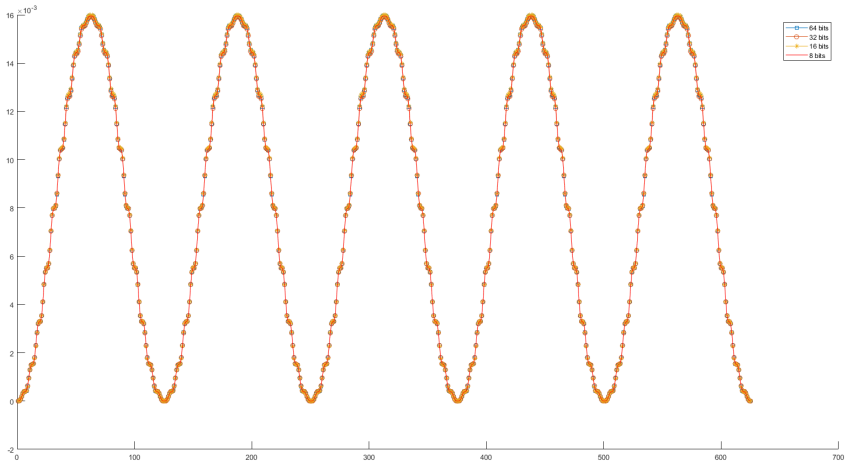


Figure 6.24: A graph showing the reconstructed information signal when using 8, 16, 32 and 64 bits throughout the process

Table XXXII: 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 (see Table XXXIII:

Table XXXIII: 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 XXXIV, 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 XXXIV: 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

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 difference will likely be greater for larger implementations. This can be avoided however by developing an IP core and moving the modules into out-of-context synthesis, requiring synthesis for the parameterized modules to be run only once.

7.3 Power

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 digital 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.

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 took half the time.

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 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 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 analyzed and conclusions drawn about the speed of execution, size of implementation, power consumption and accuracy for varying word-sizes

In addition to meeting the above requirements, the project presented Verilog modules for parameterized implementations of floating point operations. These can be used to conduct further experiments on adjusting word size in larger systems.

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, especially considering the positive results of this investigation. 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 nit-width optimization programs in an attempt to further decrease size of implementation, and see what effect is had on SWAP, precision and speed.
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
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.