ABSTRACT OF THESIS

HDL IMPLEMENTATION AND ANALYSIS OF A RESIDUAL REGISTER FOR A FLOATING-POINT ARITHMETIC UNIT

Processors used in lower-end scientific applications like graphic cards and video game consoles have IEEE single precision floating-point hardware [23]. Double precision offers higher precision at higher implementation cost and lower performance. The need for high precision computations in these applications is not enough to justify the use double precision hardware and the extra hardware complexity needed [23]. Native-pair arithmetic offers an interesting and feasible solution to this problem. This technique invented by T. J. Dekker uses single-length floating-point numbers to represent higher precision floating-point numbers [3]. Native-pair arithmetic has been proposed by Dr. William R. Dieter and Dr. Henry G. Dietz to achieve better accuracy using standard IEEE single precision floating point hardware [1]. Native-pair arithmetic results in better accuracy however it decreases the performance by 11x and 17x for addition and multiplication respectively [2]. The proposed implementation uses a *residual register to store the error residual term* [2]. This addition is not only cost efficient but also results in acceptable accuracy with 10 times the performance of 64-bit hardware. This thesis demonstrates the implementation of a 32-bit floating-point unit with residual register and estimates the hardware cost and performance.

Keywords: Native-pair floating-point unit residual VHDL

HDL IMPLEMENTATION AND ANALYSIS OF A RESIDUAL REGISTER FOR A FLOATING-POINT ARITHMETIC UNIT

BY

Akil Kaveti

Dr. William R. Dieter

Director of Thesis

Dr. YuMing Zhang

Director of Graduate Studies

March 25, 2008

Rules for the use of theses

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A Library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

_____

_____

_____

_____

_____

_____

_____

THESIS

Akil Kaveti

The Graduate School

University Of Kentucky

2008

HDL IMPLEMENTATION AND ANALYSIS OF A RESIDUAL REGISTER FOR A FLOATING-POINT ARITHMETIC UNIT

MASTERS THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Electrical Engineering at the University of Kentucky

By
Akil Kaveti
Lexington, Kentucky
Director of Thesis: Dr. William Dieter
Electrical Engineering, University of Kentucky,
Lexington, Kentucky

2008

ACKNOWLEDGEMENTS

Foremost I would like to thank my advisor, Dr. William R. Dieter for providing me the opportunity to do this thesis. I am grateful to him for his constant guidance and support. I would also like to thank Dr. Hank Dietz for his suggestions which have helped in improving this thesis.

I would like to thank my Candy for being such a wonderful person and for motivating me time and again.

Most importantly I would like to thank my parents for their love and support I dedicate my thesis to them.

Table of Contents

List of Figures

List of Tables

List of Files

Chapter 1. Introduction

This chapter briefly introduces all the topics that will be encountered and described in the later parts of the thesis. It starts by giving the reason for using floating-point numbers in computation. It discusses the floating-point arithmetic, cost involved in implementing higher precision than the existing floating-point hardware, native-pair arithmetic and its usage for better precision and accuracy, performance-cost factor in native-pair arithmetic and extra hardware support to improve the performance-cost factor. This chapter ends with the author's motivation to work on native-pair Floating-point Arithmetic unit and the organization of the thesis.

1.1. Computer Representation of Real Numbers

Real numbers may be described as numbers that can represent a number with infinite precision and are used to measure continuous quantities. Almost all computations in Physics, Chemistry, Mathematics or scientific computations, all involve operations using real numbers. Computers can only approximate real numbers, most commonly represented as fixed-point and floating-point numbers**.** In a Fixed-point representation, a real number is represented by a fixed number of digits before and after the radix point. Since the radix point is fixed, the range of fixed-point also is limited. Due to this fixed window of representation, it can represent very small numbers or very large numbers accurately within the available range. A better way of representing real numbers is floating-point representation. Floating-point numbers represent real numbers in scientific notation. They employ a sort of a sliding window of precision or number of digits suitable to the scale of a particular number and hence can represent of a much wider range of values accurately. Floating-point representation has a complex encoding scheme with three basic components: mantissa, exponent and sign. Usage of binary numeration and powers of 2 resulted in floating point numbers being represented as single precision (32-bit) and double precision (64-bit) floating point numbers. Both single and double precision numbers are defined by the IEEE 754 standard. According to the standard, a

single precision number has one sign bit, 8 exponent bits and 23 mantissa bits where as a double precision number comprises of one sign bit, 11 exponent bits and 52 mantissa bits.

