Project: FPU Design

Final Report

Pavitar Singh Devgon, CID: 00684127
Jake Humphrey, CID: 00693689
Zifan Guo, CID: 00684696
Weng Nam Lio, CID: 00683021
Raj Mukherji, CID: 00684765
Nikolaos Dionelis, CID: 00690438

Third-Year Group Project
Department of Electrical and Electronic Engineering
Imperial College London

Supervisor from Imagination Technologies: Dr Theo Drane
Supervisor from Imperial College: Dr George Constantinides

June 25, 2014

Statement of originality

We confirm that this submission is our own work. In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others. We are aware that this course is bound by penalties as set out in the College examination offenses policy.

Signed: Pavitar Singh Devgon, Jake Humphrey, Zifan Guo, Weng Lio, Raj Mukherji, Nikolaos Dionelis

Executive Summary

This report provides an overview of the design, verification and analysis of a Floating-Point Unit, as specified by Imagination Technologies. A Floating-Point Unit (FPU) is a hardware block which performs mathematical operations on binary numbers, ranging from basic addition to more complicated operations, such as normalising a vector. The binary numbers were required to conform to the IEEE 754 standard, with support for denormal numbers.

The hardware design of the FPU was detailed in VHDL code, with a C++ implementation that was behaviourally equivalent for testing. Each mathematical operation was written as a separate entity, with an overarching top-level design to interpret the inputs, link the relevant functions and output the correct result. We implemented each operation to the user-specified accuracy. However, due to time constraints, we were unable to consider many appropriate optimisations, the exploration of which could lead to future improvements.

The importance of verifying the design was impressed upon us by Imaginations, highlighting the need to ensure that the code is, above all else, accurate and that the code functions as expected. A three-step testing procedure was devised, consisting of directed testing, constrained-random testing and random testing. The results of the VHDL code were compared to a reference model to ensure accuracy. In turn, the results of the C++ code were then tested to confirm equivalence to the results of the VHDL code. We were able to at least meet the user required accuracy in all operations, and in the case of division and 2D Dot Product, we surpassed the requirements.

The design was also analysed in terms of power usage, throughput and latency, and physical size. Various approaches were attempted, but the most accurate results were given by running simulations to work out the rate at which signals in our design switched state, as an indication of the activity. Our results were benchmarked against an existing technology to evaluate the efficiency of our design. Our design ended up being comparable to or more efficient than the existing technology, except for one operation, multiplication, which was about 3.5 times slower and 30 times larger. We have so far been unable to identify the shortcomings in our own design that lead to this.

The final product was an FPU with latency 2 which implemented all required operations to the specified accuracy. Further work includes optimising or replacing algorithms, as well as pipelining the design. We also conclude that denormal support requires a disproportionately and, to us, unexpectedly large amount of resources and power whilst slowing down the timing results of the final product.

Contents

1 Introduction

Technology is ever evolving with tremendous haste and the demand for high accuracy systems is apparent. Imagination's PowerVR graphics technology division designs graphics processing units used for a wide range of applications. Notably, the PowerVR graphics processors constitute a major component of Apple's iPhone devices [1], as well as those of Samsung, Sony and others. Graphics processors rely heavily on Floating-Point Units (FPUs) to perform the necessary calculations with high accuracy and speed. For the implementation of FPUs on mobile real-time graphic displays, consideration has to be made for the size and power consumption of each module, as resources are limited. One particularly intensive aspect of graphics processing is ray-tracing, which is used to render lighting and reflections in a realistic manner. There are numerous mathematical operations that need to be calculated for each pixel, necessitating the need for a dedicated FPU.

1.1 The Project

During the Group's first meeting with the client, Imagination Technologies, the aim of the project was outlined: to create a hardware design of an efficient FPU architecture which meets the user-specified requirements. In Table 1, the mathematical operations which the FPU must implement are defined, as well as requirements for the accuracy, latency and throughput of each one. Accuracy requirements were given in terms of the *ulp*, the Unit in Last Place, where an accuracy of 4 ulp, for example, means that our answer is accurate enough if it is any of the eight floating-point numbers adjacent to the correct infinite precision answer, four on either side.

The hardware designs were written as VHDL entities, collected together in a top level design. The input and output numbers conformed to the 32-bit IEEE 754 format for floating-point numbers, with support for denormals. This will be further explained in later sections.

The project brief also outlined that a large part of the project be devoted to verification. This is due to the fact that bugs in the hardware description code would be easier to fix than bugs which propagated through to the actual hardware fabrication. The entire test space was too large to exhaustively search; for a two-input operation working in floating-point format, there are $2^{64} = 1.84 \times 10^{19}$ possible inputs. Therefore, an efficient verification plan needed to be created to test the functional correctness of the code. Imagination Technologies themselves devote a lot of manpower to testing their designs, since accuracy is the top priority when shipping products to the customer. Similarly, the Group had to be confident that the operations would satisfy the user requirements.

Due to the fact that the FPU would be used in mobile technologies, power, timing and area characteristics for the design had to be calculated. Once the proposed design reached the desired accuracy, the Group had to further optimise the design of each operation to be more efficient in terms of throughput, latency, power and area. There was often a trade-off at this point because making the FPU optimal in one metric meant sacrificing performance in another metric.

Table 1: The required operations for the FPU

Opcode[e]	Operation	Formula[e]	Accuracy	Cycles per Op	Latency[f]
0000	No Operation	p	Exact[a]	1	x
0001	Multiply	pq	Exact[a]	1	x
0010	Add	$p+q$	Exact[a]	1	x
0011	Subtract	$p-q$	Exact[a]	1	x
0100	Multiply-Accumulate (MAC)	$pq+r$	Exact[a]	1	x
0101	Divide	$p \div q$	—[d]	4	$4x$
0110	2D Dot Product	$pq+rs$	nwc[c]	1	$2x$
0111	3D Dot Product	$pq+rs+tu$	nwc[c]	1	$2x$
1000	Square Root	$p^{1/2}$	4 ulp[b]	3	$3x$
1001	Inverse Square Root	$p^{-1/2}$	4 ulp[b]	4	$4x$
1010	2D Euclidean Distance	$\sqrt{p^2+q^2}$	nwc[c]	nwc[c]	nwc[c]
1011	3D Euclidean Distance	$\sqrt{p^2+q^2+r^2}$	nwc[c]	nwc[c]	nwc[c]
1100	Normalised Vector	$\frac{p}{\sqrt{p^2+q^2+r^2}}$	—[d]	nwc[c]	nwc[c]
1101	Unused				
1110	Unused				
1111	Unused				

[a] Or as accurate as the floating-point format will allow.
[b] Units in Last Place
[c] No Worse than Chaining constituent operations
[d] No set accuracy; the Group is to specify how accurate the result will be
[e] p,q,...,u each represents a floating-point number
[f] As yet unspecified, but each operation is measured relatively

2 Architecture Implementation

The FPU, when viewed as a black box, has a 4-bit input for the operation code and 32-bit inputs for the operands. The maximum number of inputs needed is 6, for the 3D Dot Product. All operations output one 32-bit number and as previously stated will conform to the IEEE 754 format. To reduce area along with power consumption, resources could be shared across functions. Resource sharing applies to components of identical bit widths. For functions with independent hardware, scheduling could increase the throughput of the FPU. However, due to time constraints, scheduling techniques were not investigated in this project.

The FPU was designed in VHDL and C++ and both implementations were required to be bitwise equivalent. The VHDL code constituted the hardware design of the FPU, which would then be used to configure gate-level hardware. This would be the product that is shipped to customers. The C++ code was an additional design that mimicked the behaviour of the VHDL code without necessarily writing the functions in the same way. The bitwise equivalence meant that both codes produced identical outputs when given the same inputs. The C++ code is used in simulations to test the software performance before committing to hardware. This could also be shipped to the customer, if requested, for them to perform their own tests.

To simplify the top level, the FPU was made to have a latency of two cycles (clock data in, then clock result out). Ideally, in order to increase the throughput, the design should be inspected to determine the critical path and a pipelined design should be explored. In addition, pipelining would reduce the glitches in the circuit due to the registers that are inserted between the pipeline stages. Therefore, the power lost due to glitches would be reduced. On the other hand, pipelining would mean the use of more registers that leads to the increase of latency and area. However, in this project, only the addition/subtraction unit was pipelined.

3 Hardware Design of FPU

3.1 Overview

In this section, the design architecture will be presented. The VHDL code can be found on Github, a link to which is given in Appendix A. Each operation was written as a separate VHDL entity, as shown in Figure 1, with a top level design joining them together. Some of the more complicated operations were written as chained invocations of the more basic functions. For example, the 3D Euclidean Distance was calculated by performing a dot product followed by a square root. One aim of the project was to write each function independently, removing redundant stages and running steps in parallel. However, due to time constraints, this was only done for the Multiply-Accumulate and the 2D Dot Product entities.

3.2 Exception Handling

The floating-point format allows positive and negative zeros to be expressed as well as positive and negative infinities and an error condition, NaN (not a number). Positive and negative zeros differ as positive numbers would underflow to positive zero, while negative numbers underflow to negative zero. If an invalid operation was requested, such as dividing zero by itself or attempting to square root a negative number, the FPU would return a NaN.

3.3 Normalisation and Rounding

This section details a step required in every operation. The input(s) and output of each entity are single-precision floating point numbers, comprised of 32 bits. However, the result of a computation would not necessarily conveniently encode to such a format. A step was required to prepare the value for encoding which consisted of *Normalisation* and *Rounding*.

Normalisation refers to the shifting of the output significand so that the leading 1 is the MSB (bit 24). This is necessary as the IEEE 754 format uses a 23-bit significand field, with the 24th bit assumed to be a 1. This stage is complicated by the support of denormals, as they have a 0 in the MSB.

