Interim Report

for Imagination Technologies

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

Department of Electrical and Electronic Engineering
Imperial College London

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

May 23, 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 In it, we give references and citations whenever we refer to or use the published, or unpublished, work of others.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

Abstract

This report constitutes the Interim Report of the FPU design project. Starting with research, the Group explored different algorithms to implement the floating-point operations that were specified in the Requirements Document. After testing a few, the Group started optimising the designs with respect to accuracy and throughput. In this report, the architecture for every function of the FPU will be examined. The addition/subtraction and multiplication functions have already been created. In this report, the detailed design of the operations that have been created will be presented. Regarding the other operations, a less detailed analysis of the algorithms to be used will be presented.

Moreover, the verification procedure for every function of the FPU will be presented. Verification will ensure that the required accuracy is accomplished. As it was specified in the Requirements Document, the initial step will be to conduct directed testing. The next steps will be constraint-random testing and random testing. In this report, the verification procedure for every function of the FPU will be analyzed. In particular, the directed tests and the constraint random tests for every function will be presented.

Finally, this report will end with the application process. After the verification process, the requirements of latency and throughput will be considered. Efficiency in throughput, latency, area and power is critical when designing the FPU. The application process will be presented in the sections of Timing and Area Analysis and of Power Analysis.

Contents

1. Introduction

1.1 Architectural Exploration

Table 1 shows the functionality of the FPU. In the following sections, the architecture of each of the functions presented below will be examined. Hence, the architectural exploration will be based on the first three columns of the table below.

Table 1: The required operations for the FPU

Opcode[e]	Operation	Mathematical Representation[f]	Accuracy	Throughput (cycles/op)	Latency[g]
0000	No Operation	p	RTE[a]	1	x
0001	Multiply	pq	RTE[a]	1	x
0010	Add	$p+q$	RTE[a]	1	x
0011	Subtract	$p-q$	RTE[a]	1	x
0100	Multiply-Accumulate	$pq+r$	RTE[a]	1	x
0101	Divide	$p \div q$	—[b]	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[d]	3	$3x$
1001	Inverse Square Root	$p^{-1/2}$	4 ulp[d]	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	$\frac{p}{\sqrt{p^2+q^2+r^2}}$	—[b]	nwc[c]	nwc[c]
1101	Unused				
1110	Unused				
1111	Unused				

[a] Round to Nearest, tied to Even
[b] No set accuracy; the Group is to specify how accurate the result will be
[c] No Worse than Chaining constituent operations
[d] Units in Last Place
[e] Opcode decided on an ad-hoc basis; subject to change
[f] p,q,...,u each represents a floating-point number
[g] As yet unspecified, but each operation is measured relatively

Once all functions have been implemented, the design will be inspected to determine the critical path and a pipelined design will be explored. To reduce area along with power consumption, resources can be shared across functions. For functions with independent hardware, scheduling can increase the throughput of the FPU. This will be investigated further, if time permits.

1.1.1 User Interface

The maximum number of inputs required is six, namely for calculating the 3D Dot Product. All functions give one output.

In the C++ implementation, an opcode is required to specify the operation and the relevant number of operands is asked from the user. While testing, these are input as decimal numbers stored as `float` variables. This requires an auxiliary function to convert between `float` and our self-created struct `fp_t`, which stores the sign, exponent and mantissa bits of a floating point number as separate members. When the model is ready for verification, the input method will be changed to accept floating point numbers directly, which will be retained for the final design.

1.2 Verification Architecture

Regarding verification, three main steps will be taken. First, directed testing will be performed. Each directed test targets a specific feature of the design. For example, specifying inputs that could lead to an overflow can be used to validate the overflow detection and correction mechanism. Passing these tests provides a clear indication of our progress. "When dealing with FP verification by simulation, the main challenge is to generate a set of tests that comprises a representative sample of the entire space, taking into account the many corner cases" [18].

Once the design reaches a satisfactory level, the test coverage can be increased by using constrained-random testing. This aims to cover a wide variety of inputs and introduce possible corner cases not thought of while setting functional boundaries to simplify debugging. This method is more efficient since it allows verification of a greater state-space. Table 2 shows some test models that apply to all the functions of the FPU [19]. It also indicates the number of all the possible test cases for each test model. Certain constrained-random tests will be randomly generated for each model. The entire number of test cases for each model is practically impossible to be implemented.

Table 2: Test models to be applied to all functions of the FPU [19]

Description of the test model	**Number of possible test cases**
Floating-point basic types, such as +0, -0, $+\infty$ and $-\infty$	10^4
Floating-point denormals, positive and negative	2×10^7
This test model checks final results that are very close to a specified base value, such as zero or one.	(depends on the number of specified base values)
This test model checks the operation of rounding. It tests possible combinations of the sign bit and the LSB.	2×10^3
This test model checks overflow and near overflow operations	2×10^5
This test model checks underflow and near underflow operations	2×10^5
This test model checks the rounding boundaries. It tests numbers that are on the edge of a rounding boundary.	10^2

When the group is convinced that the design is of a high quality, multiple random tests will be used to ensure exhaustive functional coverage of the design, justifying that the entire design works. Both the `C++` and the `VHDL` code will be verified. The `C++` architecture will be verified using the CBMC tool (C Bounded Model Checker) based on assumptions and assertions. All of the tests on the `VHDL` code will be implemented using testbenches. Code coverage from running simulations on Modelsim will be one of the metrics used to measure the completeness of the tests performed.

The "Golden answer" according to which our design will be verified will be obtained using the arithmetic functions provided in a `VHDL` floating point package by David Bishop[1]. This library has been chosen since it conforms to the IEEE floating-point standard and includes various settings such as rounding-modes, special numbers handling, number of guard-bits etc. For more advance functions such as finding Euclidean Distances, chained-operations will be used to determine the model answer. Special care will be taken to ensure precision is kept. The golden model is subject to change if other libraries or methods are found to be more suitable.

Once verification is completed, the application process takes place. Its purpose is to check that the requirements of accuracy, throughput and latency are satisfied. Different FPU configurations will be considered to discover the best option that minimises power and latency. Table 1 shows the functions of the FPU along with the required accuracy, throughput and latency. In the following sections, the verification procedure for each of the functions presented will be examined.

[1]`http://www.vhdl.org/fphdl/Float_ug.pdf`

2. Addition and Subtraction

2.1 Architecture of Addition and Subtraction

The addition and subtraction functions will be implemented as one entity. The following figure illustrates only the concept of addition. When the exponents of the two numbers to be added are unequal, the smaller exponent is incremented until it equals the larger exponent [1]. At the same time, the mantissa of the first number is bit-shifted to the right by one unit for each exponent increment. Hence, in order to be added, the two floating-point numbers will have the same exponent. Next, the mantissa of the two numbers are summed. Finally, the result is normalized and rounded [1].

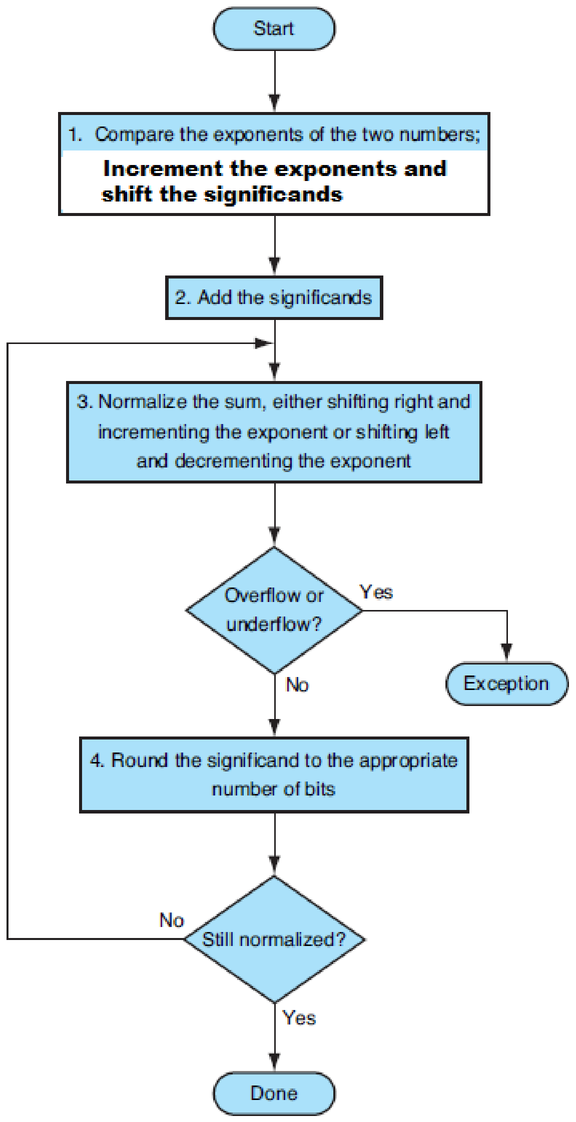
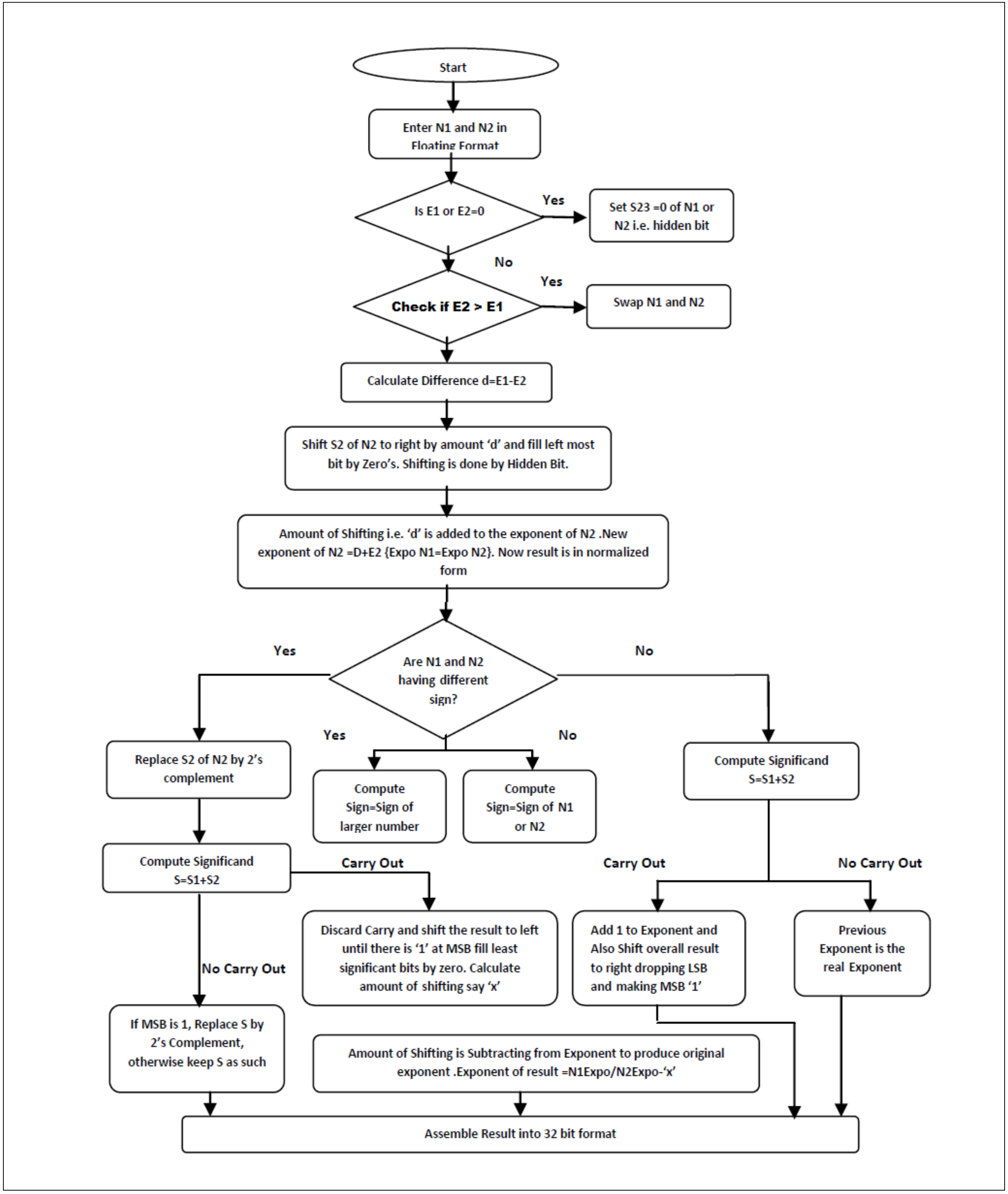

Start
1. Compare the exponents of the two numbers;
Increment the exponents and shift the significands
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

Figure 1: Flowchart for the addition function[2].

There exist many implementations that can be performed in order to utilize the above algorithm. To implement addition and subtraction with the same FP unit, the algorithm needs to be extended. Figure 2 depicts the detailed implementation of the FP addition/subtraction unit. As seen in the flowchart, there are two cases considering the sign of the two floating-point numbers. In the first case, the sign of the two numbers is the same. In the second case, the two numbers have different signs.