Most processors designed for consumer applications, such as Graphical Processing Units (GPUs) and CELL processors promise and deliver outstanding floating point performance for scientific applications while using the single precision floating point arithmetic hardware [23][6]. Video games rarely require higher accuracy in floating-point operations, the high cost of extra hardware needed in their implementation is not justified. The hardware cost of a higher precision arithmetic is lot greater than single-precision arithmetic. For example, one double precision or 64-bit floating point pipeline has approximately same cost as two to four 32-bit floating-point pipelines [1]. Most applications use 64-bit floating point to avoid losing precision in a long sequence of operations used in the computation, even though the final result may not be accurate to more than 32-bit precision. The extra precision is used so the application developer does not have to worry about having enough precision. Native-pair arithmetic presents an opportunity to increase the accuracy of a single-precision or 32-bit floating-point arithmetic without incurring the high expense of a double-precision or 64-bit floating-point arithmetic implementation. Native-pair arithmetic uses two native floating-point numbers to represent the base result and the resulting error residual term that would have been discarded in a native floating point unit [23]. One native floating-point number is represented using two native floating-point numbers. This approach has been adapted from an earlier technique known as double-double arithmetic. Double-double arithmetic is the special case of native-pair arithmetic using two 64-bit double precision floating point numbers to represent one variable; the first floating-point number representing the leading digits and the second the trailing digits [17]. Similarly in Native-pair arithmetic, two 32-bit floating-point numbers are used to represent high and low terms where low component encodes the residual error from high component representation. Though this implementation results in higher accuracy without external hardware, it also degrades in performance [2].

1.2. Hardware Assistance of Native-Pair

In order to obtain acceptable accuracy with less performance loss, addition of simple micro-architectural hardware is needed. Dieter and Dietz proposed a residual register to hold discarded information after each floating-point computation [2]. This feature not only reduces the performance cost of native-pair arithmetic but also provides lower latency and better instruction-level parallelism. A residual register has one sign bit, 8 exponent bits and 25 mantissa bits [23]. The usage of the residual register depends on what operation is being performed and at what stage or stages are the bits being discarded.

The most widely used floating-point standard is the IEEE 754 standard. The IEEE 754 standard prescribes a particular format for representing floating-point numbers in binary system, special floating-point numbers, rounding modes, exceptions and how to handle them. Floating-point operations such as addition, multiplication, division and square root have three stages viz., prenormalization, arithmetic unit and postnormalization. In the case of addition and subtraction, prenormalization increases or decreases the exponent part to align the mantissa parts, calculates the sign bit of the final result. The arithmetic unit does the basic arithmetic involving the mantissa bits. The result may not be in the appropriate format, so it is sent into the postnormalization unit. It is in the postnormalization that the result from previous stage is aligned to the IEEE 754 format, rounded depending on the rounding mode and the number with its sign, exponent and mantissa bits is given as the final result.

This thesis aims to prove that residual register hardware with minimal increase in hardware cost results in accuracy close to double-precision and hence is the more economically feasible solution for higher precision arithmetic than the double-precision hardware**.** Native-pair arithmetic presents an opportunity for more accurate and precise floating-point processing, but it also results in a decrease in performance and increase in implementation cost when compared with the single precision or 32-bit floating-point hardware [23]. The usage of residual register as the extra hardware for storing the error

residual term in native-pair arithmetic gives an implementation which has a slight increase in hardware cost coupled with performance close to that of single precision hardware [23]. Floating point arithmetic unit with residual register is implemented and its hardware utilization, maximum operable frequency is compared with the 32-bit and 64-bit floating-point arithmetic unit implementations. The main idea is to find the extra hardware cost and the performance drop resulting due to the residual register usage, moving the discarded bits into it, updating the residual register if bits are discarded more than once and also setting the sign and exponent of the residual register. The implemented floating-point unit uses the residual register with addition, subtraction and multiplication. The extra hardware needed accounted to an increase of 18% in adder and 12% in multiplier. A minimum period increase of 19% for adder and 12% for multiplier also resulted due to addition of extra hardware in the critical path. The divide and the square root portions of the floating-point unit are left unchanged.