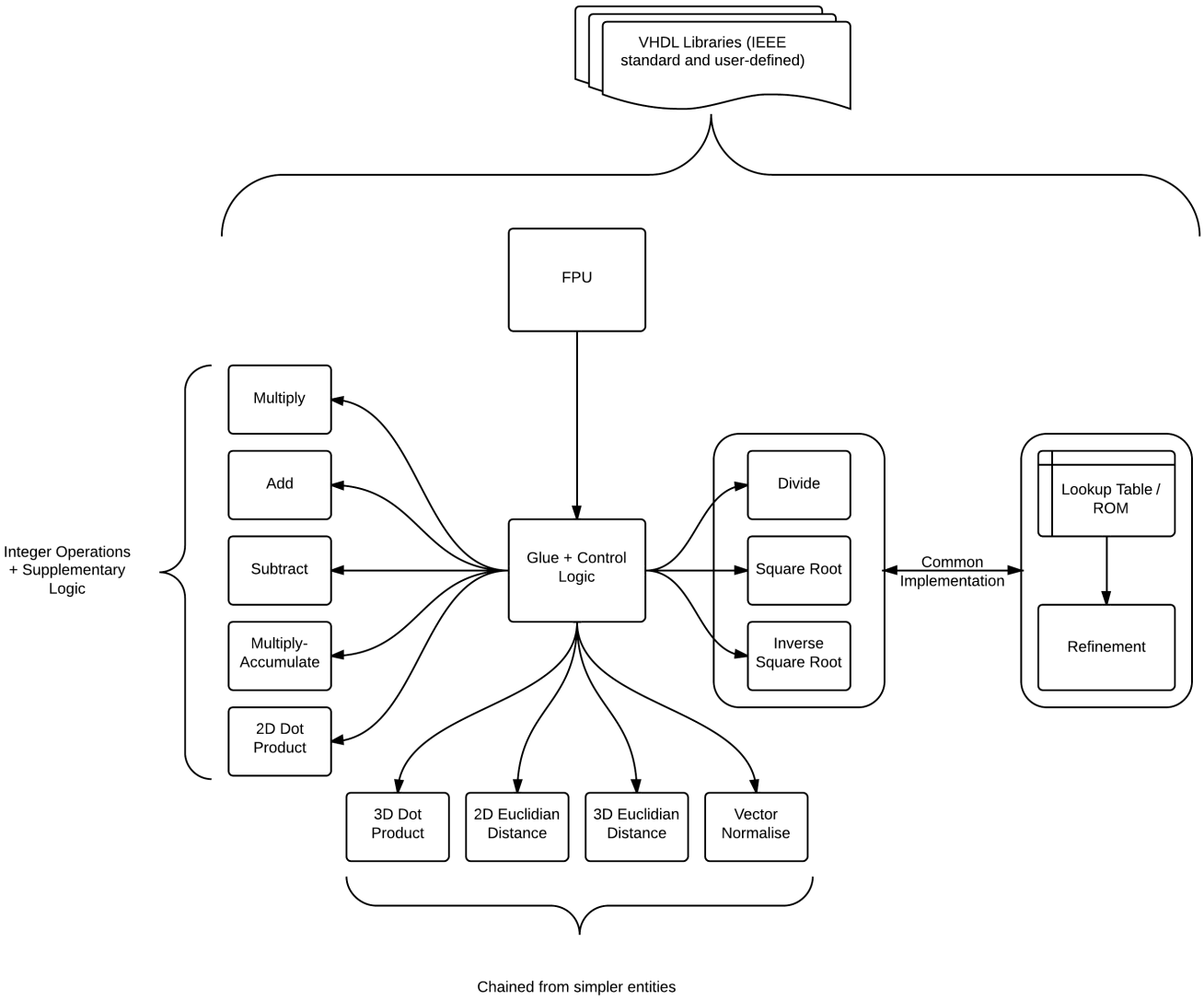

Figure 1: How the Top-level Design was partitioned into smaller tasks.

Rounding refers to the encoding of a higher-precision number to one of the available floating-point numbers. Usually, we simply use the nearest representable number. In the case where the number was equidistant to two candidates, it would be rounded to the one ending in a 0. This standard is known as *Round to Nearest, Tied to Even*, or RTE, and was used due to the fact that it was independent of sign.

These stages were necessary at the end every operation, and one of the main ways to optimise the code when fusing operations is to get rid of superfluous normalisation and rounding stages. It may appear that this approach would speed up the resulting entity over the chained entities, but this is not always the case. Omitting the normalisation and rounding steps results in higher precision at the output. This means later steps in the operation needed to work at higher precisions, invariably resulting in larger and slower hardware. The motivation was to obtain a more accurate result by eliminating rounding errors.

3.4 Basic Arithmetic Operations

3.4.1 Addition and Subtraction

A single entity was used to implement the addition and subtraction operations due to their similarity. Most floating-point algorithms that perform these two operations require 4 steps. Firstly, based on the magnitude of two numbers, a swap is performed to ensure the smaller number is always being subtracted from the larger. Thus, we do not have to account for negative numbers in the output. Secondly, the difference between the two exponents is calculated and the significand of the second operand is shifted to the right by the difference, ensuring that the numbers are of equal exponent size. This allows the third stage, a fixed point addition/subtraction, to be performed. Finally, the result may need to be normalised, caused by an overflow, and then rounded to 23 bits.

There exist many ways to implement the same algorithm in hardware. Figure 2 shows our initial design of the add/sub block. The bit width of each variable is indicated on the connectors. For addition, not every bit of precision is required to ensure accurate rounding. Instead, only three extra bits are used in addition to the output format. These are known as *guard bits*.

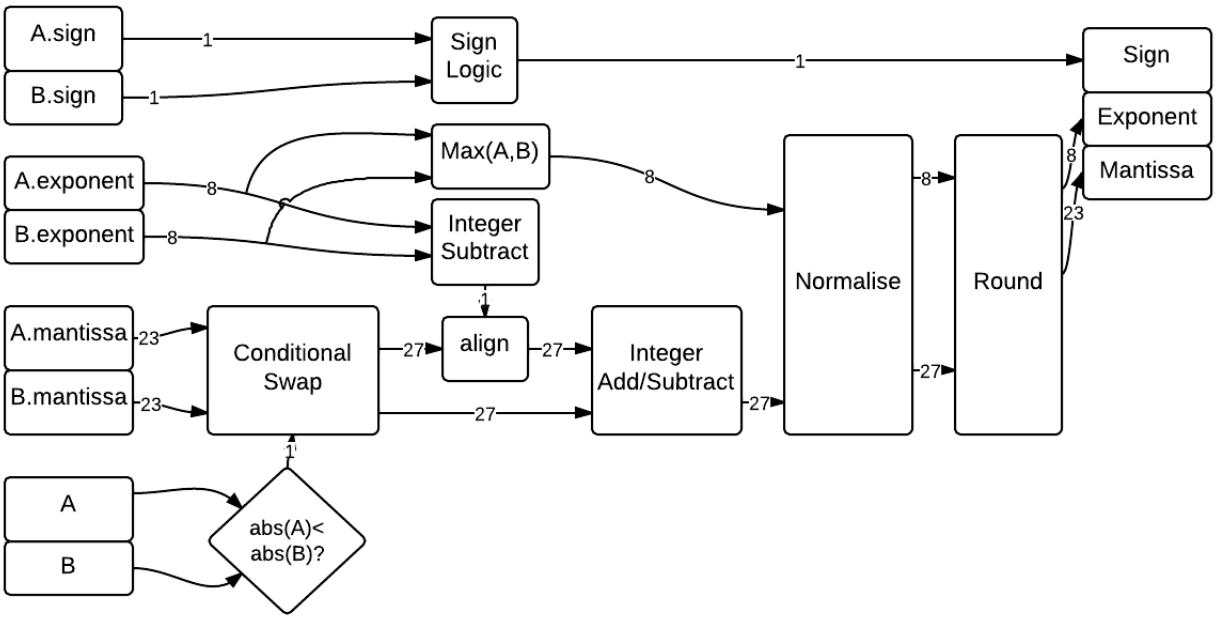

Figure 2: Architectural design of an addition/subtraction block.

The result of an addition of two normal numbers will always require at most a single shift down to normalise. However, a series of leading zeros might occur in the result when adding denormals or performing subtractions. A different normalisation step would be applicable for these cases.

Optimisations were added following the initial design of add/sub. To reduce latency, we researched and implemented a leading zero anticipator (LZA) within this block. Using an LZA, we are able to predict the number of leading zeros in the result of an addition without carrying out the operation beforehand. In most commercial FPU designs, the LZA works in parallel with the adder by calculating the amount of shifts needed for normalisation. The LZA replaced part of the normalisation step, removing the latency it introduced from the critical path, which in our design, lead to a latency improvement of 9.8%. This can be seen in Table 7, in the Application Analysis section.

LZAs typically have two stages. One that generates a vector identical to the result up to the leading 1, and a leading zero counter. The critical path of the LZA is comparable to that of fixed point addition. There are many different implementations of LZAs. In general LZAs can be exact or inexact. Exact LZAs give an accurate prediction of number of leading zeros. However, as a trade-off, they require complex logic and have longer latency. Inexact LZAs, on the other hand, are simple and fast but require an additional stage for error correction. The design we have chosen to implement is introduced by [2], which is an inexact LZA that may underestimate the number of leading zeros by one. The additional 1 bit error correction is then performed in the normalisation step.

In addition, pipelining of the add/sub block was also investigated. The concept of pipelining was presented in the end of Section 2. Figure 3 shows the different stages of the pipeline. The objective of pipelining is to maximise throughput by reducing the maximum latency between registers (the *critical path*). We did

this by splitting the workflow into 5 stages with at most one bit-shift operation per stage.

A significantly higher clock rate (231.2 MHz) is achieved and thus the throughput is improved. Due to time constraints, pipelining for all modules was not feasible for our design, and hence the pipelined adder could not be used in the final (unpipelined) FPU. However, it is an important aspect to consider as a future improvement to the project.