Start
Enter N1 and N2 in Floating Format
Is E1 or E2=0
Yes
Set S23 =0 of N1 or N2 i.e. hidden bit
No
Yes
Check if E2 > E1
Swap N1 and N2
Calculate Difference d=E1-E2
Shift S2 of N2 to right by amount 'd' and fill left most bit by Zero's. Shifting is done by Hidden Bit.
Amount of Shifting i.e. 'd' is added to the exponent of N2 .New exponent of N2 =D+E2 {Expo N1=Expo N2}. Now result is in normalized form
Yes
Are N1 and N2 having different sign?
No
Yes
No
Replace S2 of N2 by 2's complement
Compute Sign=Sign of larger number
Compute Sign=Sign of N1 or N2
Compute Significand S=S1+S2
Compute Significand S=S1+S2
Carry Out
Carry Out
No Carry Out
Discard Carry and shift the result to left until there is '1' at MSB fill least significant bits by zero. Calculate amount of shifting say 'x'
Add 1 to Exponent and Also Shift overall result to right dropping LSB and making MSB '1'
Previous Exponent is the real Exponent
No Carry Out
If MSB is 1, Replace S by 2's Complement, otherwise keep S as such
Amount of Shifting is Subtracting from Exponent to produce original exponent .Exponent of result =N1Expo/N2Expo-'x'
Assemble Result into 32 bit format

Figure 2: Flowchart for signed addition [3].

Most research papers outlined algorithms on a subentity gate-level design. The three parts of the floating point numbers will be treated separately, calculating the relevant sign and exponent with logic. When it comes to adding the significands, a fixed point adder is required. As per the agreement with the user, the integer and bitwise operations are acceptable. Hence, even though different adder arrays (such ripple, carry-save and carry-lookahead) can be utilized, the Group will use the built-in integer "+" operator in `C++` and `VHDL`. The ways in which this meet the specification will be analyzed, and a more efficient implementation may be chosen if necessary.

During the normalization stage, a hardware LZD (leading zero detector) is implemented to keep track of the number of leading zeros in the resultant significand after addition. While it brings additional overhead and latency, it was decided that it will suffice the requirement for an initial implementation suitable for testing. One other technique that satisfies the same function is to predict the number of leading zeros in the result directly from the input of adders using a LZA (leading zero anticipator) which works in parallel with the addition. However, this technique requires complex circuitry. The Group will investigate into how these two satisfy the desired functionality while meeting the specification.

2.2 Verification of Addition and Subtraction

In the beginning, directed tests will be used to verify specific properties of addition. Initially test cases will involve positive normal numbers. When the design passes each test model, additional test features such as negative numbers will be added. For example, the fact that the addition of two positive numbers can not be negative will be used as a test case. Moreover, the fact that the addition $x + y$ is the same as the addition of $y + x$ will also be used (i.e. commutativity of addition). This concept of commutativity can be seen in the following table with the addition of $+0$ and -0. The important relationships in Table 3 are that $(+0) + (-0) = +0$ and that $(-0) + (+0) = +0$ so that commutativity holds.

Table 3: 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 will be examined. Table 4 shows such test cases. Positive denormals are used in these test cases. In addition, test cases with negative denormal numbers should also be investigated. Test cases with negative denormals are presented in Table 5. The two most important relationships shown are the following: $(+\infty) + (-\infty) = \text{NaN}$ and $(-\infty) + (+\infty) = \text{NaN}$.

Table 4: 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 5: Test cases for the addition/subtraction unit with negative denormals

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

Cases which lead to overflow and underflow must be checked. In Table 6, test cases for overflow and underflow are presented. The first two cases refer to overflow testing and the last two cases refer to underflow testing.

Table 6: Test cases for checking overflow and underflow

Addition								
Input 1			**Input 2**			**Output**		
Sign	**Significand**	**Exponent**	**Sign**	**Significand**	**Exponent**	**Sign**	**Significand**	**Exponent**
+	1.000000	104	+	1.7FFFFB	127	+	1.7FFFFC	127
+	1.7FB138	33	+	1.7FFFFC	127	+	1.7FFFFC	127
+	0.731A35	-126	-	0.000D18	-126	+	0.730D1D	-126
-	0.074A61	-126	-	0.1C402B	-126	-	0.238A8C	-126

Subsequently, rounding of the result must be tested. Table 7 presents test cases that check that rounding works as expected.

Table 7: Test cases for ensuring that rounding works correctly

Addition								
Input 1			**Input 2**			**Output**		
Sign	**Significand**	**Exponent**	**Sign**	**Significand**	**Exponent**	**Sign**	**Significand**	**Exponent**
+	1.7FFFFD	-93	-	1.000000	-77	-	1.000000	-67
+	1.7FFFF9	-89	+	1.000000	-63	+	1.000000	-63
-	1.07B5CE	-82	+	0.091331	-126	-	1.07B5CE	-82
-	1.000000	-26	+	1.1E6C16	-74	-	1.000000	-26

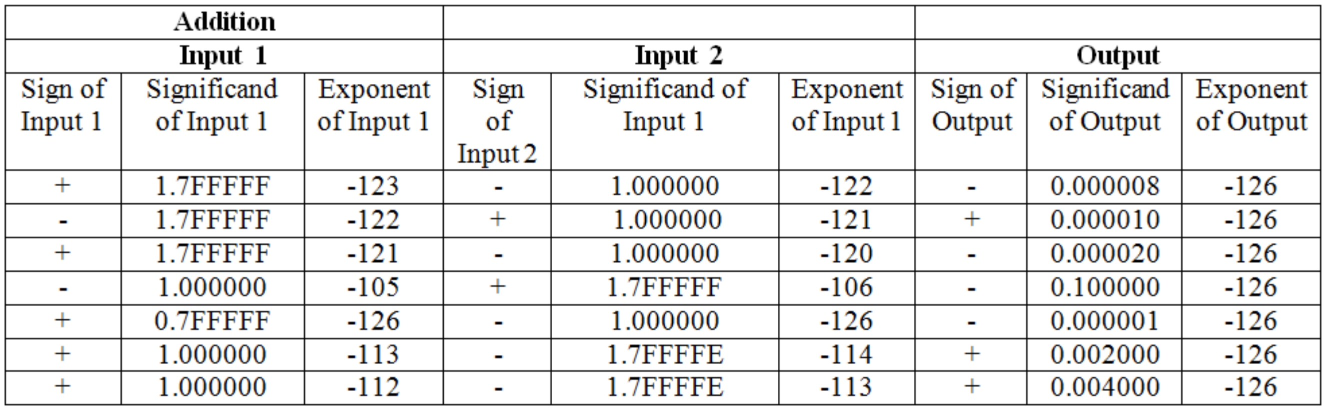
Furthermore, in the verification process of the addition/subtraction unit, the notion of cancellation is critical. Cancellation occurs when the exponent of the result is lower than either of the exponents of the inputs [19]. This occurs when the inputs are relatively close in magnitude. Table 8 shows specific test cases that will be utilized to validate cancellation.

Table 8: Test cases concerning the concept of cancellation in the addition/subtraction unit

Addition								
Input 1			**Input 2**			**Output**		
Sign of Input 1	Significand of Input 1	Exponent of Input 1	Sign of Input 2	Significand of Input 1	Exponent of Input 1	Sign of Output	Significand of Output	Exponent of Output
-	1.7FFFFD	-6	+	1.000000	-5	+	1.400000	-28
+	1.0FF622	-32	-	1.0FF61F	-32	+	1.400000	-54
+	1.7FFFF8	-9	-	1.000002	-8	-	1.400000	-29
+	1.7FFFF8	31	-	1.000005	32	-	1.100000	12
-	1.000055	-104	+	1.7FFFD1	-105	-	1.590000	-121
-	1.7FFBB6	67	+	1.0006B4	68	+	1.0D9000	56
+	1.7FFE6D	-91	-	1.00006E	-90	-	1.1BC000	-105

The concept of cancellation can be extended to denormal numbers. Some of the test cases that will be used for this purpose is presented in Table 9.

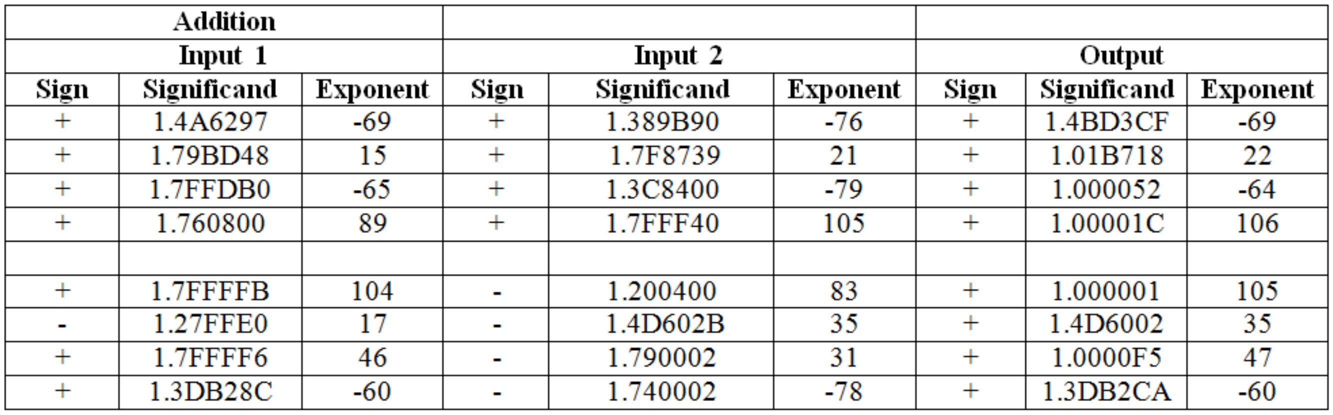
Table 9: Test cases concerning the concept of cancellation in the addition/subtraction unit for denormals



Addition								
Input 1			Input 2			Output		
Sign of Input 1	Significand of Input 1	Exponent of Input 1	Sign of Input 2	Significand of Input 1	Exponent of Input 1	Sign of Output	Significand of Output	Exponent of Output
+	1.7FFFFF	-123	-	1.000000	-122	-	0.000008	-126
-	1.7FFFFF	-122	+	1.000000	-121	+	0.000010	-126
+	1.7FFFFF	-121	-	1.000000	-120	-	0.000020	-126
-	1.000000	-105	+	1.7FFFFF	-106	-	0.100000	-126
+	0.7FFFFF	-126	-	1.000000	-126	-	0.000001	-126
+	1.000000	-113	-	1.7FFFFE	-114	+	0.002000	-126
+	1.000000	-112	-	1.7FFFFE	-113	+	0.004000	-126

The next and final step in the verification process is to check the round bit and the sticky bit. Table 10 shows test cases in order to verify that the sticky bit is calculated correctly. In the tests presented below, the sticky bit should be 0.

Table 10: Test cases checking the round bit and the sticky bit



Addition								
Input 1			Input 2			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent	Sign	Significand	Exponent
+	1.4A6297	-69	+	1.389B90	-76	+	1.4BD3CF	-69
+	1.79BD48	15	+	1.7F8739	21	+	1.01B718	22
+	1.7FFDB0	-65	+	1.3C8400	-79	+	1.000052	-64
+	1.760800	89	+	1.7FFF40	105	+	1.00001C	106
+	1.7FFFFB	104	-	1.200400	83	+	1.000001	105
-	1.27FFE0	17	-	1.4D602B	35	+	1.4D6002	35
+	1.7FFFF6	46	-	1.790002	31	+	1.0000F5	47
+	1.3DB28C	-60	-	1.740002	-78	+	1.3DB2CA	-60

Most of the above verification steps will be applied to the rest of the functions.