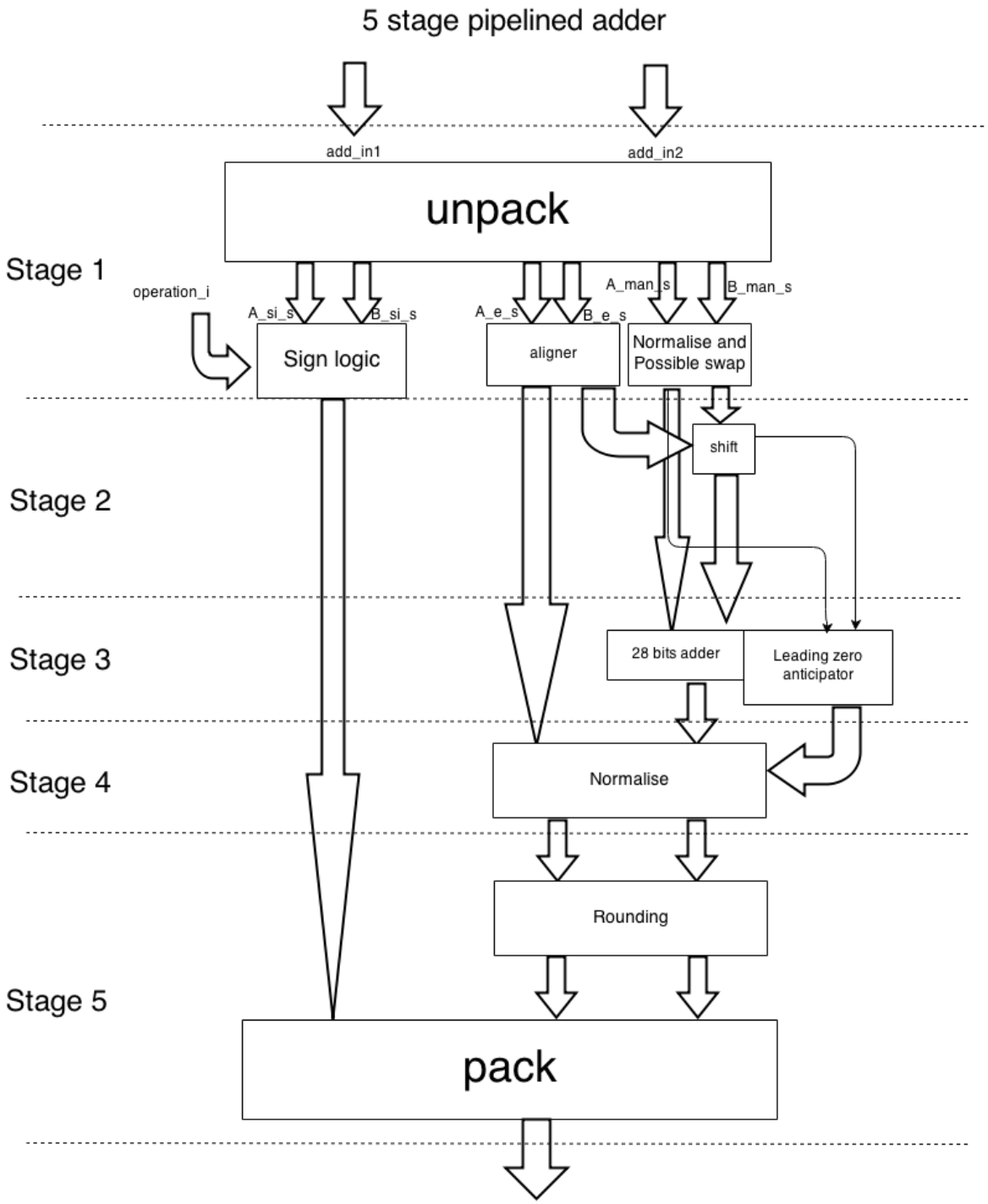

Figure 3: Architectural design of the pipelined adder.

3.4.2 Multiplication

Floating-point multiplication is considerably simpler than addition. This operation reduces to an addition of the exponents and a multiplication of the significands, both of which are encoded as integers, or similar fixed-points, and can be performed in parallel.

Alternative multiplication methods were explored, including Booth encoding, radix-4 separation and long multiplication. A resolution was reached where each method was compared for its ease of implemention in code, area and accuracy of the results. The choice was made to use the fixed-point multiplication inherent in the VHDL environment. This required no extra code to be written and reduced debugging time, while also giving high accuracy results. A futher improvement would be to compare the power comsumption and timing analysis of our implementation with the proposed algorithms we researched.

3.4.3 Multiply-Accumulate ($a \times b + c$)

The product of the first two operands (a and b) is calculated the same way as in the multiplication, using exponent addition and a significand multiplication. However, since this product and the third operand (c) are in different formats, comparison of their magnitude and thus swapping cannot be performed in the same way as the add/sub block. Instead, if a subtraction is required, the significand of c (c.significand) is bit inverted. The significand is aligned as usual by shifting left or right, however now it requires 72 bits to align with the 48 bit product. Next, the ab product is added to the lower 48 bits of c.significand and carried over to the top 24 bits. If the top 24 bits are all zeros (all ones for subtraction), a selector will pass the bottom 48 bits of the result to the normalisation step. Otherwise, the top 48 bits will be passed.

The main difference when comparing the multiply-accumulate function with the addition function is that the normalisation stage must be able to handle a two's complement result and a 48 bit significand. Both a leading zero counter and a leading one counter are needed to normalise the result because of the two's complement format. Dealing with wider bit-width also introduces extra logic and latency. Figure 4 shows the design of the multiply-accumulate function.

3.4.4 2D dot product

The 2D dot product was implemented using two 24-bit multipliers and a 48-bit adder, however there is only one stage of normalisation and rounding. This ensures that the result will be more accurate than a chained implementation as no information is lost during propagation.

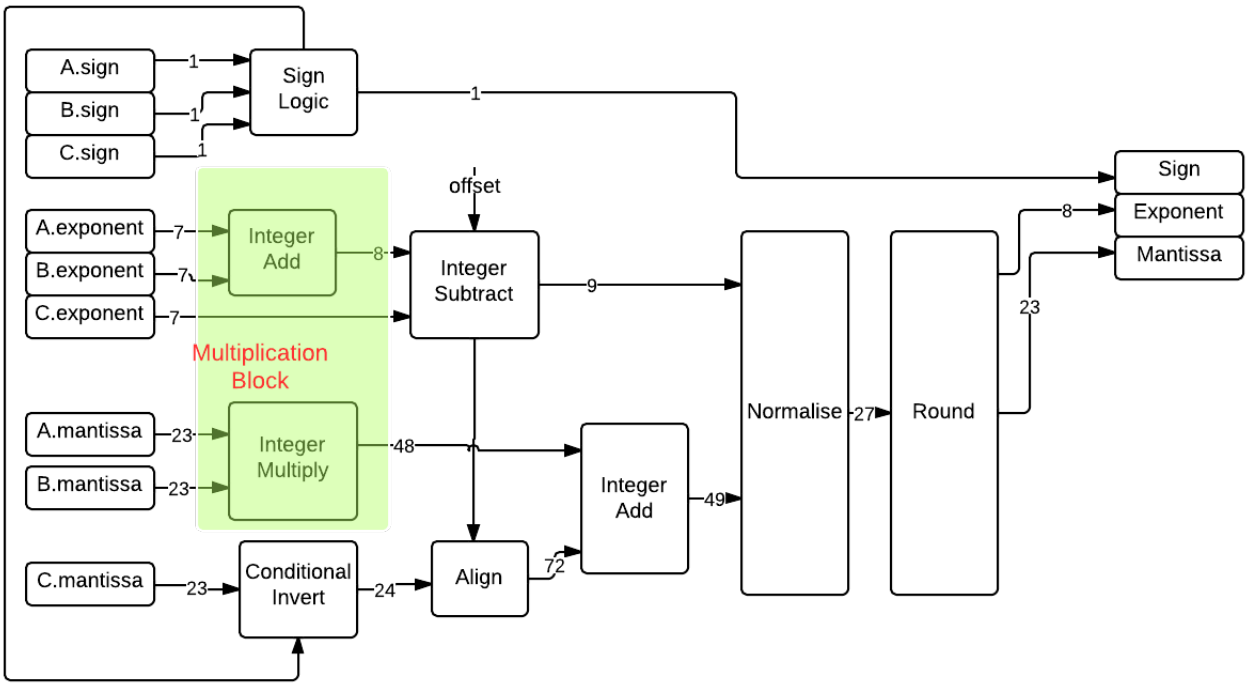

Figure 4: Architectural design of the Multiply-Accumulate block.

3.5 Iterative Operations

A common algorithm we used for three entities involved iterative approximations. An initial estimate of the result was found and the accuracy of this was improved using successive approximation methods, for example the Newton–Raphson Method.

For division and square root, the initial guesses were stored in memory, in a lookup table generated at compile time. An approximate value was estimated by looking up a truncated version of the input in the table. For square root this was an approximation of the final answer, but for division the reciprocal of the input denominator was approximated, which would later be multiplied by the numerator. A trade-off was apparent here between the space and time constraints; if we wanted to reduce the time required to perform each operation, the lookup tables needed to store more accurate estimates and thus increase in size, meaning the FPU as a whole requires a larger area.

The initial guess for the inverse square root function was obtained via the Fast Inverse Square Root algorithm, colloquially referred to as the "Quake Method."[1] This method was chosen as it requires only a single shift and integer subtract, and yet gives an intial result correct to $\sim 3.5\%$[3]

The refinement stages for each operation use the following algorithms:

Division A variant of Goldschmidt's method[4], which achieves quadratic convergence.

[1] "Fast Inverse Square Root." Wikipedia. Wikimedia Foundation, 21 June 2014. `http://en.wikipedia.org/wiki/Fast_inverse_square_root`

That is, it doubles the number of correct bits with each successive iteration, while using only two fixed-point multiplies.

Square Root The Babylonian Method, a specific instance of Newton's Method, which uses only a single fixed-point division and subtraction at each iteration. It is also quadratically convergent.

Inverse Square Root Halley's Method, which converges cubically, but uses three multiply-accumulate operations and one multiplication.

The main priority was to obtain a functionally correct result. Admittedly, more care could have been taken when selecting these algorithms. Future work will be to optimise the choice of algorithm with respect to performance tradeoffs such as latency and area.

3.6 Chained Operations

Due to time constraints, not all operations could be uniquely implemented and optimised. They were instead written as chained operations, designed by having one parent entity instantiating the constituent subentities together. The 3D dot product was chained from a 2D dot product and multiply-accumulate. The 2D Euclidean Distance was found by chaining the 2D dot product and square root. The 3D Euclidean Distance calculated the 3D dot product followed by a square root. Finally, the normalised vector operation divided one operand by the 3D Euclidean Distance of the others.

3.7 Top Level

When it came time to put everything together, all the subentities had been implemented combinatorially. This meant that the FPU would not be able to be pipelined, and would have to run at the speed of the slowest entity. Taking this into account, we decided to minimise area by using the fewest entities necessary to implement every operation.

It turned out that each operation could be expressed as some combination of the 3D Dot Product, Divide, and Square Root entities. Thus, the top level was designed to route the data through these entities, as well as connect up constants to some of the inputs, to implement the operations.

Registers were placed before and after the central control logic and subentities so that inputs could be held constant for the duration of the calculation, and also the previous result could be read from the output whilst the next calculation was being performed. This resulted in an FPU with latency 2.

4 Verification

4.1 Main Three-Step Testing Procedure

Historically, the difficulty of verifying floating-point arithmetic circuits came to prominence due to the popular bug of the Intel Pentium processor[2]. After the division bug of the processor, Intel utilised word-level decision diagrams for verification [5]. However, in this project, formal verification tools, with decision diagrams, were not used due to unfamiliarity with formal methods and due to time restrictions. Instead, test simulations were utilised to verify the proposed design. The verification strategy was split into the three phases of directed testing, constrained-random testing and random testing.

4.1.1 Directed Testing

Firstly, directed testing on basic operations including multiplication, addition and division was performed. Each directed test targeted a specific feature of the design. For example, specifying inputs that could lead to an overflow was used to validate the overflow detection and correction mechanism in the design. Passing these tests provided a clear indication of the design's progress. The test vectors used for these directed tests were found from the Generic Test Plan (GTP) of IBM `FPgen` [6], a test generation framework targeted toward verification of the floating-point datapath. The output of our design is then compared against the output generated by the Golden Model, which will be discussed in the next section.

Apart from limiting the inputs, `FPgen` test vectors also account for more complex constraints such as those on the final result, intermediate results and the relationship between inputs. The GTP is coverage-orientated and each test targets a specific area of the FPU or a particular feature of the floating-point number test space. Directed testing targets many corner cases, which, as expected, allowed bug discoveries in the early stages of verification, which could then be quickly passed back to the design stage. Table 2 presents some of the test models that were used for directed testing.

4.1.2 Constrained-Random Testing

Once the design reached a satisfactory level, the test coverage was increased using constrained-random tests. These aim to cover a wider variety of inputs and introduce possible corner cases that were not considered during directed testing. Constrained-random tests also make it easier to deduce the origin of a given bug. Table 3 shows some test models that apply to all the functions of the FPU. It also indicates the number of all possible test cases for each test model. Hence, it is apparent that the entire number of test cases for each model is impractical to implement. A test generator was written such that test patterns can be specified at compile time and create the constrained random test files as defined.

[2] "Pentium FDIV Bug." Wikipedia. Wikimedia Foundation, 06 June 2014. Web. `http://en.wikipedia.org/wiki/Pentium_FDIV_bug`.

Table 2: Directed test models from IBM `FPgen` test suite [6].

Test file	**Description of the test model**	**FPU operations**
Basic Types	This model tests all combinations of Floating-point basic types, such as normals, denormals, zeros, infinities and NaNs.	All
Hamming-Distance	This test model checks final results that are very close to a specified base value, such as zero or one.	All
Rounding	This test model checks the rounding boundaries. It tests numbers that are on the edge of a rounding boundary.	All
Overflow	This test model checks overflow and near overflow operations.	All
Underflow	This test model checks underflow and near underflow operations.	All
Sticky Bit Calculation	This model checks that the sticky bit is calculated correctly.	All
Rounding Boundaries	This model targets numbers that are on the edge of a rounding boundary.	All
Inputs with Special Significands	This model tests special patterns in the significands of the input operands.	Mult, Div, Sqrt
Addition: Shift (-And-Special-Significands)	This model tests the combination of different shift values between the inputs(, with special patterns in the significands of the inputs).	Add, sub
Addition: Cancellation (-And-Denormal-Results)	This model tests all combinations of cancellation values(, with all possible unbiased exponent values of subnormal results).	Add, Sub
Division by Zero	This model will test the Divide By Zero exception, with inputs combinations of zero, random non-zero number, infinity and NaN.	Div, Sqrt
Division: Trailing-Zeros	This model will create test-cases such that the significand of the intermediate results will cover a specific pattern.	Div

Table 3: Constrained-random test models that were applied to all functions of the FPU.

Test File Number	**Description of the Test Model**	**Number of possible test cases (2 inputs)**
Test 1	Positive normal floating-point numbers	5×10^{18}
Test 2	Negative and positive normal floating-point numbers	2×10^{19}
Test 3	Zeros	4
Test 4	Positive denormal floating-point numbers	7×10^{13}
Test 5	Negative and positive denormal floating-point numbers	3×10^{14}
Test 6	Denormal and normal floating-point numbers: the denormal floating-point numbers have less than half probability of occurence.	2×10^{19}
Test 7	Denormal and normal floating-point numbers: the denormal floating-point numbers have half probability of occurence.	7×10^{16}
Test 8	Infinity, NaN, denormal floating-point numbers, normal, floating-point numbers, negative floating-point numbers.	2×10^{19}

4.1.3 Random Testing

In the end, when the Group was convinced that the design was of a high quality, multiple random tests of 50 000 input pairs were run as a small step towards exhaustive functional coverage of the design, justifying that the entire design works. It is worth noting that performing verification by running test simulations cannot fully prove the correctness of the design since all "large-scale" tests coverage are still infinitessimal in comparison to the entire test space. To provide definitive proof, formal tools would be needed.

Both the C++ and the VHDL code have been verified with the aforementioned three-step testing procedure. Moreover, the properties of each operation have been used for verification. Tables that show the properties of each operation are presented in Appendix B. The C++ code was verified using the same test input vectors. The corresponding output vectors were compared with the correct solutions. All of the tests on the VHDL code were implemented using Modelsim testbenches written by the Group.

4.2 Golden Model

For the basic operations of addition, multiplication, division and square root, the "Golden answer" against which the proposed design was verified was obtained using the arithmetic functions provided in a VHDL IEEE floating point package by David Bishop[3]. This library has been chosen since it conforms to the IEEE floating-point standard and includes various settings such as rounding-modes, special-number handling, number of guard-bits etc.

[3]Bishop, David. Floating Point Package Users Guide. Rep. N.p., n.d. Web. `http://www.vhdl.org/fphdl/Float_ug.pdf`.

Obtaining the exact answer for the multiply-accumulate function to be used as the "Golden Model" has been problematic. In the beginning of the verification process of multiply-accumulate, the `mac()` function from the VHDL Floating Point Library was used. However, during the process, the `mac()` function was found to produce incorrect results for some set of inputs when compared to the results obtained by performing the operations on pen and paper. This led to an investigation of a new trustworthy "Golden Model" to verify the proposed multiply-accumulate function. The `ieee.math_real` package was considered but was swiftly rejected since it is only a model of the real-number line using double precision floating point, which implies double rounding and thus introduces inaccuracy at corner cases.

The third method that was considered for verifying the multiply-accumulate function was to perform the directed tests using the test vectors from the IBM test suite and to compare the obtained results with the model answer provided. Unfortunately, discrepancies were found between the Group's results and the model answer and, in most cases, the model answer from the test suite was blatantly wrong[4]. It was therefore impossible to trust the model answer provided and use the IBM test suite as the "Golden Model".

Lastly, an error-free floating point multiplication algorithm and an error-free summation algorithm were investigated. Using the Dekker product and the twoSum algorithms [7], one is able to obtain the rounded result and the exact error of that result in each operation. By summing up the rounded results and the errors, one can be confident that the resulting value is as close to the infinite-precision answer as possible. However, these algorithms do not work in cases of overflow and underflow. This implies that the floating-point domain cannot be covered entirely and it is impossible to verify the proposed design using this configuration. If the Group had more time, W.Kahan's PARANOIA could also be investigated as a potential candidate for obtaining the exact answer for the multiply-accumulate function.

For more advanced functions such as finding Euclidean Distances, the Dekker product and the twoSum algorithms were used to obtain the "correct" result [7]. In cases of overflow and underflow, the `ieee.math_real` package was used to estimate the "correct" result. Since the required accuracy for these functions is *no worse than chained*, any result between the "correct" result and the result from chaining basic operations was accepted from the design. It is interesting that for some operations there are more than one way of chaining[5] which means the range of the acceptable result can cover the entire floating point space.

4.3 Additional Methods Considered for C++ Verification

In addition to the three-step testing procedure, several other methods were considered for the verification of the C++ code. In particular, the C++ architecture could be verified using the CBMC software tool (C Bounded Model Checker) by declaring assumptions and assertions. Another tool we looked into was W.Kahan's PARANOIA, an application to test the design with emphasis on potential floating point errors caused by rounding. Hard corner cases are used to classify the design's rounding as exact or truncated. In

[4] From the test file `underflow.fptest` [6], an example to illustrate this argument is:
`b32*+ =0 xu +1.1C0954P-29 -1.51EA3DP-119 +0.000004P-126 -> +1.59CB70P33 xu`

[5] To illustrate this concept, the 3D dot product can be chained from three multipliers and two adders, or from one 2D dot product and a multiply-accumulate function.

addition, PARANOIA specifies the design's precision and exponent range, determines whether underflow is gradual and whether the square root is monotonic [8]. Finally, the program can verify if addition and multiplication are commutative operations. However, due to time limitations and unfamilarity with the tools, these methods were not implemented, but could be implemented in the future.

4.4 Additional Methods Considered for VHDL Verification

Apart from the aforementioned verification procedure for the VHDL code, Formality could also be used to verify certain functions of the proposed design. Formality is a software application that uses formal techniques to prove or disprove the functional equivalence of two designs [9]. Due to time constraints, verification with this application was investigated but not implemented. An analysis of how Formality would be used is presented in Appendix D.

The design verification procedure with Formality would be performed as follows. A functionally correct design (i.e. a "Golden" reference design) would be used. This reference design would be compared to the proposed design (i.e. the implementation design). As mentioned in the previous section, the "Golden answer" according to which the proposed design would be verified would be obtained using the arithmetic functions provided in the VHDL floating point package by David Bishop. Therefore, Formality would prove or disprove whether the two designs are functionally equivalent.

4.5 Meeting the accuracy requirements

Table 4 shows the accuracy of our design in comparison with the specification. It is clear that our design satisfied (and in some cases surpassed) all accuracy requirements. Unfortunately, the multiply-accumulate function was not fully verified (as stated in the previous section) but did pass the tests we were able to perform on it and we are fairly confident it is bit exact. The square root and reciprocal square root operations are both only accurate for normal numbers, with a 25% chance where the result falls outside the 4 ulp range with random denormal inputs.

Table 4: Comparing our design with the specification

Operation	**Specified Accuracy**	**Achieved Accuracy**
Multiply	Bit exact	Bit exact
Add/Sub	Bit exact	Bit exact
Multiply-Accumulate	Bit exact	Bit exact (?)
Divide	No set accuracy	1 ulp
(Reciprocal) Square Root	4 ulp	4 ulp (for normals)
2D Dot Product	No worse than chained	1 ulp
3D Dot Product	No worse than chained	No worse than chained
2D/3D Euclidean Distance	No worse than chained	No worse than chained
Normalised Vector	No set accuracy	No worse than chained

5 Application Analysis

Once the designs have been verified, application analysis can begin. The purpose of this section is to evaluate the design with respect to the user requirements for throughput and latency. Table 1 shows the functions of the FPU along with the required accuracy, throughput and latency. Different FPU configurations were considered to ascertain the option that minimises power and latency. Other consumer requirements, such as power consumption and area usage also need to be considered. For testing purposes, we chose to implement the design on the commercially available Artix 7, a lower-end, power efficient FPGA, which we felt would best support our design requirements.

5.1 Timing and Area Analysis

In this project, the software tool Synplify Premier by Synopsys was used to investigate timing and area usage. Synplify Premier was chosen because it enables the inspection of these metrics for FPGA implementations from different vendors, such as Altera and Xilinx.

In general, timing analysis allows the inspection of the critical path, identifying the part of the design that would benefit most with optimisation. Initially, the timing analysis for each function was examined. This allowed the identification of the relative performance of each block, before the top level entity was examined. Table 5 shows the results of some of the FPU operations. Those involving the square root could not be synthesised and thus could not be tested.