3. Multiplication

3.1 Architecture of Multiplication

Multiplication of two floating point numbers can be expressed as

$$\begin{aligned}&((-1)^{s_1} \times 2^{e_1-bias} \times 1.m_1) \cdot ((-1)^{s_2} \times 2^{e_2-bias} \times 1.m_2)\\ =&(-1)^{s_1 \oplus s_2} \times (2^{(e_1+e_2-bias)-bias}) \times (1.m_1 \cdot 1.m_2)\end{aligned}$$

where s = sign, e = exponent and m = mantissa. Hence, the product can be determined from the following

$$s = s_1 \oplus s_2, e = (e_1 + e_2 - bias), 1.m = (1.m_1 \cdot 1.m_2)$$

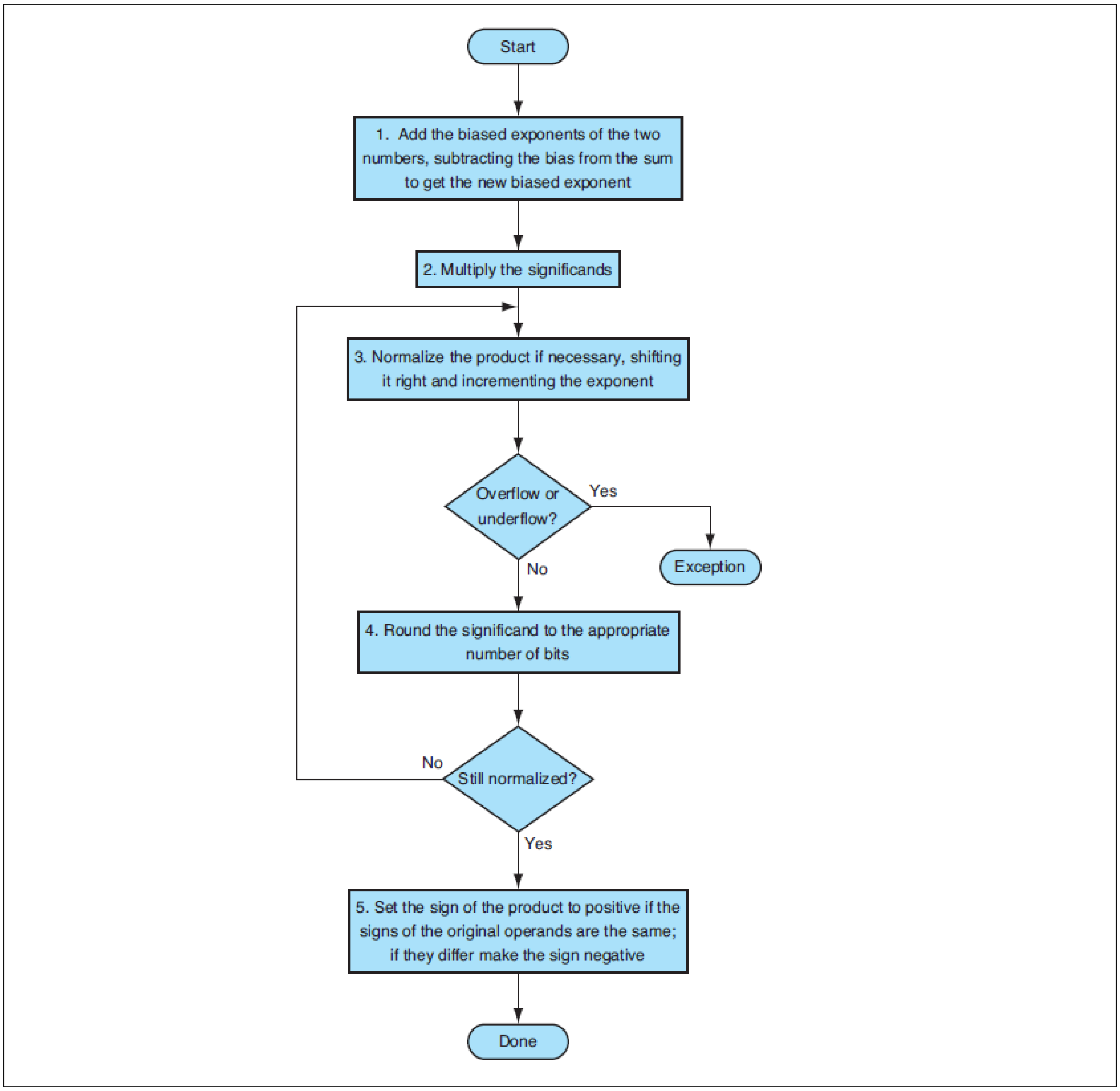

Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Figure 3: Flowchart for the multiplication operation[4].

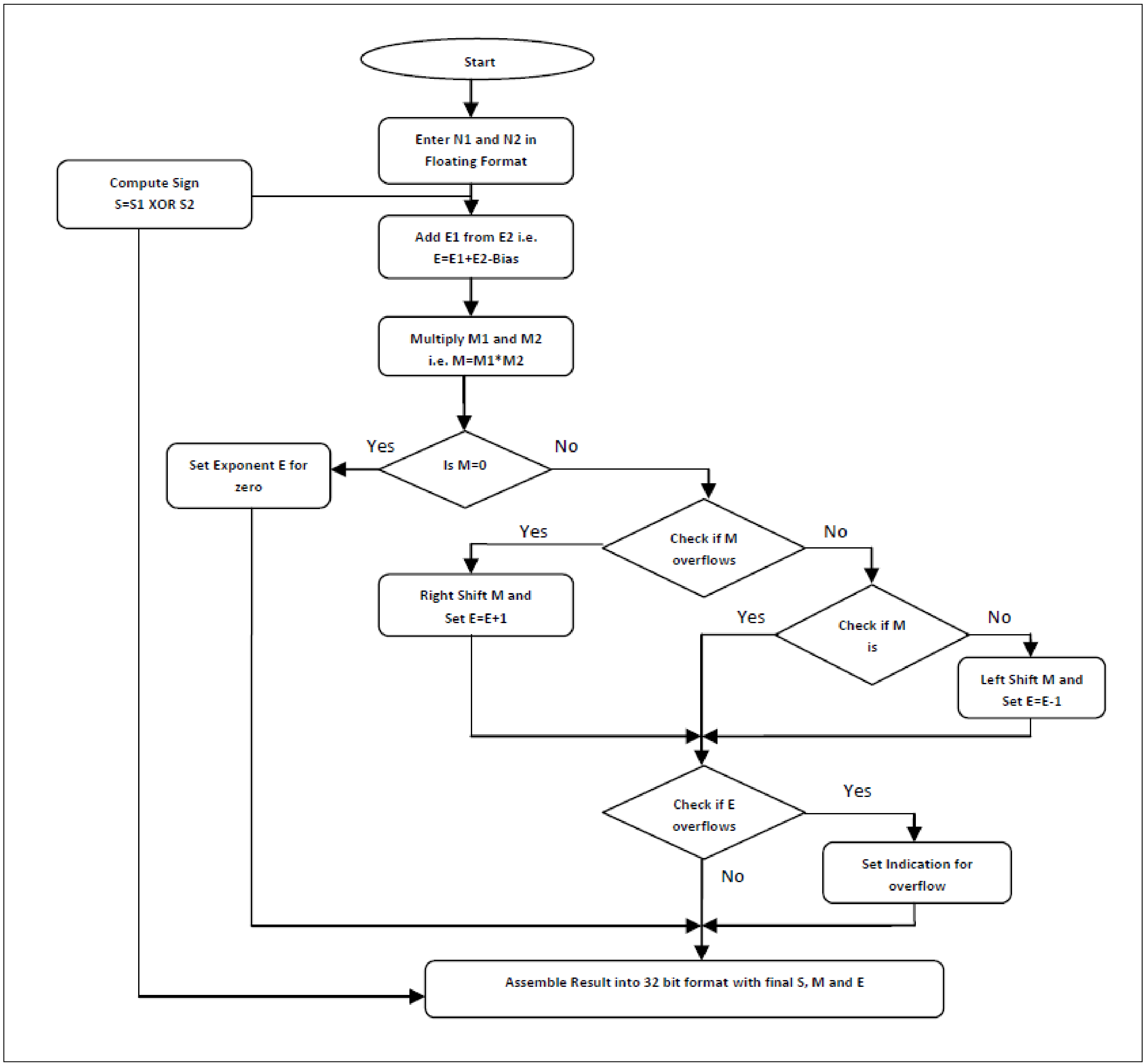

Start
Enter N1 and N2 in Floating Format
Compute Sign S=S1 XOR S2
Add E1 from E2 i.e. E=E1+E2-Bias
Multiply M1 and M2 i.e. M=M1*M2
Yes
Is M=0
No
Set Exponent E for zero
Yes
Check if M overflows
No
Right Shift M and Set E=E+1
Yes
Check if M is
No
Left Shift M and Set E=E-1
Check if E overflows
Yes
Set Indication for overflow
No
Assemble Result into 32 bit format with final S, M and E

Figure 4: Flowchart for the multiplication operation [3].

To implement a floating point multiplier, the sign and the exponent are calculated using normal multiplication standards[2]. Similar to addition, there are built-in integer multiplication functions which is used in our initial naive design. However, since the VHDL "*" operator is not supported in some VHDL synthesis tools, bit-level algorithms for the fixed point multiplication of the significands will be tested. The first, basic bit-wise implementation that was examined is long multiplication; each bit of one operand is AND'd with the other operand to produce a partial product. These are shifted by the relevant amount and summed to give the final product, in an analogous manner to decimal multiplication. This will require 24 additions (for each bit in the significand).

[2]If the signs of the operands are different, the product will be negative; if they are the same, the product will be positive. The exponents of the operands add together under multiplication.

Another method that was examined is splitting each 24-bit significand into two 12-bit numbers, resulting in four 12-bit multiplications. This could be nested, halving the number of bits each time to reduce the size of the multiplications. However, the trade-off here is that each split requires more multiplications and additions.

Booth Encoding is another family of multiplication algorithms. This had the best appeal. In radix-2, one operand's bits would be recoded into +1, 0 or -1. Each +1 would add one the other operand to the product and each -1 would subtract it, after shifting the relevant number of placeholders. Radix-4 instead recodes overlapping three bit intervals of operand into the range 2 to -2, similarly adding or subtracting a multiple of the other operand. Due to three bit recoding, this reduces the number of additions to 12.

One problem which arose in `C++` is that the normal `int` type can only hold 32 bits, while a 24-bit multiplication could produce up a result up to 48 bits. Since only 23 bits are needed, the least significant bits can be effectively thrown away, as long as rounding is performed. Each time the next operand is added, instead of shifting up, the product is instead shifted down to preserve the number of bits and to keep the size minimal. A `long int` could be used, but this method is more efficient for memory storage.

3.2 Verification of Multiplication

The first check is to test the commutativity of multiplication: $x \times y = y \times x$. Next, specific test cases are used so that directed testing is performed. Table 11 and Table 12 show specific test cases that check the process of rounding with Table 12 aiming to perform rounding boundaries check.

Table 11: The table illustrates test vectors in order to check rounding

Multiply								
Input 1			Input 2			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent	Sign	Significand	Exponent
+	1.100000	89	-	1.07F1F4	-67	-	1.18F032	22
+	1.6AE000	116	+	1.525C00	-60	+	1.410028	57
-	1.30000	-125	+	1.7DB928	64	-	1.2E6F4C	-60
+	1.25E000	-41	+	1.286200	-41	+	1.5A3500	-26

Table 12: The table depicts test vectors in order to check the boundaries of rounding

Multiply								
Input 1			Input 2			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent	Sign	Significand	Exponent
+	1.73876B	-119	-	1.116D43	124	-	1.0A57A4	6
+	1.695161	-22	+	1.562542	38	+	1.432BFD	17
+	1.53B053	-5	+	1.723B91	37	+	1.484DF1	33
-	1.73D6BF	-116	-	1.2566C1	76	+	1.1D8B49	-39

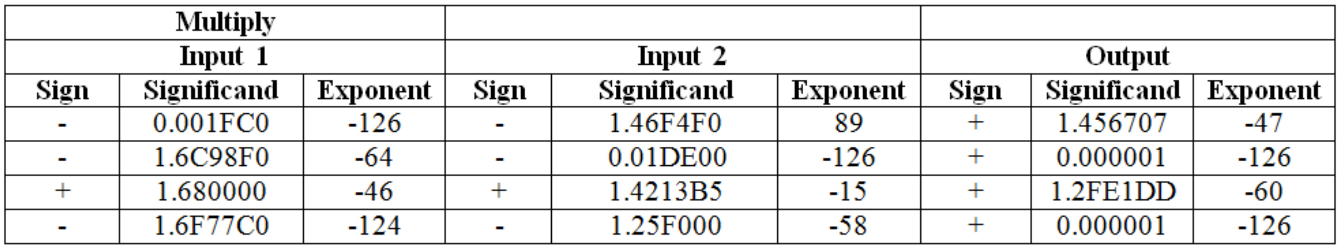
The next step is to consider overflow and underflow. Table 13 presents test cases for ensuring that overflow and underflow do not occur. The first two test cases refer to overflow, and the last two test cases refer to underflow.

Table 13: The table shows test vectors that ensure that overflow and underflow do not occur

Multiply								
Input 1			Input 2			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent	Sign	Significand	Exponent
-	1.0A2000	79	+	1.6D3C00	48	-	1.7FFFFF	127
-	1.5F2000	79	-	1.12DC00	48	+	1.7FFFFF	127
+	1.000000	-64	+	1.7FFFFE	-63	+	0.7FFFFF	-126
-	1.000000	-71	-	1.000000	-55	+	1.000000	-126

Moreover, the sticky bit needs to be checked in the multiplication function. Table 14 shows test cases that will be utilized to check that the sticky bit calculation is correct.

Table 14: Test cases to check correct sticky bit calculation



Multiply								
Input 1			Input 2			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent	Sign	Significand	Exponent
-	0.001FC0	-126	-	1.46F4F0	89	+	1.456707	-47
-	1.6C98F0	-64	-	0.01DE00	-126	+	0.000001	-126
+	1.680000	-46	+	1.4213B5	-15	+	1.2FE1DD	-60
-	1.6F77C0	-124	-	1.25F000	-58	+	0.000001	-126

Positive denormal numbers handling are examined in the test cases presented in Table 15.

Table 15: 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 $\lvert x\rvert$	Positive Denormal $\lvert y\rvert$	$(\lvert x\rvert * \lvert y\rvert)$, including rounding
Positive Denormal	$+\infty$	$+\infty$
Positive Denormal	NaN	NaN
$+\infty$	+0	NaN
$+\infty$	Positive Denormal	$+\infty$
$+\infty$	$+\infty$	$+\infty$
$+\infty$	NaN	NaN
NaN	+0	NaN
NaN	Positive Denormal	NaN
NaN	$+\infty$	NaN
NaN	NaN	NaN

The next and final check is to test negative denormal numbers. Table 16 below shows some specific cases that need to be checked in the multiplication unit.

Table 16: 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

4. Multiply-Accumulate

4.1 Architecture of Multiply-Accumulate

The Fused Multiply Add (FMA) operation will be examined as a technique of implementing the multiply-accumulate function. Using FMA improves the performance and accuracy of the FPU since the rounding operation is performed only one time [8]. This also improves the latency of the system.

For an initial implementation, a chained implementation to test the functionality of the design will be used, once the verification of addition and multiplication has been completed. The chained implementation will be used as a baseline for benchmarking. To improve performance and latency, the initial normalization stage of addition can be done in parallel with multiplication. A single rounding and normalization will operate on the fused result. The FMA algorithm will be analyzed in detail once addition and multiplication modules have been sufficiently optimized.

4.2 Verification of Multiply-Accumulate

In the beginning, the process of rounding will be tested. Table 17 shows test cases that ensure that rounding is correctly performed.

Table 17: Test cases to ensure rounding works correctly

Multiply-Accumulate a*b + c											
Input a			Input b			Input c			Output		
Sign	Significand	e	Sign	Significand	e	Sign	Significand	e	Sign	Significand	e
-	1.120000	105	+	1.54F049	-39	+	1.7268E0	66	-	1.726688	57
+	1.278448	29	+	1.640000	-54	+	1.3B2B2F	-27	+	1.2C9736	-24
+	1.7FFF8B	-81	-	1.000000	-35	-	0.02AC00	-126	-	1.00001B	-115
-	1.614000	-99	-	1.54570A	116	+	1.47BB9C	7	+	1.3AEE8D	18

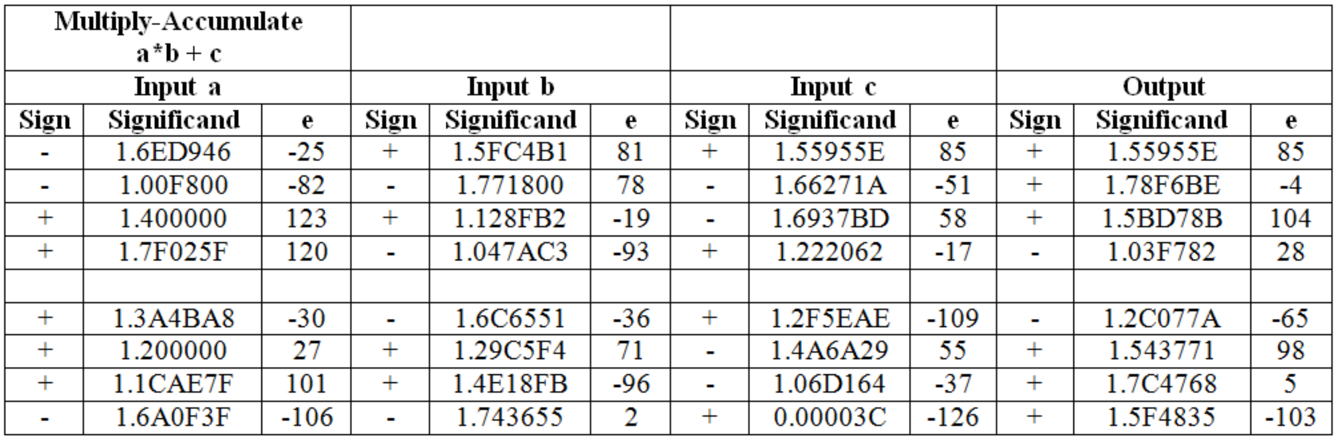
Moreover, overflow and underflow should be checked in the Multiply-Accumulate function. Table 18 depicts test cases that check overflow and underflow. The first two test cases refer to overflow, and the last two test cases refer to underflow.

Table 18: Test cases where overflow and underflow should not occur

Multiply-Accumulate a*b + c											
Input a			Input b			Input c			Output		
Sign	Significand	e	Sign	Significand	e	Sign	Significand	e	Sign	Significand	e
+	1.0ABD10	72	+	1.000000	52	+	1.6EA85A	127	+	1.7FFFFC	127
+	1.4A2800	91	+	1.267000	36	-	1.5B98C0	122	+	1.7FFFFD	127
+	1.7B6503	-117	-	1.36AC00	4	+	1.3364B8	-112	+	0.7FFFFF	-126
-	1.600000	-70	+	1.0E2BC0	-52	+	1.006648	-121	+	1.000000	-126

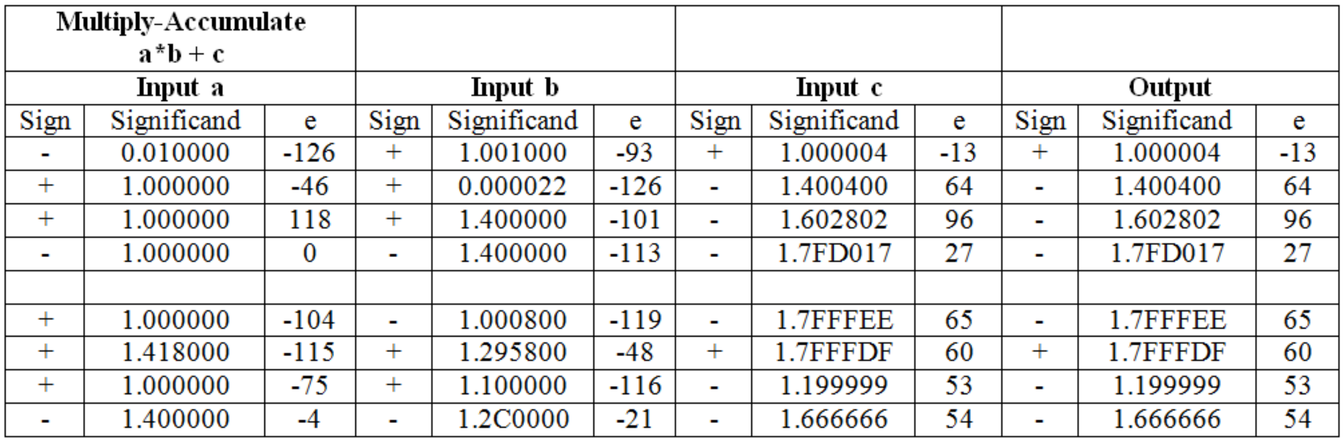
Next, the shift between the addends of the multiply-add operation should be checked. Table 19 and 20 present test cases in order to ensure that the shift works correctly.

Table 19: Test cases for ensuring that shift works as expected in the Multiply-Accumulate unit



Multiply-Accumulate a*b + c											
Input a			Input b			Input c			Output		
Sign	Significand	e	Sign	Significand	e	Sign	Significand	e	Sign	Significand	e
-	1.6ED946	-25	+	1.5FC4B1	81	+	1.55955E	85	+	1.55955E	85
-	1.00F800	-82	-	1.771800	78	-	1.66271A	-51	+	1.78F6BE	-4
+	1.400000	123	+	1.128FB2	-19	-	1.6937BD	58	+	1.5BD78B	104
+	1.7F025F	120	-	1.047AC3	-93	+	1.222062	-17	-	1.03F782	28
+	1.3A4BA8	-30	-	1.6C6551	-36	+	1.2F5EAE	-109	-	1.2C077A	-65
+	1.200000	27	+	1.29C5F4	71	-	1.4A6A29	55	+	1.543771	98
+	1.1CAE7F	101	+	1.4E18FB	-96	-	1.06D164	-37	+	1.7C4768	5
-	1.6A0F3F	-106	-	1.743655	2	+	0.00003C	-126	+	1.5F4835	-103

Table 20: Test cases for ensuring that shift works as expected with special significands



Multiply-Accumulate a*b + c											
Input a			Input b			Input c			Output		
Sign	Significand	e	Sign	Significand	e	Sign	Significand	e	Sign	Significand	e
-	0.010000	-126	+	1.001000	-93	+	1.000004	-13	+	1.000004	-13
+	1.000000	-46	+	0.000022	-126	-	1.400400	64	-	1.400400	64
+	1.000000	118	+	1.400000	-101	-	1.602802	96	-	1.602802	96
-	1.000000	0	-	1.400000	-113	-	1.7FD017	27	-	1.7FD017	27
+	1.000000	-104	-	1.000800	-119	-	1.7FFFEE	65	-	1.7FFFEE	65
+	1.418000	-115	+	1.295800	-48	+	1.7FFFDF	60	+	1.7FFFDF	60
+	1.000000	-75	+	1.100000	-116	-	1.199999	53	-	1.199999	53
-	1.400000	-4	-	1.2C0000	-21	-	1.666666	54	-	1.666666	54

Furthermore, possible problems that could arise from cancellation should be checked. The concept of cancellation, which was presented in the addition/subtraction verification process, will also be used in the multiply-accumulate unit. Cancellation describes a situation whereby the exponent of the result is lower than either of the exponents of the addends. This occurs when the addends are relatively close in magnitude [19]. Table 21 illustrates test cases for the concept of cancellation in the Multiply-Accumulate function [33] and Table 22 presents test cases regarding cancellation with denormal numbers.

Table 21: Test cases where cancellation occurs

Multiply-Accumulate a*b + c											
Input a			Input b			Input c			Output		
Sign	Significand	e	Sign	Significand	e	Sign	Significand	e	Sign	Significand	e
+	1.402957	-114	+	1.2AD267	1	-	1.003963	-112	+0		
+	1.4FB72D	31	+	1.5211EF	16	-	1.2A72D0	48	+	1.400000	2
-	1.6A4927	-10	-	1.49E1DF	-90	-	1.38C226	-99	-	0.000038	-126
+	1.235107	106	+	1.5F7EB9	-29	-	1.0E946D	78	+	1.700000	34
+	1.62EF21	70	-	1.356283	-88	+	1.20CA66	-17	+	1.680000	-60
+	1.57AE50	-1	-	1.3018C7	107	+	1.145CB7	107	-	1.400000	65
-	1.0573B5	60	+	1.7AE9FF	64	+	1.02CCFA	125	+	1.5A8000	86
+	1.2033E7	30	+	1.73B48D	23	-	1.188241	54	+	1.0EC000	16

Table 22: Test cases cancellation occurs with denormal numbers

Multiply-Accumulate a*b + c											
Input a			Input b			Input c			Output		
Sign	Significand	e	Sign	Significand	e	Sign	Significand	e	Sign	Significand	e
+	1.482D81	-109	-	1.6E1281	6	+	1.3A28C6	-102	-	0.000001	-126
-	1.22BE57	-99	+	1.62F299	-3	+	1.104642	-101	+	0.000002	-126
-	1.21384B	-43	-	1.56D163	-58	-	1.0748EC	-100	+	0.000004	-126
-	1.531E63	1	+	1.3BACB5	-101	+	1.1AC5A0	-99	+	0.000008	-126
+	1.5347A1	-70	-	1.345C61	-29	+	1.14DA9B	-98	-	0.000010	-126
-	1.2E92CB	14	-	1.5D3D1D	-112	-	1.16DE66	-97	-	0.000020	-126

Moreover, there are special corner cases that need to be considered and tested. Cases in which the multiplication causes some event in the product while the addition cancels this event should be examined [19]. Notably, such an event takes place when overflow/underflow occurs in the product ab, but in the addition $ab + c$ the overflow/underflow is not apparent. Tables 23, 24 and 25 show test cases that check different possible combinations of inputs and reveal these special cases. Notably, Table 24 examines special cases near overflow and Table 25 examines special cases near underflow.