Table 5: Results of Timing and Area Analysis

Operation	**Estimated Frequency /MHz**	**Latency /ns**	**Normalised Latency**	**Area /LUTs**
Add/Sub (naive)	65.1	15.4	1	1642
Add/Sub (with LZA)	71.5	14.0	0.9	1631
Multiplication	62.1	16.1	1	1211
Division	54.2	18.5	1.1	1371
Multiply-Accumulate	39.9	25.1	1.6	6066
2D Dot Product	46.3	21.6	1.3	4272
3D Dot Product	20.8	48.1	3.0	10070

When all functions had been implemented and tested, the throughput and latency for each operation was verified against the specification in the Requirements Document. In the above table, the normalised latency is the latency relative to the multiplication implementation. The requirements specified that the latency is measured relative to the multiplication unit.

Multiply-accumulate did not achieve the required latency of equal to the multiplier's latency. This seems to be an infeasible requirement as the multiplier logic is a strict subset of the multiply-accumulate logic

and would invariably result in a higher latency. Consequently, the 3D dot product, which instantiates the multiply-accumulate operation, also does not satisfy its requirement of twice the multiplier latency.

5.2 Benchmarking the proposed VHDL code quality against FloPoCo

By comparing the proposed solution to existing technologies, the solution's effectiveness can be evaluated. One open-source single-precision floating-point compiler that we used for this purpose is FloPoCo.

The latency of corresponding operators with FloPoCo's generated VHDL design was compared to benchmark the proposed solution, and the result is presented in Table 6. We only had time to compare the operators shown in the table. Also note that FloPoCo does not support operations with denormal numbers[10].

Table 6: Timing comparison with FloPoCo

Operation	**Frequency /MHz**	**FloPoCo Frequency /MHz**	**Area /LUTs**	**FloPoCo Area /LUTs**
Add/Sub(LZA)	71.5	73.5	1631	687
Multiply	62.1	216.8	1211	41
Divide	54.2	22.5	1371	1232

As can be seen from the table, the Add/Sub block was on par with that of FloPoCo in terms of maximum clock frequency, while being 2.5 times larger. Our Multiplier, on the other hand, fell short of FloPoCo's benchmark, running 3.5 times slower and using 30 times the area. Finally, our Divider ran nearly 2.5 times faster with about the same size.

5.3 Power Analysis

For power analysis, the FPU system would be implemented on an FPGA board. Calculating the power consumption prior to the board build ensures that the proposed design does not exceed a fixed power budget or a thermal limit. The power consumption can be divided into the following components, which can be analysed separately.

Pre-programmed quiescent power The amount of power consumed by the FPGA before it has been programmed.

Inrush programming power The amount of power required when programming the FPGA until programming is complete.

Post-programmed quiescent power Power consumed by the FPGA at zero frequency.

Dynamic power — Power consumed at non-zero frequency components. The dynamic power is dependent upon two important signal properties: toggle rate and static probability. The toggle rate is the average number of times the signal changes value per unit time. The static probability of the signal is the fraction of time for which the signal is at logic 1.

There are several external factors which also affect the power consumption, independent of our design. Device selection is key as there are various families of devices from several companies, each with different power characteristics. The environmental conditions are responsible for the operating temperature of the device and in turn the static or quiescent power consumption. Device resource usage is the most significant portion of the power consumed and is dependent on the number, type and loading of pins, hard logic boards and global signals.

For power measurements, the Xilinx ISE Design Suite was used. Initially, the project navigator was used to map the hardware description files (VHDL files) onto the physical Xilinx components, creating a hardware map (ncd files). The mapped design was then loaded onto the XPower Analyser to obtain initial estimates for the power utilisation and temperature of our device. These initial estimates were tested again using the XPower Estimator speadsheet. The default settings provided by this spreadsheet were used, with the toggle rate set to 12.5/100.

Finally, more accurate power measurements were obtained by using a different feature of the XPower Analyser. This time, a Value Change Dump file (.vcd) was created which contained timing information, such as the toggle times for each signal. Hence, with the VCD, the toggle rate of the signal can be more accurately represented and a better approximation for the power of the device can be obtained. The results are presented in Table 7. It should be noted that the quiescent (static) power will remain constant at 0.082 W since this was calculated for the Artix 7 board itself.

Table 7: Power analysis for the operations

Operation	**Power/W (default toggle rate)**	**Power/W (simulated toggle rate)**
Add/Sub (naive)	0.036	0.006
Add/Sub (with LZA)	0.041	0.007
Division	0.032	0.032
Multiply Accumulate	0.039	0.006

The central column gives the power calculated with the default toggle rate generated by the XPower tool. This is an overestimate of the power, as we can see in the right-hand column. Using the actual toggle rate, we can reduce dynamic power significantly in some operations. We can see the division operation is more power intensive than the other operations due to the iterative nature of the algorithm.

To put these power numbers into context, we calculated the performance per Watt (Table 8) by dividing the total power consumption, including the quiescent power, by the number of floating-point operations performed per second (FLOPS). Since all operations have a throughput of 1, performance measured in FLOPS is essentially the same as the estimated frequency obtained in the timing analysis section. This table now allows us to see that the add/sub functions can calculate almost twice as many operations given the same power as the division or multiply-accumulate functions.

Table 8: Performance per watts of each operations

Operation	**Performance/Watt (MFlops/W)**
Add/Sub (naive)	740
Add/Sub (with LZA)	803
Division	475
Multiply Accumulate	453

In addition to the Xilinx Suite, we considered using other power analysis tools, such as Synopsys-PrimeTime. This would have provided more support and a different perspective to the analysis. Ultimately we decided not to use this tool; it can sometimes be inaccurate as it depends on probability distributions and assumes signals to be independent.

6 Conclusion and Future Work

In this report, we have discussed the complete design of an IEEE 754 compliant single precision FPU architecture. The FPU supports input/output formats of normals, denormals, infinities, zeros and NaNs, except for the square root operations, which may return inaccurate outputs for denormal inputs. The algorithms used to implement the hardware have been extensively outlined along with any optimisations.

In the design section, implementing denormal support required a lot of work that could have perhaps been better spent optimising the entities in terms of latency and area. Though denormal support was part of the specification, and given higher priority than optimisations, we would advise that future similar projects take into account the disproportionate amount of work, as well as the VHDL code, required for denormal support compared to the relatively small proportion of the number space that they occupy, before deciding whether it is worth implementing.

In the verification section, a three-step testing procedure was introduced to ensure the correctness and accuracy of the VHDL and C++ designs. In addition, the way in which the proposed design meets the accuracy requirements was presented. After this, the power, area and timing metrics were measured and compared against the initial specification set by Imagination Technologies. All passed except for the latency requirements of multiply-accumulate and 3D dot product as detailed in section 5.1.

Furthermore, a timing and area benchmark was performed against an open source floating point compiler FLoPoCo. From this comparison, in Section 5.2, it was concluded that the proposed implementation has better performance than FloPoCo's implementation in terms of frequency in all of the functions apart from multiplication.

Finally, in this project, we were not able to fully verify the Multiply-Accumulate operation since a definitive "Golden Model" could not be found. In terms of future work, we would look to investigate other floating point test suite such as PARANOIA, a floating-point test program by W.Kahan. PARANOIA was presented in the end of Section 4.3. Similar to the IBM FPgen test suite, PARANOIA contains input test vectors that target specific part of an FPU and their corresponding output, which can be used as a reference model for the Multiply-Accumulate function.

Overall, the FPU designed in VHDL was able to meet the user requirements. In industry, the VHDL code could be used to configure hardware and could also be used in further application testing. Alternatively, if there is enough confidence in the design, we would look to manufacture an ASIC, which, as a dedicated device, would be more efficient than the extraneous components on the FPGA. The FPU could be used in a testbench to run actual graphics processing tests, in realisation of its eventual implementation.

7 References

[1] Apple, Imagination Technologies Extend IPhone, IPad Graphics Chip License Pact, *"Apple, Imagination Technologies Extend IPhone, IPad Graphics Chip License Pact."*, N.p., n.d., Web, accessed 05 May 2014. `http://appleinsider.com/articles/14/02/06/apple-imagination-technologies-extend-iphone-ipad-graphics-chip-license-pact`.

[2] Suzuki H., Morinaka H., Makino H., Nakase Y., Mashiko K., and Sumi T.. *Leading-zero anticipatory logic for high-speed floating point addition. Solid-State Circuits, IEEE Journal of , vol.31, no.8, pp.1157,1164, Aug 1996 doi: 10.1109/4.508263* `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=508263&isnumber=11139`.

[3] Chris Lomont *Fast Inverse Square Root* `http://www.lomont.org/Math/Papers/2003/InvSqrt.pdf`. retrieved 2014-06-24.

[4] Oberman, S.F.. *Floating point division and square root algorithms and implementation in the AMD-K7TM microprocessor," Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on , vol., no., pp.106,115, 1999 doi: 10.1109/ARITH.1999.762835* `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=762835&isnumber=16513`

[5] Chen, Yirng-An, and Randal E. Bryant. *An Efficient Graph Representation for Arithmetic Circuit Verification. Rep. N.p., n.d. Web.* `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00969437`.

[6] FPgen Team. *Floating-Point Test-Suite for IEEE.* Rep. N.p.: IBM Labs in Haifa, n.d. Web. https://www.research.ibm.com/haifa/projects/verification/fpgen/papers/ieee-test-suite-v2.pdf.

[7] Constantinides, George A., and Manouk V. Manoukian. *Accurate Floating Point Arithmetic through Hardware Error-Free Transformations. Rep. Imperial College London, 2011. Web.* http://cas.ee.ic.ac.uk/people/gac1/pubs/ManoukARC11.pdf.

[8] Verdonk, Brigitte, Annie Cuyt, and Dennis Verschaeren. *A Precision- and Range-Independent Tool for Testing Floating-Point Arithmetic I: Basic Operations, Square Root, and Remainder. Rep. University of Antwerp, 2001. Web.* http://cant.ua.ac.be/node/6711.

[9] *Formality, User Guide. Rep. N.p., n.d. Web.* http://www.vlsiip.com/formality/ug.pdf.

[10] Constantinides, George, David Boland, and Xuan You Tan. *FPGA Paranoia: Testing Numerical Properties of FPGA Floating Point IP-cores. Rep. Imperial College London,, n.d. Web.* http://cas.ee.ic.ac.uk/people/gac1/pubs/DavidARC12.pdf.

[11] Viitanen, Timo, Pekka Jaaskelainen, and Jarmo Takala. *INEXPENSIVE CORRECTLY ROUNDED FLOATING-POINT DIVISION AND SQUARE ROOT WITH INPUT SCALING. N.p., n.d. Web.* https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/21884/viitanen_inexpensive_correctly_rounded_floating_point.pdf?sequence=3.