Table 23: Test cases for ensuring that special corner cases work as expected

Multiply-Accumulate a*b + c											
Input a			**Input b**			**Input c**			**Output**		
Sign	**Significand**	**e**	**Sign**	**Significand**	**e**	**Sign**	**Significand**	**e**	**Sign**	**Significand**	**e**
+	1.2B9463	-65	+	1.732C93	46	-	1.22FBBA	-18	+	1.636400	-52
+	1.2B2CFE	-58	+	1.400000	26	-	1.001010	-30	-	1.7F7CC3	-32
-	1.514EFC	62	-	1.7D4000	1	-	1.500680	64	-	1.771D35	56
+	1.27F400	64	+	1.3AB000	-47	-	1.75F800	17	-	1.014040	10
-	1.7AC000	-96	-	1.479525	-13	-	1.437D56	-108	-	0.000900	-126
+	1.440E69	-16	+	1.6CADCB	118	-	1.448600	70	+	1.354262	103
+	1.2A8300	-31	-	1.27C000	0	-	1.000000	-55	-	1.5F76AF	-31
-	1.22FF8F	92	-	1.5DCCF9	0	+	1.38F3D2	68	+	1.0D3921	93
+	1.3EB2B6	38	-	0.00017C	-126	-	1.6BCB76	-109	-	1.0F603A	-102
+	1.091A40	38	-	1.6F0000	38	+	1.000016	77	+	1.144000	61
+	1.47BB75	-16	+	1.240F44	2	-	1.000000	-13	-	1.167B00	-43

Table 24: Test cases for ensuring that special corner cases near overflow work as expected

Multiply-Accumulate a*b + c											
Input a			**Input b**			**Input c**			**Output**		
Sign	**Significand**	**e**	**Sign**	**Significand**	**e**	**Sign**	**Significand**	**e**	**Sign**	**Significand**	**e**
-	1.118E00	76	-	1.612000	51	-	1.6570D6	127	+	1.54794C	124
+	1.000000	113	+	1.000000	15	-	1.4A6540	127	+	1.566B00	125
-	1.000000	66	+	1.000000	62	+	1.75D819	127	-	1.227E70	123
+	1.164000	7	-	1.5A1700	120	+	1.778524	127	-	1.07ADB8	123
-	1.500000	10	+	1.12AD70	118	+	1.7B3918	127	-	1.617A94	127

Table 25: Test cases for ensuring that special corner cases near underflow work as expected

Multiply-Accumulate a*b + c											
Input a			**Input b**			**Input c**			**Output**		
Sign	**Significand**	**e**	**Sign**	**Significand**	**e**	**Sign**	**Significand**	**e**	**Sign**	**Significand**	**e**
-	1.6F8000	-91	-	1.08D180	-59	+	1.5E81A8	-68	+	1.5E81A8	-68
-	1.2330F7	-69	-	1.48CB88	-81	-	1.458FB2	115	-	1.458FB2	115
+	1.7C0000	-46	+	1.020820	-104	+	1.667D0C	67	+	1.667D0C	67
+	1.0FDE09	-73	+	1.63C3F1	-77	+	1.68DE07	84	+	1.68DE07	84
-	1.3A4BA8	-116	-	1.6C6551	-36	+	1.03412E	-49	+	1.03412E	-49
-	1.62AE4B	-97	+	1.108E40	-53	-	1.44653E	62	-	1.44653E	62
-	1.080000	-126	+	1.70F0F0	-24	-	1.667709	91	-	1.667709	91
-	1.4D0432	-24	+	1.1FD4C4	-126	+	1.743655	-30	+	1.743655	-30

5. Division

5.1 Architecture of Division

Similar to multiplication, division of two floating point numbers can be expressed as

$$((-1)^{s_1} \times 2^{e_1 - bias} \times 1.m_1) \div ((-1)^{s_2} \times 2^{e_2 - bias} \times 1.m_2)$$
$$=(-1)^{s_1 \oplus s_2} \times (2^{(e_1 - e_2 + bias) - bias}) \times \left(\frac{1.m_1}{1.m_2}\right)$$

where s = sign, e = exponent and m = mantissa. Hence, the quotient can be determined from the following

$$s = s_1 \oplus s_2, e = (e_1 - e_2 + bias), 1.m = \left(\frac{1.m_1}{1.m_2}\right)$$

Figure 5 shows the general flowchart for the floating point division operation. As with multiplication, the sign bit and exponent of the quotient can be obtained with normal bitwise and basic add/subract operations as investigated in previous sections. For the mantissa calculation, there exist many different algorithms to perform fixed point division. These algorithms can be separated into the three categories of digit recurrence, functional iteration and approximation algorithms [5]. One of these categories, the functional iteration, will be examined further in the following two sections.

5.1.1 Functional Iteration - The Newton-Raphson Method

Firstly, the Group examined the division by functional iteration algorithms, specifically the Newton-Raphson method. Division-through-Multiplication underpins this method because a strict division can be converted into the product of a number and a reciprocal: $\frac{x}{y} = x \times \frac{1}{y}$. The Newton-Raphson iterative method can be used to approximate $\frac{1}{y}$ by finding a numerical solution to the equation $f(z) = \frac{1}{z} - y = 0$. The following equation arises for approximating $\frac{1}{y}$ [24]:

$$z_{i+1} = z_i \times (2 - (z_i \times y)), \text{ where i is the current iteration number.}$$

From the above equation, it can be seen that the algorithm performs two multiplications and one addition/ subtraction in every iteration[3]. The equation for the error is $err_i = \frac{1}{y} - z_i$ and, hence, $err_{i+1} = y \times (err_i)^2$. Thus, it is observed that the error decreases quadratically with every iteration i. Latency depends on the number of iterations. Furthermore, from the above equation, it can be seen that the Newton-Raphson method depends on the initial approximation z_0. When a more accurate starting approximation is used, the total number of iterations required to achieve a certain precision is reduced [25]. The final approximation converges to the correct result of $\frac{1}{y}$ only if the initial approximation z_0 satisfies $0 < z_0 < \frac{2}{y}$ [28].

For the initial low-precision approximation z_0, a look-up table can be utilized. However, this method will not be used due to the problems that are associated with ROM in hardware. Specifically, the look-up table would increase the delay of the design. The off-chip communication delay between the FPU and the ROM would slow the system. Moreover, the ROM's size would grow exponentially if one extra bit of accuracy is needed. Instead, another technique that can be utilized for the initial value z_0 is linear approximation [31].

[3]This could also be interpreted as one multiply-accumulate operation and one standard multiplication

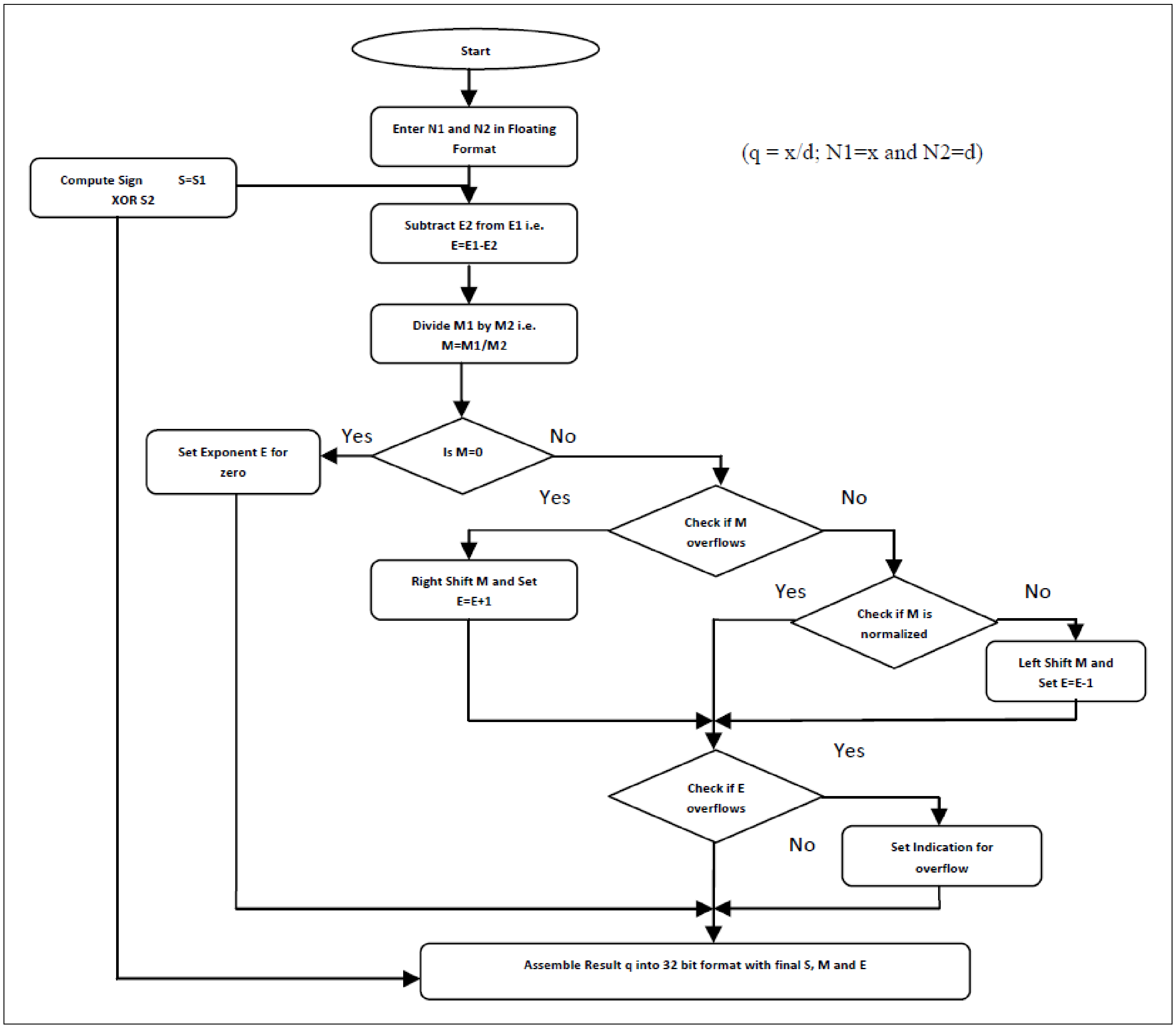

Start
Enter N1 and N2 in Floating Format
(q = x/d; N1=x and N2=d)
Compute Sign S=S1 XOR S2
Subtract E2 from E1 i.e. E=E1-E2
Divide M1 by M2 i.e. M=M1/M2
Yes
Is M=0
No
Set Exponent E for zero
Yes
Check if M overflows
No
Right Shift M and Set E=E+1
Yes
Check if M is normalized
No
Left Shift M and Set E=E-1
Yes
Check if E overflows
No
Set Indication for overflow
Assemble Result q into 32 bit format with final S, M and E

Figure 5: Flowchart diagram for the division operation [3].

The Newton-Raphson method can be implemented with the Multiply-Accumulate structure, which was presented in Section 4.. The algorithm requires a floating-point multiplication after approximating $\frac{1}{y}$ in order to evaluate $\frac{x}{y}$. However, this can cause problems in the process of rounding. To obtain a correctly rounded result, both the result of approximating $\frac{1}{y}$ and the result of the floating-point multiplication $x \times \frac{1}{y}$ need to be correctly rounded.

5.1.2 Functional Iteration - Goldschmidt's Algorithm

An alternative functional iteration algorithm is Goldschmidts algorithm, which is related to the Taylor series. In particular, Goldschmidts algorithm uses the Taylor series of $\frac{1}{1+s}$ around 0 [25]. Therefore, the division $\frac{x}{y}$ is approximated by $x \times \frac{1}{y} = x \times g(s)$, where $g(s) = \frac{1}{1+s} = 1 - s + s^2 - s^3 + s^4 - ...$ and $\frac{1}{y} = g(s) = \frac{1}{1+s}$. Hence, $y = 1 + s$.

As a result of $x \times \frac{1}{y} = x \times (1 - s + s^2 - s^3 + s^4 - ...)$, the following equation is engendered [25].

$$x \times \frac{1}{y} = x \times [(1 - s) \times (1 + s^2) \times (1 + s^4) \times (1 + s^8)...]$$

The equations of the algorithm depends on the ith iteration. Therefore: $\frac{x}{y} = \frac{a_i}{b_i}$. The main idea is to make b_i converge to 1. Thus, a_i will converge to the correct answer of the division function. The following two equations arise. The term r_i is used, for which $r_{i+1} = 2 - b_{i+1}$ and $r_0 = 2 - y$.

$$a_{i+1} = a_i \times r_i$$
$$b_{i+1} = b_i \times r_i$$

Goldschmidts algorithm does two multiplications and one addition/subtraction in each iteration. The number of total iterations required depends on the precision needed in the result. The Newton-Raphson method, which was presented in the previous section, uses the same number of basic operations, but the two techniques are not the same. They quadratically converge to different solutions.

On the one hand, the Newton-Raphson method approximates $\frac{1}{y}$. Hence, for computing $\frac{x}{y}$, floating-point multiplication is needed after the Newton-Raphson method. This is the main disadvantage of this method. On the other hand, Goldschmidts algorithm approximates $\frac{x}{y}$ directly. A floating-point multiplication is not needed after the iterative algorithm is performed. Goldschmidts algorithm reorders the operations in the Newton-Raphson method in order to reduce latency and to increase parallelism [27].

In this report, the Newton-Raphson method was preferred due to its self-correcting nature. In the Newton-Raphson method, the error does not propagate with each iteration. In other words, later iterations of the algorithm correct the errors from the preceding iterations [25]. On the contrary, Goldschmidts algorithm is not self-correcting and the error propagates through each iteration. Since the accuracy of the result is of primary concern in this project, the Newton-Raphson method was preferred.

5.2 Verification of Division

The initial verification check is to ensure that the code does not involve a comparison check to 0 before computing a division [15]. The comparison check will not avoid the generation of infinite results. In this concept, the notion of a floating point number is critical. When $\frac{1}{x}$ is evaluated and x is a floating-point number close to zero, then overflow will arise at some point in the program. In this case, the comparison check to 0 will not prevent the overflow.

Next, division with denormal numbers must be checked. The following table presents the test cases that will be used for ensuring that division works with denormal numbers.

Table 26: 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 are examined even further. The following table depicts certain test cases of denormal numbers with $+0$, -0, $+\infty$, $-\infty$ and with NaN.

Table 27: The figure shows test cases for the division unit using denormals examined even further

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
+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 is to check the division of certain normal numbers. This can be observed from the following table.

Table 28: 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

6. 2D Dot Product

6.1 Architecture of 2D Dot Product

In this section, the following vectors x and y will be used.

$$x = \begin{pmatrix} a \\ b \end{pmatrix} \qquad y = \begin{pmatrix} d \\ e \end{pmatrix}$$

The dot-product of two vectors is defined as $x^{\tau}y = (a \times d) + (b \times e)$. One technique to evaluate the dot-product is to use a pipeline that will simultaneously execute the following three steps [30]. Let x_i and y_i be the elements of the vectors x and y respectively. In this case, i takes values from 1 up to 2. The Long Accumulator is a fixed-point register in which the final result of the scalar product is built up [30].

1) input the two factors x_i and y_i,
2) compute x_i * y_i, and
3) add the product from (2) to the Long Accumulator.

The following figure depicts the way in which the 2D dot-product will be evaluated [38]. The structure will utilize the multiplication unit and the addition/subtraction unit.

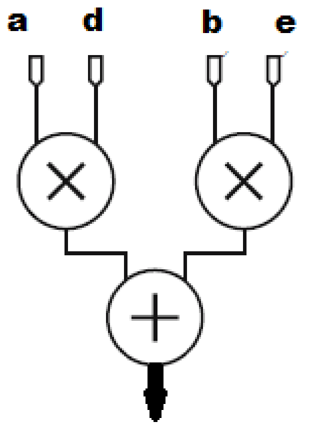


Figure 6: The figure presents the structure of the 2D dot-product.

6.2 Verification of 2D Dot Product

The initial check is to ensure that $x^{\tau}y = y^{\tau}x$. Next, the verification test vectors will be identical to the test vectors used for addition and for multiplication. Apart from these test cases, ill-conditioned dot products should also be considered [29]. In ill-conditioned vectors, a small error in the value of the input causes a large error in the value of the output. Ill-conditioned vectors have a large condition number, where the condition number is defined as $c = 2 \times \frac{|x^{\tau}| \times |y|}{|x^{\tau} \times y|}$.

If x_i and y_i are the components of the vectors x and y respectively, then an alternative representation of the condition number can be formulated. In terms of the dot product, the condition number is also defined as $c = \frac{2 \times \sum_{i=1}^{n} |x_i \times y_i|}{|\sum_{i=1}^{n} x_i \times y_i|}$. In this case, n is 2. Hence, $c = \frac{2 \times \sum_{i=1}^{2} |x_i \times y_i|}{|\sum_{i=1}^{2} x_i \times y_i|} = \frac{2 \times (|a \times d| + |b \times e|)}{|a \times d + b \times e|}$.

7. 3D Dot Product

7.1 Architecture of 3D Dot Product

In this section, the following vectors x and y will be used.

$$x = \begin{pmatrix} a \\ b \\ c \end{pmatrix} \qquad y = \begin{pmatrix} d \\ e \\ f \end{pmatrix}$$

The dot-product of two vectors is defined as $x^T y = a \times d + b \times e + c \times f$. In this case, three terms are used in the addition process. Based on summation theory, in order to improve the accuracy in a summation with many terms, the addends should be sorted. However, due to the existence of only three terms in the addition, sorting will not be performed.

The following figure depicts the way in which the 3D dot-product will be evaluated [38]. The structure will utilize the multiplication unit and the addition/subtraction unit.

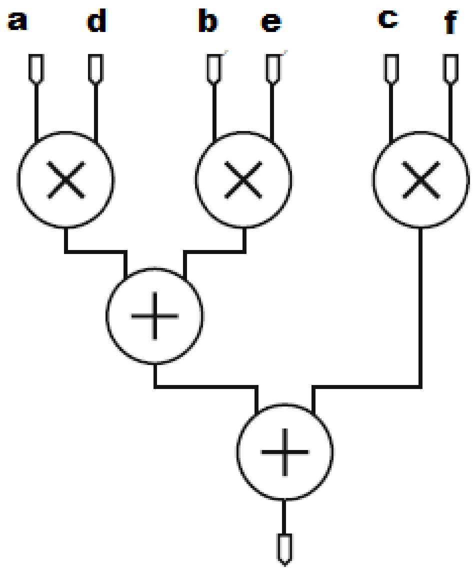


Figure 7: The figure presents the structure of the 3D dot-product.

7.2 Verification of 3D Dot Product

The verification process for the 3D dot product is identical to the 2D dot product.

8. Square Root

8.1 Architecture of Square Root

The algorithms for the computation of the square root resemble the algorithms for the computation of division. Specifically, there exist many different algorithms for the square root function. The algorithms can be separated into the three categories of digit recurrence, functional iteration and approximation algorithms [6]. These algorithms differ only in the way that the square root of the mantissa is evaluated.

For the square root computation, the floating-point number is firstly normalized. Next, a second normalization might be needed so as to make the exponent even [6]. Afterwards, the following equation is applied.

$$\sqrt{m \times 2^e} = \sqrt{2^c \times m} \times 2^{\frac{e-c}{2}}, \text{ where } \tfrac{e-c}{2} \text{ is an integer due to down-rounding.}$$

In this report, functional iteration will be used. The algorithm for approximating the square root is Newton's iteration. The iterative equation $x_{n+1} = (\frac{1}{2}) \times (x_n + \frac{s}{x_n})$, which corresponds to $x_{n+1} = 0.5 \times x_n \times (3 - s \times x_n^2)$, will be utilized until convergence to approximate $\frac{1}{\sqrt{s}}$. In the equation, the initial value x_0 should be between 0 and $\frac{\sqrt{3}}{\sqrt{s}}$. Next, for finding $\sqrt{s}$, the result from x_{n+1} will be multiplied by s [36].

For the initial approximation of x_0, the same technique as in division will be used. Specifically, the initial low-precision approximation can be found from a look-up table. However, due to the delay in the communication between the FPU and the ROM, this technique will not be used. Instead, linear approximation can be used to obtain x_0.

8.2 Alternative Structures for the Square Root function

The evaluation of $\sqrt{x}$ can also be performed using the CORDIC algorithm. Based on J.Walther ([13], [14], 1971), the generalized CORDIC iteration can be represented with the following three equations.

$$x_{n+1} = x_n - m \times d_n \times y_n \times 2^{-s}$$
$$y_{n+1} = y_n + d_n \times x_n \times 2^{-s}$$
$$z_{n+1} = z_n - w_s$$

Based on J.M.Muller [11], the recursive relation for the square root function is the following.

$$\sqrt{x} = K \times \sqrt{(x + \tfrac{1}{4K^2})^2 - (x - \tfrac{1}{4K^2})^2}$$

The K variable in the above equation is: $K = \prod_{i=0}^{\infty}(\sqrt{1 + 2^{-2i}})$ [11]. The recursive algorithm can only be used with floating point numbers. Based on the book `Computer Arithmetic Algorithms`, written by I.Koren, the multiplicative normalization algorithm (i.e. CORDIC) is utilized for the evaluation of trigonometric functions such as $sin(x)$ [12]. However, the algorithm could also be implemented for the square root function. Due to time constraints and due to the efficiency of Newton's iteration in terms of accuracy and throughput, this algorithm will not be investigated further.

Another implementation of the square root involves the use of Taylor series. The equation with which $\sqrt{x}$ is approximated can be seen below. However, the use of Taylor series leads to a poor approximation due to being a local approximation only. Orthogonal polynomials, such as Lagrange and Chebyshev, are better than the Taylor series but they do not work efficiently when the domain is infinite. They are mainly used for interpolation and not for approximation. In particular, based on J.M.Muller [16], in order to approximate a square root on the interval of [0, 1] with an absolute error of 10^{-7}, a polynomial of degree 54 is needed.

$$\sqrt{x} = 1 - \frac{1}{2} \times y - \frac{1}{2\times4} \times y^2 - \frac{1\times3}{2\times4\times6} \times y^3 - ... - \frac{1\times3\times5\times...\times(2i-3)}{2\times4\times6\times...\times2i} \times y^i, \text{ where } y = 1 - x.$$

8.3 Verification of Square Root

Neither underflow nor overflow can occur when computing the square root [22]. In fact, the square root only overflows when the input is $+\infty$. Table 33 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 29: The figure shows test cases for the square root function

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

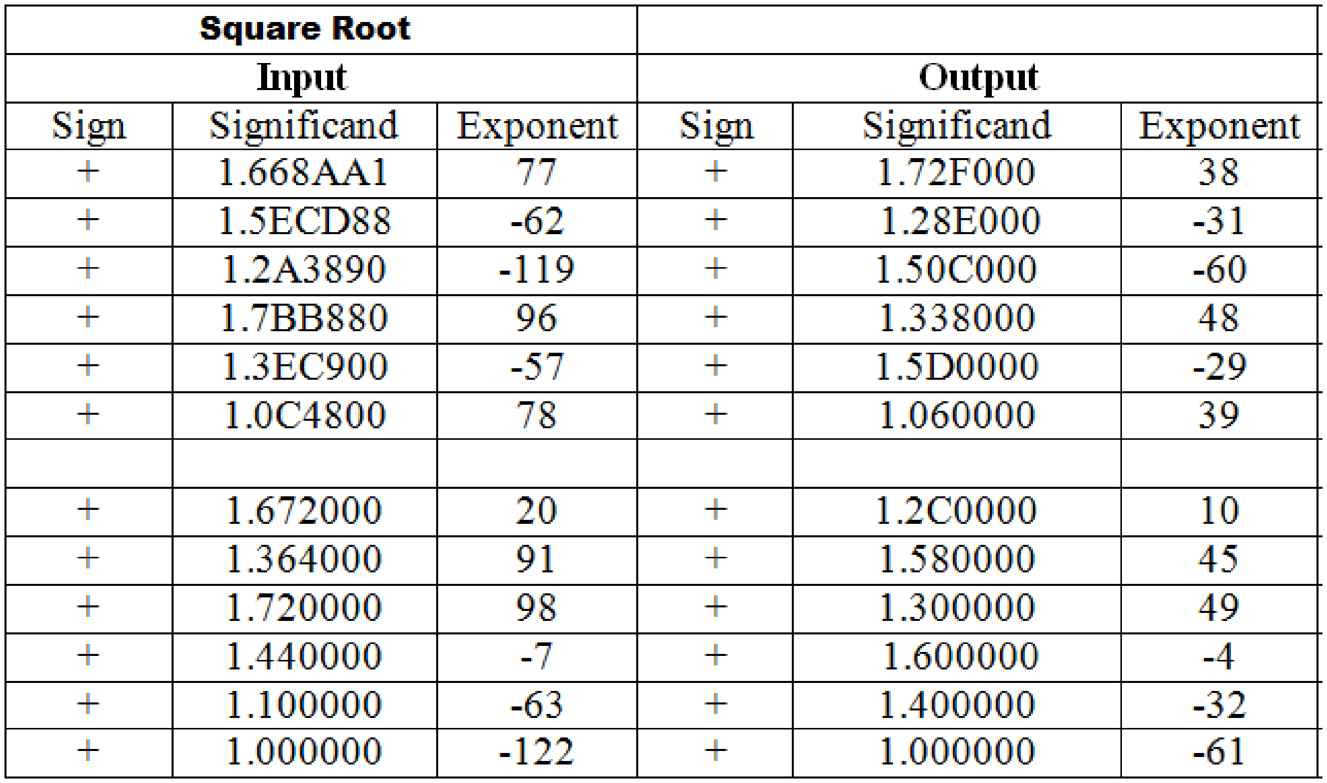
The square root unit should also be checked with denormals. Table 30 presents such test cases.

Table 30: 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

In the square root unit, the sticky bit calculation is important. Table 31 shows test cases for which the sticky bit is checked. In all of the following tests, the sticky bit should be 0.

Table 31: Test cases for ensuring that that the square root function works as expected



Square Root					
Input			**Output**		
Sign	Significand	Exponent	Sign	Significand	Exponent
+	1.668AA1	77	+	1.72F000	38
+	1.5ECD88	-62	+	1.28E000	-31
+	1.2A3890	-119	+	1.50C000	-60
+	1.7BB880	96	+	1.338000	48
+	1.3EC900	-57	+	1.5D0000	-29
+	1.0C4800	78	+	1.060000	39
+	1.672000	20	+	1.2C0000	10
+	1.364000	91	+	1.580000	45
+	1.720000	98	+	1.300000	49
+	1.440000	-7	+	1.600000	-4
+	1.100000	-63	+	1.400000	-32
+	1.000000	-122	+	1.000000	-61

Furthermore, apart from the above test cases, a model to test special patterns in the significands of the input operands will be used. Table 32 presents these test cases.