[12] *"Floating-Point Test Generator - FPgen." IBM Research.* IBM, 23 Feb. 2001. Web. https://www.research.ibm.com/cgi-bin/haifa/test_suite_download.pl?first=elenag&second=webmaster.

Appendices

Appendix A Code Repository

All of the VHDL and C++ code, as well as testbenches, test vectors, and documentation will be hosted for the forseeable future at https://github.com/EightAndAHalfTails/eee3-imgtec-fpu.

Appendix B Verification Tables

In this appendix, tables that show the properties of each operation are presented. These table were used in the verification process.

B.1 Tables for Addition

The fact that the addition $x + y$ is the same as the addition of $y + x$ (i.e. commutativity of addition) was used for verification. This concept of commutativity can be seen in the following table with the addition of $+0$ and -0. The important relationships in Table 9 are that $(+0) + (-0) = +0$ and that $(-0) + (+0) = +0$ so that commutativity holds.

Table 9: Test cases for the addition/subtraction unit using +0 and -0

Input1	Input2	Output
+0	+0	+0
+0	-0	+0
-0	+0	+0
-0	-0	-0

Moreover, test cases that incorporate denormal numbers were examined. Table 10 shows such test cases. Positive denormals are used in these test cases. In addition, test cases with negative denormal numbers were also investigated. Test cases with negative denormals are presented in Table 11. The two most important relationships shown are the following: $(+\infty) + (-\infty) = \text{NaN}$ and $(-\infty) + (+\infty) = \text{NaN}$.

Table 10: Test cases for the addition/subtraction unit with positive denormals

Input1	Input2	Output
+0	+0	+0
+0	Positive Denormal	Positive Denormal
+0	$+\infty$	$+\infty$
+0	NaN	NaN
Positive Denormal	+0	Positive Denormal
Positive Denormal $\lvert x\rvert$	Positive Denormal $\lvert y\rvert$	$(\lvert x\rvert + \lvert y\rvert)$, including rounding
Positive Denormal	$+\infty$	$+\infty$
Positive Denormal	NaN	NaN
$+\infty$	+0	$+\infty$
$+\infty$	Positive Denormal	$+\infty$
$+\infty$	$+\infty$	$+\infty$
$+\infty$	NaN	NaN
NaN	+0	NaN
NaN	Positive Denormal	NaN
NaN	$+\infty$	NaN
NaN	NaN	NaN

Table 11: Test cases for the addition/subtraction unit with negative denormals

Input1	Input2	Output
+0	Negative Denormal	Negative Denormal
+0	-∞	-∞
Positive Denormal	+0	Positive Denormal
Positive Denormal $\|x\|$	Negative Denormal -$\|y\|$	$(\|x\| - \|y\|)$, including rounding
Positive Denormal	-∞	-∞
+∞	-0	+∞
+∞	Negative Denormal	+∞
+∞	-∞	NaN
-∞	+∞	NaN
NaN	-0	NaN
NaN	Negative Denormal	NaN
NaN	-∞	NaN

B.2 Tables for Multiplication

Multiplication with denormal numbers was verified. Positive denormal number handling is examined in the test cases presented in Table 12.

Table 12: Test cases for positive denormal numbers in the multiplication unit

Input1	Input2	Output
$+0$	$+0$	$+0$
$+0$	Positive Denormal	$+0$
$+0$	$+\infty$	NaN
$+0$	NaN	NaN
Positive Denormal	$+0$	$+0$
Positive Denormal $\|x\|$	Positive Denormal $\|y\|$	$(\|x\| * \|y\|)$, including rounding
Positive Denormal	$+\infty$	$+\infty$
Positive Denormal	NaN	NaN
$+\infty$	$+0$	NaN
$+\infty$	Positive Denormal	$+\infty$
$+\infty$	$+\infty$	$+\infty$
$+\infty$	NaN	NaN
NaN	$+0$	NaN
NaN	Positive Denormal	NaN
NaN	$+\infty$	NaN
NaN	NaN	NaN

Furthermore, multiplication with negative denormal numbers was also verified. Table 13 shows some specific cases that have been checked in the multiplication unit.

Table 13: Test cases for the multiplication unit with negative denormals

Input1	Input2	Output
-0	+0	-0
-0	Positive Denormal	-0
-0	$+\infty$	NaN
-0	NaN	NaN
Negative Denormal	+0	+0
Negative Denormal -$\|x\|$	Positive Denormal $\|y\|$	-($\|x\| * \|y\|$), including rounding
Negative Denormal -$\|x\|$	Negative Denormal -$\|y\|$	($\|x\| * \|y\|$), including rounding
Negative Denormal	$+\infty$	$-\infty$
Negative Denormal	NaN	NaN
$-\infty$	+0	NaN
$-\infty$	Positive Denormal	$-\infty$
$-\infty$	$+\infty$	$-\infty$
$-\infty$	NaN	NaN

B.3 Tables for Multiply-Accumulate

For the verification of Multiply-Accumulate, the following table was utilised. Table 14 shows tests using ± 0 and $\pm\infty$. Regarding NaN, when any of the inputs is NaN, then the result is NaN.

Table 14: Test cases for the multiply-accumulate unit utilizing ± 0 $\pm\infty$ and NaN. The multiply-accumulate function is $a \times b + c$.

Input a	Input b	Input c	Output
$+0$	$+0$	$+0$	$(+0) + (+0) = +0$
-0	-0	$+0$	$(+0) + (+0) = +0$
-0	-0	-0	$(+0) + (-0) = +0$
-0	$+0$	$+0$	$(-0) + (+0) = +0$
$+0$	-0	$+0$	$(-0) + (+0) = +0$
$+0$	-0	-0	$(-0) + (-0) = -0$
-0	$+0$	-0	$(-0) + (-0) = -0$
$+\infty$	$+\infty$	$+\infty$	$(+\infty) + (+\infty) = +\infty$
$-\infty$	$-\infty$	$+\infty$	$(+\infty) + (+\infty) = +\infty$
$-\infty$	$-\infty$	$-\infty$	$(+\infty) + (-\infty) = +\infty$
$-\infty$	$+\infty$	$+\infty$	$(-\infty) + (+\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$(-\infty) + (+\infty) = +\infty$
$+\infty$	$-\infty$	$-\infty$	$(-\infty) + (-\infty) = -\infty$
$-\infty$	$+\infty$	$-\infty$	$(-\infty) + (-\infty) = -\infty$

B.4 Tables for Division

The division with denormal numbers has been checked. The following table presents the test cases that were used for ensuring that division works with denormal numbers.

Table 15: Test cases for the division unit utilizing denormals

Numerator	Denominator	Output
Positive Denormal	Positive Number > 1	$+0$
Positive Denormal	Positive Number < 1	Positive Denormal or Positive Number
Negative Denormal	Negative Number > 1	$+0$
Negative Denormal	Negative Number < 1	Positive Denormal or Positive Number
Positive Denormal	Negative Number > 1	$+0$
Positive Denormal	Negative Number < 1	Negative Denormal or Positive Number
Negative Denormal	Positive Number > 1	$+0$
Negative Denormal	Positive Number < 1	Negative Denormal or Positive Number

Next, denormal numbers were examined even further. The following two tables depict certain test cases of denormal numbers with $+0$, -0, $+\infty$, $-\infty$ and with NaN. The most important relationships in the following tables are $\frac{PositiveDenormal}{+0} = +\infty$, $\frac{PositiveDenormal}{-0} = -\infty$, $\frac{NegativeDenormal}{+0} = -\infty$ and $\frac{NegativeDenormal}{-0} = +\infty$. Based on the research paper [11], a popular problem in the division unit is when the result of division is NaN instead of $\pm\infty$. Hence, table 16 shows test cases with positive denormals, ± 0, $\pm\infty$ and NaN. Table 17 shows test cases with negative denormals, ± 0, $\pm\infty$ and NaN.

Table 16: The figure shows test cases for the division unit using positive denormals

Numerator Input	Denominator Input	Output
$+0$	Positive Denormal	$+0$
-0	Positive Denormal	-0
Positive Denormal	$+0$	$+\infty$
Positive Denormal	-0	$-\infty$
$+\infty$	Positive Denormal	$+\infty$
$-\infty$	Positive Denormal	$-\infty$
Positive Denormal	$+\infty$	$+0$
Positive Denormal	$-\infty$	-0
NaN	Positive Denormal	NaN
Positive Denormal	NaN	NaN

Table 17: The figure shows test cases for the division unit using negative denormals

Numerator Input	Denominator Input	Output
$+0$	Negative Denormal	-0
-0	Negative Denormal	$+0$
Negative Denormal	$+0$	$-\infty$
Negative Denormal	-0	$+\infty$
$+\infty$	Negative Denormal	$-\infty$
$-\infty$	Negative Denormal	$+\infty$
Negative Denormal	$+\infty$	-0
Negative Denormal	$-\infty$	$+0$
NaN	Negative Denormal	NaN
Negative Denormal	NaN	NaN

The next test was to check the division of certain normal numbers. This can be observed from the following table. The most important relations in table 18 are $\frac{PositiveNumber}{+0} = +\infty$ and $\frac{PositiveNumber}{-0} = -\infty$. These results should not output NaN.

Table 18: Test cases for the division unit

Numerator	Denominator	Output
Negative Number	Negative Number	Positive Number
Positive Number	Positive Number	Positive Number
Positive Number	Negative Number	Negative Number
Negative Number	Positive Number	Negative Number
$+0$	$+0$	NaN
-0	$+0$	NaN
Negative Number	-0	$+\infty$
$+\infty$	$+0$	$+\infty$
NaN	-0	NaN
-0	Negative Number	$+0$
$-\infty$	Negative Number	$+\infty$
NaN	Positive Number	NaN
$+0$	$-\infty$	-0
Negative Number	$-\infty$	$+0$
$+\infty$	$+\infty$	NaN
NaN	$-\infty$	NaN
-0	NaN	NaN
Positive Number	NaN	NaN
$-\infty$	NaN	NaN
NaN	NaN	NaN
-0	$+0$	NaN
Negative Number	-0	$+\infty$
$+\infty$	$+0$	$+\infty$
NaN	-0	NaN
-0	Negative Number	$+0$
$-\infty$	Negative Number	$+\infty$
NaN	Positive Number	NaN
$+0$	$-\infty$	-0
Negative Number	$-\infty$	$+0$
$+\infty$	$+\infty$	NaN
NaN	$-\infty$	NaN
-0	NaN	NaN
Positive Number	NaN	NaN
$-\infty$	NaN	NaN
NaN	NaN	NaN

B.5 Tables for 2D Dot Product

The following test cases were examined. The test cases below examine the use of $+0$ and -0. The commutativity property of addition is utilised: $(+0) + (-0) = +0$ and $(-0) + (+0) = +0$.