Table 32: Test cases with special patterns in the significands of the input operands

Square Root					
Input			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent
-	1.7FFFFD	-95	NaN		
+	1.7FB7FF	-126	+	1.34EB7C	-63
+	1.2AAAAA	116	+	1.13CD3A	58
+	1.555555	-94	+	1.253F4EP	-47
+	1.666666	0	+	1.2BBAE2	0
+	1.7FFB1B	-36	+	1.350338	-18
+	1.00045E	-98	+	1.00022F	-49
+	1.7FFFFF	124	+	1.3504F3	62
+	1.000000	-87	+	1.3504F3	-44
+	1.7FB7FF	-126	+	1.34EB7C	-63
+	1.2AAAAA	116	+	1.13CD3A	58
+	1.555555	-94	+	1.253F4E	-47
+	1.666666	0	+	1.2BBAE2	0
+	1.7FFB1B	-36	+	1.350338	-18
+	1.00045E	-98	+	1.00022F	-49

As it was mentioned in section 1.2, a model that will test final results that are close to a specific value is needed. Table 33 indicates cases that test final results that are very close to a specified value, such as zero or one.

Table 33: Test cases for which the final results are very close to 0 or 1

Square Root					
Input			Output		
Sign	Significand	Exponent	Sign	Significand	Exponent
+	1.000005	0	+	1.000002	0
+	1.000401	0	+	1.000200	0
+	1.000800	0	+	1.000400	0
+	1.040801	0	+	1.020000	0
+	1.100000	1	+	1.400000	0

A final verification check will be to test the monotonicity of the square root function. Based on mathematical theory, the square root function does not decrease. If $x > y$, then $\sqrt{x} > \sqrt{y}$. This relationship will be tested for random floating point numbers so as to check that it is not violated.

9. Inverse Square Root

9.1 Architecture of Inverse Square Root

The technique to evaluate the inverse square root function $\frac{1}{\sqrt{x}}$ is identical to the technique used for the square root function. Specifically, the iterative equation $x_{n+1} = (\frac{1}{2}) \times (x_n + \frac{s}{x_n}) = 0.5 \times x_n \times (3 - s \times x_n^2)$ will be used until convergence. Convergence occurs when x_0 is between 0 and $\frac{\sqrt{3}}{\sqrt{s}}$. In contrast to the square root function, the result from x_{n+1} should not be multiplied by s.

9.2 Verification of Inverse Square Root

The following table shows the initial test cases that will be used to verify the inverse square root function.

Table 34: The table depicts test cases for ensuring that the inverse square root function works as expected

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

Apart from these initial tests, the methodology for testing the inverse square root function will be identical to the methodology for testing the square root function.

10. 2D Euclidean Distance

10.1 Architecture of 2D Euclidean Distance

In this section, the following vector a will be used.

$$a = \begin{pmatrix} x \\ y \end{pmatrix}$$

The 2D Euclidean Distance $\sqrt{x^2 + y^2}$ could be implemented with the 2D dot-product function followed by the square root function if it was known that overflow/underflow would not occur. In other words, if it was known that overflow/underflow would not occur, then the equation $|a| = \sqrt{x^2 + y^2} = \sqrt{a^\tau a}$ would be used. However, this is not the case since overflow or underflow might occur in the intermediate calculations.

Scaling can be used so as to avoid overflow and underflow [35]. Hence, for the calculation of the Euclidean Distance, all the elements of the vector should be divided by the largest element of the vector. In the end, the final result for the Euclidean Distance is found by multiplying the scaled vector by the largest element of the initial vector. For instance, when $x = 5$ and $y = 10$, then the following scaled vector arises.

$$b = a_{scaled} = \begin{pmatrix} \frac{5}{10} \\ \frac{10}{10} \end{pmatrix} = \begin{pmatrix} 0.5 \\ 1 \end{pmatrix}$$

Therefore: $|a| = 10 \times |b|$ and $|a| = 10 \times \sqrt{0.25 + 1} = 11.18$, which is the correct result. However, this algorithm will not be used. Based on [35], this algorithm does not produce neither accurate nor fast results. On the one hand, the results of the algorithm are not accurate due to rounding errors. On the other hand, the algorithm is not efficient in terms of latency due to traversing the array more than one time.

Another way to avoid overflow/underflow is to use the following equations dependent on whether $|x|$ is larger or smaller than $|y|$. This algorithm requires the functions of division, addition, square root and multiplication [37]. This algorithm will be implemented and tested.

$$\begin{cases} |x| \times \sqrt{1 + (\frac{y}{x})^2}, & |y| \le |x| \\ |y| \times \sqrt{(\frac{x}{y})^2 + 1}, & |x| \le |y| \end{cases}$$

An algorithm that uses one iteration of the vector a is described in [34]. This algorithm utilizes floating-point division. Due to the fact that we consider a 2D vector, this algorithm will not be implemented and tested.

10.2 Alternative Structures for the 2D Euclidean Distance

Another way to implement the 2D Euclidean Distance $\sqrt{x^2 + y^2}$ is to use the vectoring mode of the CORDIC algorithm. The CORDIC algorithm was also presented as an alternative structure of the square root function. To evaluate $\sqrt{x^2 + y^2}$, the following two iterative equations can be used. The term δ_i is defined as $\delta_i = \pm(\frac{1}{2})^i$ and $i = 0, 1, 2, 3, ...$ [21].

$$x_{i+1} = x_i + y_i \times \delta_i$$
$$y_{i+1} = y_i - x_i \times \delta_i$$

However, the above algorithm fails when a 333-444-555 triangle is used. Specifically, when x equals 333 and when y equals 444, the algorithm produces an incorrect result [21]. Due to this problem and due to the difficulty of implementing the algorithm in hardware, the vectoring mode of the CORDIC will not be examined further.

10.3 Verification of 2D Euclidean Distance

The test cases that will be considered are identical to the test cases for the addition and the multiplication functions. Apart from these test cases, the following tests will also be performed. The equation $(-0)^2 = (+0)$ has been used.

Table 35: Test cases for the 2D Euclidean Distance

2D Euclidean Distance $\sqrt{x^2 + y^2}$		
Input x	**Input y**	**Output**
+0	+0	+0
+0	-0	+0
-0	+∞	+∞
-0	-0	+0
-0	NaN	NaN
+∞	+0	+∞
+∞	-0	+∞
-∞	+∞	+∞
+∞	-∞	+∞
-∞	NaN	NaN

11. 3D Euclidean Distance

11.1 Architecture and Verification of 3D Euclidean Distance

In this section, the following vector a will be used.

$$a = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

The technique to calculate the 3D Euclidean Distance $\sqrt{x^2 + y^2 + z^2}$ is identical to the technique used in the previous section to evaluate the 2D Euclidean Distance. In addition, the verification techniques for the 2D and 3D Euclidean Distances are also identical.

12. Normalised Vector

12.1 Architecture of Normalised Vector

The normalized vector is the final function of the FPU. In n dimensions, let x be a vector in R^n and consider normalization. The function $f(x)$ that is obtained is the following: $f(x) = \frac{x}{\sqrt{\sum_{i=0}^{n} x_i^2}}$.

Approximating $f(x)$ with multi-variate orthogonal polynomials, such as Lagrange and Chebyshev polynomials, does not seem feasible when n is large and x is unbounded. In this report, the case in which $n = 3$ will be investigated. Hence, the following vector x is engendered.

$$x = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

The architecture that will be examined in this report will concern the normalization in one dimension. In other words, the architecture of $\frac{a}{\sqrt{a^2+b^2+c^2}}$ will be investigated. In order for the entire vector to be normalized, the process should be performed three times.

Compound methods will be the first to be used, chaining the Euclidean distance and inverse square root with a multiply to perform a normalisation. However, more efficent algorithms exist and will be investigated in further depth once the other functions are running reasonably efficently.

12.2 Verification of Normalised Vector

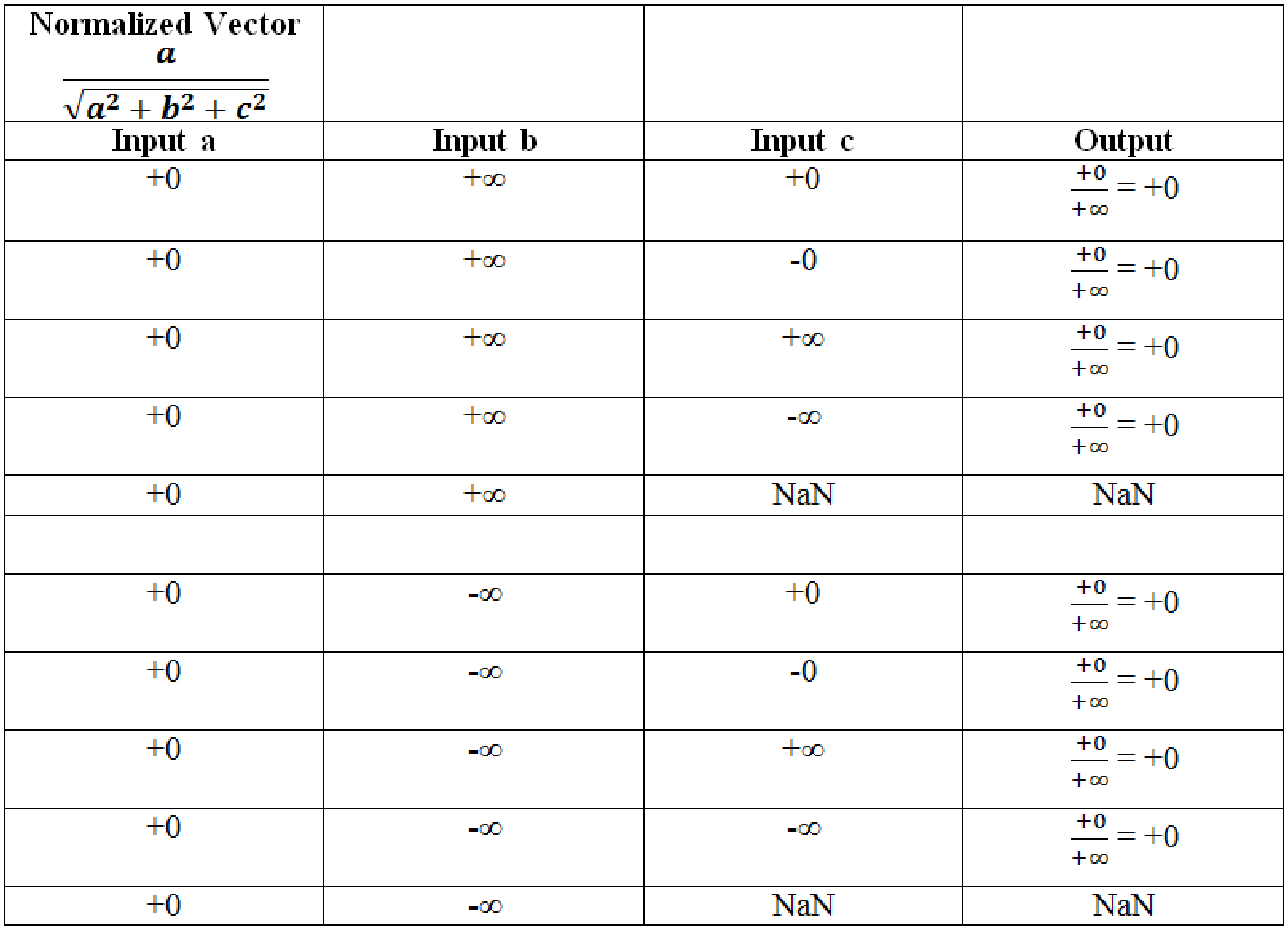
The verification tests from the addition, multiplication, square root and division functions will be used. Apart from these tests, certain results with $+0$ and -0 will be tested. These results can be seen in Table 36.

Table 36: Test cases for the normalized vector function

Normalized Vector $\frac{a}{\sqrt{a^2+b^2+c^2}}$			
Input a	**Input b**	**Input c**	**Output**
+0	+0	+0	$\frac{+0}{+0}$ = NaN
+0	+0	-0	$\frac{+0}{+0}$ = NaN
+0	+0	+∞	$\frac{+0}{+\infty}$ = +0
+0	+0	-∞	$\frac{+0}{+\infty}$ = +0
+0	+0	NaN	NaN
+0	-0	+0	$\frac{+0}{+0}$ = NaN
+0	-0	-0	$\frac{+0}{+0}$ = NaN
+0	-0	+∞	$\frac{+0}{+\infty}$ = +0
+0	-0	-∞	$\frac{+0}{+\infty}$ = +0
+0	-0	NaN	NaN

The next step is to test certain results with $+0$, $+\infty$ and -0, $-\infty$. This can be seen in Table 37.