Table 19: Test cases for the 2D Dot Product. The 2D dot product is $a \times d + b \times e$.

Input a	Input d	Input b	Input e	Output
$+0$	$+0$	$+0$	$+0$	$(+0) + (+0) = +0$
-0	-0	-0	-0	$(+0) + (+0) = +0$
-0	-0	$+0$	$+0$	$(+0) + (+0) = +0$
$+0$	$+0$	-0	-0	$(+0) + (+0) = +0$
$+0$	-0	$+0$	-0	$(-0) + (-0) = -0$
-0	$+0$	-0	$+0$	$(-0) + (-0) = -0$
$+0$	$+0$	$+0$	-0	$(+0) + (-0) = +0$
$+0$	$+0$	-0	$+0$	$(+0) + (-0) = +0$
-0	$+0$	$+0$	$+0$	$(-0) + (+0) = +0$
$+0$	-0	$+0$	$+0$	$(-0) + (+0) = +0$

Next, test cases that involve the use of $\pm\infty$ and NaN were used. Table 20 shows these test cases. As before, the commutativity property of addition is utilised: $(+\infty) + (-\infty) = +\infty$ and $(-\infty) + (+\infty) = +\infty$.

Table 20: Further test cases for the 2D Dot Product. The 2D dot product is $a \times d + b \times e$.

Input a	Input d	Input b	Input e	Output
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$(+\infty) + (+\infty) = +\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$(+\infty) + (+\infty) = +\infty$
$-\infty$	$-\infty$	$+\infty$	$+\infty$	$(+\infty) + (+\infty) = +\infty$
$+\infty$	$+\infty$	$-\infty$	$-\infty$	$(+\infty) + (+\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$-\infty$	$(-\infty) + (-\infty) = -\infty$
$-\infty$	$+\infty$	$-\infty$	$+\infty$	$(-\infty) + (-\infty) = -\infty$
$+\infty$	$+\infty$	$+\infty$	$-\infty$	$(+\infty) + (-\infty) = +\infty$
$+\infty$	$+\infty$	$-\infty$	$+\infty$	$(+\infty) + (-\infty) = +\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	$(-\infty) + (+\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$+\infty$	$(-\infty) + (+\infty) = +\infty$
NaN	± 0 or $\pm\infty$	± 0 or $\pm\infty$	± 0 or $\pm\infty$	NaN
± 0 or $\pm\infty$	NaN	± 0 or $\pm\infty$	± 0 or $\pm\infty$	NaN
± 0 or $\pm\infty$	± 0 or $\pm\infty$	NaN	± 0 or $\pm\infty$	NaN
± 0 or $\pm\infty$	± 0 or $\pm\infty$	± 0 or $\pm\infty$	NaN	NaN

B.6 Tables for 3D Dot Product

The following test cases were examined. The test cases in table 21 examine the use of $(+0)$ and (-0). The commutativity property of addition is utilised: $(+0)+(+0)+(-0)=+0$, $(+0)+(-0)+(+0)=+0$ and $(-0)+(+0)+(+0)=+0$. The relations of $(+0)+(-0)+(-0)=(+0)+(-0)=+0$, $(-0)+(+0)+(-0)=(+0)+(-0)=+0$ and $(-0)+(-0)+(+0)=(-0)+(+0)=+0$ are also critical in the table below.

Table 21: Test cases for the 3D Dot Product. The 3D dot product is $a \times d + b \times e + c \times f$.

Input a	Input d	Input b	Input e	Input c	Input f	Output
$+0$	$+0$	$+0$	$+0$	$+0$	$+0$	$(+0)+(+0)+(+0)=+0$
-0	-0	-0	-0	-0	-0	$(+0)+(+0)+(+0)=+0$
-0	-0	$+0$	$+0$	$+0$	$+0$	$(+0)+(+0)+(+0)=+0$
$+0$	$+0$	-0	-0	$+0$	$+0$	$(+0)+(+0)+(+0)=+0$
$+0$	-0	$+0$	-0	$+0$	-0	$(-0)+(-0)+(-0)=-0$
-0	$+0$	-0	$+0$	-0	$+0$	$(-0)+(-0)+(-0)=-0$
$+0$	$+0$	$+0$	$+0$	$+0$	-0	$(+0)+(+0)+(-0)=+0$
$+0$	$+0$	$+0$	$+0$	-0	$+0$	$(+0)+(+0)+(-0)=+0$
$+0$	$+0$	$+0$	-0	$+0$	$+0$	$(+0)+(-0)+(+0)=+0$
$+0$	$+0$	-0	$+0$	$+0$	$+0$	$(+0)+(-0)+(+0)=+0$
$+0$	-0	$+0$	$+0$	$+0$	$+0$	$(-0)+(+0)+(+0)=+0$
-0	$+0$	$+0$	$+0$	$+0$	$+0$	$(-0)+(+0)+(+0)=+0$
$+0$	$+0$	$+0$	-0	$+0$	-0	$(+0)+(-0)+(-0)=+0$
$+0$	-0	$+0$	$+0$	$+0$	-0	$(-0)+(+0)+(-0)=+0$
$+0$	-0	$+0$	-0	$+0$	$+0$	$(-0)+(-0)+(+0)=+0$

Next, test cases that involve the use of $\pm\infty$ and NaN have been used for the verification of the 3D dot product. Table 22 shows test cases for $+\infty$ and $-\infty$. As before, the commutativity property of addition is utilised: $(+\infty)+(+\infty)+(-\infty)=+\infty$, $(+\infty)+(-\infty)+(+\infty)=+\infty$ and $(-\infty)+(+\infty)+(+\infty)=+\infty$. In the table below, the relations of $(+\infty)+(-\infty)+(-\infty)=(+\infty)+(-\infty)=+\infty$, $(-\infty)+(+\infty)+(-\infty)=(+\infty)+(-\infty)=+\infty$ and $(-\infty)+(-\infty)+(+\infty)=(-\infty)+(+\infty)=+\infty$ are also important. Regarding test cases with NaN, when any of the inputs is NaN, then the result should be NaN.

Table 22: Further test cases for the 3D Dot Product. The 3D dot product is $a \times d + b \times e + c \times f$.

Input a	Input d	Input b	Input e	Input c	Input f	Output
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$(+\infty) + (+\infty) + (+\infty) = +\infty$
$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$(+\infty) + (+\infty) + (+\infty) = +\infty$
$-\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$(+\infty) + (+\infty) + (+\infty) = +\infty$
$+\infty$	$+\infty$	$-\infty$	$-\infty$	$+\infty$	$+\infty$	$(+\infty) + (+\infty) + (+\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$+\infty$	$-\infty$	$-\infty$	$(+\infty) + (+\infty) + (+\infty) = +\infty$
$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$(-\infty) + (-\infty) + (-\infty) = -\infty$
$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$(-\infty) + (-\infty) + (-\infty) = -\infty$
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$-\infty$	$(+\infty) + (+\infty) + (-\infty) = +\infty$
$+\infty$	$+\infty$	$+\infty$	$+\infty$	$-\infty$	$+\infty$	$(+\infty) + (+\infty) + (-\infty) = +\infty$
$+\infty$	$+\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$(+\infty) + (-\infty) + (+\infty) = +\infty$
$+\infty$	$+\infty$	$+\infty$	$-\infty$	$+\infty$	$+\infty$	$(+\infty) + (-\infty) + (+\infty) = +\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$(-\infty) + (+\infty) + (+\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$(-\infty) + (+\infty) + (+\infty) = +\infty$
$+\infty$	$+\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$(+\infty) + (-\infty) + (-\infty) = +\infty$
$+\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$(+\infty) + (-\infty) + (-\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$-\infty$	$(-\infty) + (+\infty) + (-\infty) = +\infty$
$-\infty$	$+\infty$	$+\infty$	$+\infty$	$-\infty$	$+\infty$	$(-\infty) + (+\infty) + (-\infty) = +\infty$
$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$+\infty$	$(-\infty) + (-\infty) + (+\infty) = +\infty$
$-\infty$	$+\infty$	$-\infty$	$+\infty$	$+\infty$	$+\infty$	$(-\infty) + (-\infty) + (+\infty) = +\infty$

B.7 Tables for Square Root

Table 23 illustrates some specific corner cases for the function of the square root. The interesting relation in the following table is the following: $\sqrt{-0} = -0$.

Table 23: The figure shows test cases for the square root function

Input	Output
-0	+0
$+\infty$	$+\infty$
-0	-0
Negative Number	NaN
NaN	NaN

The square root unit was also checked with denormals. Table 24 presents such test cases.

Table 24: The table depicts test cases for ensuring that the square root function works as expected

Input	Output
+Denormal	+Denormal
+Denormal	+Normal
+Normal < 1	+Normal < 1
+1	+1
Positive number	Positive number
Max normal	Normal

B.8 Tables for Inverse Square Root

The following table shows the test cases that have been used to verify the inverse square root function.

Table 25: The table depicts test cases for ensuring that the inverse square root function works as expected

Input	Output
Negative number	NaN
+0	$+\infty$
-0	$-\infty$
$+\infty$	+0
$-\infty$	NaN
+Denormal	Positive number
+Normal < 1	Positive number
+1	+1
Positive number	(Positive number) or (Denormal)
Max normal	Normal

B.9 Tables for 2D Euclidean Distance

The following tests have been performed. In the table below, the equations $(-0)^2 = (+0)$ and $(-\infty)^2 = (+\infty)$ have been used.

Table 26: Test cases for the 2D Euclidean Distance. The 2D Euclidean Distance is: $\sqrt{x^2 + y^2}$.

Input x	Input y	Output
$+0$	$+0$	$+0$
$+0$	-0	$+0$
-0	$+0$	$+0$
-0	-0	$+0$
$+0$	$+\infty$	$+\infty$
$+0$	$-\infty$	$+\infty$
-0	$+\infty$	$+\infty$
-0	$-\infty$	$+\infty$
-0	NaN	NaN
$+0$	NaN	NaN
$+\infty$	$+0$	$+\infty$
$+\infty$	-0	$+\infty$
$-\infty$	$+0$	$+\infty$
$-\infty$	-0	$+\infty$
$+\infty$	$+\infty$	$+\infty$
$+\infty$	$-\infty$	$+\infty$
$-\infty$	$+\infty$	$+\infty$
$-\infty$	$-\infty$	$+\infty$
$-\infty$	NaN	NaN
$+\infty$	NaN	NaN

B.10 Tables for 3D Euclidean Distance

The following tests have been performed so as to ensure that the function works with ± 0, $\pm\infty$ and NaN.

Table 27: Test cases for the 3D Euclidean Distance. The 3D Euclidean Distance is: $\sqrt{x^2+y^2+z^2}$.

Input x	Input y	Input z	Output
± 0	± 0	± 0	$+0$
$\pm\infty$	$\pm\infty$	$\pm\infty$	$+\infty$
$\pm\infty$	± 0	± 0	$+\infty$
± 0	$\pm\infty$	± 0	$+\infty$
± 0	± 0	$\pm\infty$	$+\infty$
NaN	± 0	± 0	NaN
± 0	NaN	± 0	NaN
± 0	± 0	NaN	NaN
NaN	$\pm\infty$	$\pm\infty$	NaN
$\pm\infty$	NaN	$\pm\infty$	NaN
$\pm\infty$	$\pm\infty$	NaN	NaN

B.11 Tables for Normalised Vector

Certain results with $+0$ and -0 have been tested. These results can be seen in Table 28.

Table 28: Test cases for the normalised vector function. The normalised Vector is: $\frac{a}{\sqrt{a^2+b^2+c^2}}$.

Input a	Input b	Input c	Output
$+0$	$+0$	$+0$	$\frac{+0}{+0} = \text{NaN}$
$+0$	$+0$	-0	$\frac{+0}{+0} = \text{NaN}$
$+0$	$+0$	$+\infty$	$\frac{+0}{+\infty} = +0$
$+0$	$+0$	$-\infty$	$\frac{+0}{+\infty} = +0$
$+0$	$+0$	NaN	NaN
$+0$	-0	$+0$	$\frac{+0}{+0} = \text{NaN}$
$+0$	-0	-0	$\frac{+0}{+0} = \text{NaN}$
$+0$	-0	$+\infty$	$\frac{+0}{+\infty} = +0$
$+0$	-0	$-\infty$	$\frac{+0}{+\infty} = +0$
$+0$	-0	NaN	NaN

The next step was to test certain results with $+0$, $+\infty$ and -0, $-\infty$. This can be seen in Table 29.

Table 29: Further test cases for the normalised vector function. The normalised Vector is: $\frac{a}{\sqrt{a^2+b^2+c^2}}$.

Input a	Input b	Input c	Output
$+0$	$+\infty$	$+0$	$\frac{+0}{+\infty} = +0$
$+0$	$+\infty$	-0	$\frac{+0}{+\infty} = +0$
$+0$	$+\infty$	$+\infty$	$\frac{+0}{+\infty} = +0$
$+0$	$+\infty$	$-\infty$	$\frac{+0}{+\infty} = +0$
$+0$	$+\infty$	NaN	NaN
$+0$	$-\infty$	$+0$	$\frac{+0}{+\infty} = +0$
$+0$	$-\infty$	-0	$\frac{+0}{+\infty} = +0$
$+0$	$-\infty$	$+\infty$	$\frac{+0}{+\infty} = +0$
$+0$	$-\infty$	$-\infty$	$\frac{+0}{+\infty} = +0$
$+0$	-0	NaN	NaN

In the above tables, the input a is constant. The numerator of the function is input a, and it is equal to $+0$. The same concept can be used for changing the input a to another value. Hence, the next tests for ensuring that the normalised function works correctly involve the use of -0, $+\infty$, $-\infty$ and of NaN in the numerator. Table 30, Table 31 and Table 32 show these test cases. Finally, the last verification check was to use NaN in the numerator. When the input a is NaN, then the result is NaN.

Table 30: Test cases for the normalised vector function. The normalised Vector is: $\frac{a}{\sqrt{a^2+b^2+c^2}}$.

Input a	Input b	Input c	Output
-0	$+\infty$	$+0$	$\frac{-0}{+\infty} = -0$
-0	$+\infty$	-0	$\frac{-0}{+\infty} = -0$
-0	$+\infty$	$+\infty$	$\frac{-0}{+\infty} = -0$
-0	$+\infty$	$-\infty$	$\frac{-0}{+\infty} = -0$
-0	$+\infty$	NaN	NaN
-0	$-\infty$	$+0$	$\frac{-0}{+\infty} = -0$
-0	$-\infty$	-0	$\frac{-0}{+\infty} = -0$
-0	$-\infty$	$+\infty$	$\frac{-0}{+\infty} = -0$
-0	$-\infty$	$-\infty$	$\frac{-0}{+\infty} = -0$
-0	-0	NaN	NaN

Table 31: More test cases for the normalised vector function. The normalised Vector is: $\frac{a}{\sqrt{a^2+b^2+c^2}}$.

Input a	Input b	Input c	Output
$+\infty$	$+\infty$	$+0$	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$+\infty$	-0	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$+\infty$	$+\infty$	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$+\infty$	$-\infty$	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$+\infty$	NaN	NaN
$+\infty$	$-\infty$	$+0$	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$-\infty$	-0	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$-\infty$	$+\infty$	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	$-\infty$	$-\infty$	$\frac{\pm\infty}{+\infty} = \text{NaN}$
$+\infty$	-0	NaN	NaN

Table 32: Further test cases for the normalised vector function. The normalised Vector is: $\frac{a}{\sqrt{a^2+b^2+c^2}}$.

Input a	Input b	Input c	Output
$-\infty$	$+\infty$	$+0$	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$+\infty$	-0	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$+\infty$	$+\infty$	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$+\infty$	$-\infty$	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$+\infty$	NaN	NaN
$-\infty$	$-\infty$	$+0$	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$-\infty$	-0	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$-\infty$	$+\infty$	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	$-\infty$	$-\infty$	$\frac{-\infty}{+\infty} = \text{NaN}$
$-\infty$	-0	NaN	NaN

Appendix C Effect of Verification

Over the duration of the project, several errors and bugs were found and corrected. Figure 5 shows the verification timeline. The x-axis is working days and the y-axis is the percentage of the tests passed. The x-axis is normalised to days after the module in question was written.

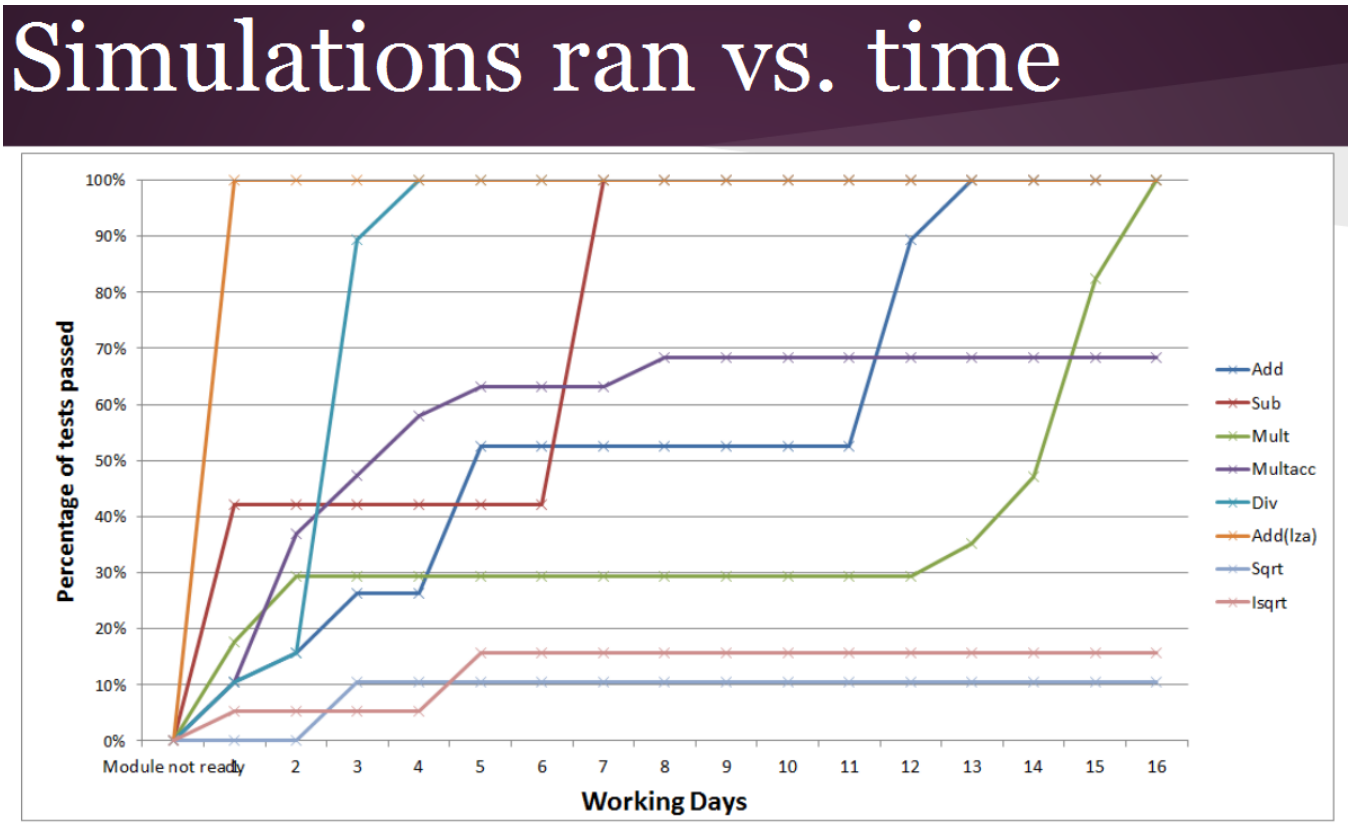

Figure 5: Verification Timeline

Appendix D Formal Verification with Formality

D.1 Verification of Addition and Subtraction using Formality

The formal verification tool Formality could be used to verify certain properties of the addition/subtraction unit. Specifically, Formality could be used to prove the property of commutativity in addition: $x+y = y+x$. Moreover, Formality could be used to prove that if one of the addends is a NaN, then the result is a NaN. Such test cases are presented in the end of Table 10 and Table 11.

Another useful property that could be verified with Formality is $x + y = x$, when x and y have large exponent differences. In particular, when x is a floating-point number with a large exponent and when y is a floating-point number with a small exponent, then the result is equal to the number with the larger exponent (i.e. x).

D.2 Verification of Multiplication using Formality

The formal verification tool Formality could be used to verify the property of commutativity in the multiplication function: $x \times y = y \times x$.

D.3 Verification of 2D Dot Product using Formality

The formal verification tool Formality could be used to verify certain properties of the 2D dot product unit. Specifically, Formality could be utilised to prove the property of commutativity: $x^\intercal y = y^\intercal x$. Furthermore, Formality could be utilised to prove that $a \times d + 0 \times e = a \times d$. In other words, the formal tool could verify that the result is $a \times d$ when $b = 0$.

D.4 Verification of Square Root using Formality

The formal verification tool Formality could be used to verify certain properties of the square root function. In particular, Formality could be utilised to prove the results of Table 24, which tests the square root of denormal numbers. In addition, Formality can also be utilised to test the monotonicity of the square root function. Notably, if $0 \geq a < b$, then $\sqrt{a} < \sqrt{b}$.