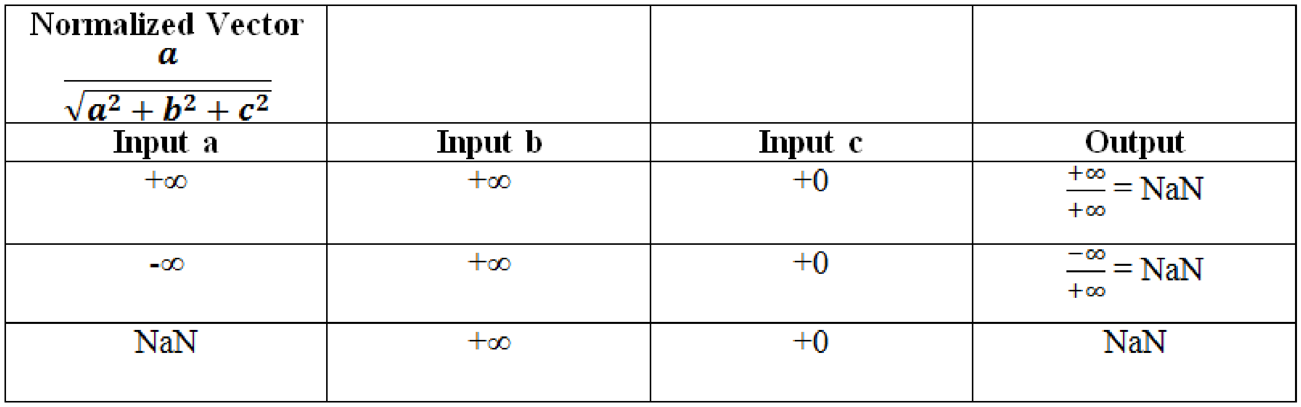
Table 37: Further test cases for the normalized vector function



Normalized Vector $\frac{a}{\sqrt{a^2+b^2+c^2}}$			
Input a	**Input b**	**Input c**	**Output**
+0	+∞	+0	$\frac{+0}{+\infty} = +0$
+0	+∞	-0	$\frac{+0}{+\infty} = +0$
+0	+∞	+∞	$\frac{+0}{+\infty} = +0$
+0	+∞	-∞	$\frac{+0}{+\infty} = +0$
+0	+∞	NaN	NaN
+0	-∞	+0	$\frac{+0}{+\infty} = +0$
+0	-∞	-0	$\frac{+0}{+\infty} = +0$
+0	-∞	+∞	$\frac{+0}{+\infty} = +0$
+0	-∞	-∞	$\frac{+0}{+\infty} = +0$
+0	-∞	NaN	NaN

The final tests for ensuring that the normalized function works correctly involve the use of $+\infty$ and of $-\infty$ in the numerator. Table 38 shows these cases.

Table 38: Final test cases for the normalized vector function



Normalized Vector $\frac{a}{\sqrt{a^2+b^2+c^2}}$			
Input a	**Input b**	**Input c**	**Output**
+∞	+∞	+0	$\frac{+\infty}{+\infty} = \text{NaN}$
-∞	+∞	+0	$\frac{-\infty}{+\infty} = \text{NaN}$
NaN	+∞	+0	NaN

13. Timing and Area Analysis

In this project, the software tool `Synplify Premier` by Synopsys will be used to investigate timing and area analysis. Synplify Premier is chosen because it enables the inspection of these metrics for FPGAs from different vendors.

Timing analysis allows the inspection of the critical path of a design. This critical path can be utilized in order to enhance the design. Initially, timing analysis for each function will be examined. This allows the identification of the relative performance of each block. Hence, optimisation will be performed accordingly. When all functions have been implemented and tested, the throughput and latency for each operation can be verified against the specification in the Requirements Document.

14. Power Analysis

Ultimately, the FPU system will be analysed on an FPGA board. There are several physical constraints to account for and in this section, a detailed analysis of the way the power consumed will be measured is presented. The power analysis architecture will be common to all functions.

To tackle the intricate and challenging question of power consumption, let us first break down the power consumed by an FPGA into its various components:
1)Pre-programmed quiescent power: The amount of power consumed by the FPGA before it has been programmed.
2)Inrush programming power : The power required to program the FPGA.
3)Post-programmed quiescent power : Power consumed by the FPGA at 0MHz frequency.
4)Dynamic power : Power consumed at non-zero frequency components.

The following factors are inherent to the power consumption of a device:
1)Device selection : There are various families of devices from several companies that have different power characteristics. The properties of these devices affecting the power usage range from process technology and supply voltage to electrical design and device architecture.
2)Environmental conditions : The environmental conditions are responsible for the operating temperature of the device and in turn the static or quiescent power consumption. To keep this at a minimum, we must utilise device cooling solutions to keep the device in the thermal range of its ideal operating point.
3)Device resource usage : Device resource usage is the most significant portion of the power consumed. It is dependant on the number, type and loading of pins, hard logic boards and global signals.
4)Signal activities : The behaviours of our signal within the design is responsible for a significant portion of the dynamic power. It is dependant on two important behaviours of the signal: toggle rate and static probability. Toggle rate is the average number of times the signal changes value per unit time . Consequently the static probability of the signal is the fraction of time for which the signal is at logic 1.

For calculating the power, an extremely desirable function is that we are able to ascertain the power consumed prior to the board build. This is to ensure that the board, post-build, does not exceed a fixed power budget or a thermal limit. For this reason, there has been a demand for software-based power calculators that can be used both before and after the PCB assembly to estimate the power.

Now we shall go through the various power calculators for different FPGA's on the market. We have not yet confirmed the make and type of FPGA and in turn the power calculator we will be using since it depends on availability. The following FPGA types will be considered.

1) Altera : Power Play & ArriallGX beta powered spreadsheet
2) Xilinx : Xpower
3) Lattice : Power Calculator

15. References

[1] Nachtigal, Michael, and Himanshu Thapliyal. *Design of a Reversible Floating-Point Adder Architecture. Rep. N.p.: U of South Florida, 2011. 11th IEEE International Conference on Nanotechnology.* Web. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6144358`.

[2] Hennessy, John L., and David A. Patterson. *"3. Arithmetic for Computers." Computer Organization and Design: The Hardware/software Interface. 4rth ed.* San Francisco, CA: Morgan Kaufmann, 1998. 252. Print.

[3] Grover, Naresh, and M.K. Soni. *Design of FPGA Based 32-bit Floating Point Arithmetic Unit and Verification of Its VHDL Code Using MATLAB. Rep. Faridabad, India: Manav Rachna International U, 2014.* Web. `http://www.mecs-press.org/ijieeb/ijieeb-v6-n1/v6n1-1.html`.

[4] Hennessy, John L., and David A. Patterson. *"3. Arithmetic for Computers." Computer Organization and Design: The Hardware/software Interface. 4rth ed.* San Francisco, CA: Morgan Kaufmann, 1998. Print.

[5] Muller, J. M. *"8.6 Floating-Point Division." Handbook of Floating-point Arithmetic.* Boston: Birkhauser, 2010. 262-65. Print.

[6] Muller, J. M. *"8.7.2 Computing the significand square root." Handbook of Floating-point Arithmetic.* Boston: Birkhauser, 2010. 262-265. Print.

[7] W. S. Brown. *A Simple but Realistic Model of Floating-Point Computation.* Rep. N.p.: Bell Laboratories, 1981. Web. `http://dl.acm.org/citation.cfm?id=355975`.

[8] Ibrahim, Eng. Walaa Abd El Aziz. *Binary Floating Point Fused Multiply Add Unit.* Tech. N.p.: Cairo U, 2012. Web. 12 May 2014. `http://eece.cu.edu.eg/~hfahmy/thesis/2012_03_bfpfma.pdf`.

[9] Muller, J. M. *"2. Definitions and Basic Notions." Handbook of Floating-point Arithmetic.* Boston: Birkhauser, 2010. N. pag.52. Print.

[10] Hsiao, Shen-Fu, Po-Han Wu, Chia-Sheng Wen, and Li-Yao Chen. *Design of a Programmable Vertex Processor in OpenGL ES 2.0 Mobile Graphics Processing Units.* Rep. Taiwan: National Sun Yat-Sen U, 2013. Web. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6533843&tag=1`.

[11] Muller, Jean-Michel. *"6. The CORDIC algorithm." Elementary Functions: Algorithms and Implementation.* N.p.: n.p., n.d. 107. Print.

[12] Koren, Israel. *"9.4 Inverse Trigonometric Functions." Computer Arithmetic Algorithms. 2nd ed.* Natick, MA: K Peters, 2002. 235-37. Print.

[13] J. Walther. *A unified algorithm for elementary functions.* In Joint Computer Conference Proceedings, 1971. Reprinted in E. E. Swartzlander, Computer Arithmetic, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[14] Muller, Jean-Michel. *"6. The CORDIC algorithm." Elementary Functions: Algorithms and Implementation.* N.p.: n.p., n.d. 105. Print.

[15] Monniaux, David. *The Pitfalls of Verifying Floating-point Computations.* Rep. N.p.: CNRS / Ecole Normale Superieure, May 22, 2008. Web. `http://hal.archives-ouvertes.fr/docs/00/28/14/29/PDF/floating-point-article.pdf`.

[16] Muller, Jean-Michel. *"3. Polynomial Approximations." Elementary Functions: Algorithms and Implementation.* N.p.: n.p., n.d. 32. Print.

[17] *THE MPFR LIBRARY: ALGORITHMS AND PROOFS.* Rep. N.p.: n.p., n.d. Web. `http://www.cs.berkeley.edu/~fateman/generic/algorithms.pdf`.

[18] Aharoni, Merav, and Sigal Asaf. *FPgen - A Test Generation Framework for Datapath Floating-Point Verification.* Rep. N.p.: IBM Haifa Research Labs, n.d. Web. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1252469&tag=1`.

[19] . 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`.

[20] Ashenden, Peter J. *The Designer's Guide to VHDL. 2nd ed.* N.p.: Systems on Silicon, n.d. 178. Print.

[21] Ronald F. Gleeson. *THE SQUARE ROOT CORDIC.* Rep. N.p.: TRENTON STATE COLLEGE, 26 JULY 1991. Web. `http://www.dtic.mil/dtic/tr/fulltext/u2/a242318.pdf`.

[22] 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. N.p.: U of Antwerp, n.d. Web. `http://dl.acm.org/citation.cfm?id=382404`.

[23] Muller, J. M. *"9.6 Division." Handbook of Floating-point Arithmetic.* Boston: Birkhauser, 2010. 305-313. Print.

[24] Mario P. Vestias, and Horacio C. Neto. *Decimal Division Using the NewtonRaphson Method and Radix-1000 Arithmetic.* Rep. Lisbon, Portugal: Springer Science+Business Media, 2013. Web. `http://www.google.gr/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CCYQFjAA&url=http%3A%2F%2Fwww.springer.com%2Fcda%2Fcontent%2Fdocument%2Fcda_downloaddocument%2F9781461413615-c1.pdf%3FSGWID%3D0-0-45-1365807-p174191065&ei=wf50U4DIA4GN7Qa1g4HABQ&usg=AFQjCNGrUCIl4sPNLJlB5jn2W7e0TMkW-A&bvm=bv.66699033,d.ZWU`.

[25] Stuart F. Oberman, and Michael J. Flynn. *Division Algorithms and Implementations.* Rep. N.p.: IEEE TRANSACTIONS ON COMPUTERS, AUGUST 1997. IEEE. Web. `http://hackipedia.org/Algorithms/Optimization/misc,%20adiv/divalgo_TOC.pdf`.

[26] Muller, J. M. *"8.6.2 Computing the significand quotient." Handbook of Floating-point Arithmetic.* Boston: Birkhauser, 2010. 263. Print.

[27] Oberman, Stuart F. *Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7TM Microprocessor.* Rep. Sunnyvale: California Microprocessor Division, Advanced Micro Devices, n.d. IEEE. Web. `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=762835`.

[28] Parhami, Behrooz. *"16.3 Division by Reciprocation." Computer Arithmetic: Algorithms and Hardware Designs.* New York: Oxford UP, 2000. 265-66. Print.

[29] OGITA, TAKESHI, SIEGFRIED M. RUMP, and SHINICHI OISHI. *ACCURATE SUM AND DOT PRODUCT.* Rep. 2005. Web. `http://www.ti3.tuhh.de/paper/rump/OgRuOi05.pdf`.

[30] Kulisch, Ulrich. *"1.3. High-performance Scalar Product Units (SPU)". Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units.* Wien: Springer, 2002. Print.

[31] Agrawal, Gaurav, and Ankit Khandelwal. *A Newton Raphson Divider Based on Improved Reciprocal Approximation Algorithm.* Rep. 2006. Web. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.134.725&rep=rep1&type=pdf`.

[32] Schwarz, Eric M., and Michael J. Flynn. *Hardware Starting Approximation For The Square Root Operation.* Rep. Web. `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=378103`.

[33] *"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`.

[34] James L Blue. *A Portable Fortran Program To Find the Euclidean Norm of a Vector.* Rep. Bell Laboratories. Web. 1978. `http://degiorgi.math.hr/~singer/aaa_sem/Float_Norm/p15-blue.pdf`.

[35] Pedram, Ardavan, Andreas Gerstlauer, and Robert A. Van De Geijn. *Floating Point Architecture Extensions for Optimized Matrix Factorization.* Rep. U of Texas at Austin. Web. `http://www.cs.utexas.edu/users/flame/pubs/ARITH.pdf`.

[36] Muller, J. M. *"NewtonRaphson-Based Square Root with an FMA." Handbook of Floating-point Arithmetic.* Boston: Birkhauser, 2013. 167. Print.

[37] Brisebarre, Nicolas, and Mioara Joldes. *Augmented Precision Square Roots, 2-D Norms, and Discussion on Correctly Rounding P X2 + Y2.* Rep. IEEE, n.d. Web. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=05992105`.

[38] DETREY, JEREMIE, and FLORENT DE DINECHIN. *A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic. Rep. Ecole Normale Superieure De Lyon, n.d. Web.* `<http://hal.inria.fr/docs/00/54/22/12/PDF/DetreyDinechin.pdf>`.