A floating-point unit coded in VHDL was adopted for the purpose of developing a Native-pair Floating point unit from it [19]. The initial part of this thesis was to debug the code and make it fully pipelined to generate outputs on continuous clock cycles. Signals were added to carry the input operands and the intermediate outputs through the pipeline to wherever needed. Those signals which were present in the final stages and required input operands to be set have been moved to starting stage in order to eliminate the need to carry input operands. The Native-pair floating point unit is implemented by adding the residual register hardware to the debugged native floating point unit. The debugged code is a single precision or 32-bit floating point unit and was scaled to serve as a 64-bit floating point unit. The synthesis reports for the three implementations viz., 32-bit version, native-pair version or 32-bit with residual register and 64-bit version were obtained using Xilinx 9.1 ISE tool and a comparison of their resource utilizations and minimum periods is obtained.

1.3. Thesis Organization

In Chapter 2 forms the background of this thesis. It discusses in detail the IEEE 754 floating-point arithmetic standard, IEEE 754 floating-point addition/subtraction, multiplication, Native-pair arithmetic, Native-pair arithmetic algorithms. Chapter 3

describes the working of 32-bit floating-point unit and native-pair floating-point unit with residual register. The different components of a floating-point unit are discussed in this chapter. Also covered in this chapter is where the residual register is added, how it is set or updated, when it is complemented and how its sign is set, usage of the MOVRR instruction. Chapter 4 describes how the Native-pair floating-point unit is tested. This chapter covers the test- benches used to test the implementation. Chapter 5 consists of the post map and route simulation reports, synthesis reports of native-pair floating point unit, 32-bit floating point unit and 64-bit floating point unit.

Chapter 6 compares the synthesis reports, provides a more detailed analysis of the implementation. Chapter 7 concludes the thesis and discusses the avenue for future research.

Chapter 2. Background

Hardware supporting different floating-point precisions and various formats have been adopted over the years. Amongst the earliest programmable and fully automatic computing machines, the Z3 built of relays and performed calculations using 22-bit word lengths in binary floating-point arithmetic [21]. The first commercial computer supporting floating-point, the Z4, had floating point hardware that supported 32-bit word length comprising of 7 bit exponent, 1 sign bit and 24 mantissa bits [22]. The second one was the IBM 704 in 1954 whose floating point hardware supported a format consisting of 1 sign bit, 8-bit exponent and 29-bit magnitude. IBM considered the 704 format as single precision and later in the IBM 7094 double precision was introduced which had a sign bit, 17-bit exponent and 54-bit magnitude [20]. The DEC – Digital Equipment Corporation's PDP 11/45 had an optional floating point processor. This processor is considered a predecessor to the IEEE 754 standard as it had a similar single precision format. The NorthStar FPB-A was a S100 bus floating point microprogram controlled processor, *built* on medium and small scale TTL parts and PROM memories to perform high speed decimal floating point arithmetic operations. It supported 2, 4, 6, 8, 10, 12, 14 digit precision and 7-bit base-10 exponent [25] [23]. The MASPAR MP1 supercomputer performed floating point operations using 4-bit slice operations on the mantissa with special normalization hardware and supported 32-bit and 64-bit IEEE 754 formats.

The CELL processor, most DSPs and GPUs support the IEEE 32-bit format. The Intel X87 floating point mechanism allows 32-bit, 64-bit and 80-bit operands but processes these operands using an 80-bit pipeline [23] [6] [7]. The standardization of IEEE 754 floating point standard in 1985 has greatly improved the portability of floating-point programs. This standard has been widely accepted and is used by most processors built since 1985.

2.1. IEEE 754 Floating-point standard

The IEEE 754 floating-point standard is the most widely used standard for floating-point computations and is followed in most of the CPU and FPU (Floating point unit) implementations. The standard defines a format for floating-point numbers, special

numbers such as the infinite's and NAN's, a set of floating-point operations, the rounding modes and five exceptions. IEEE 754 specifies four formats of representation: single-precision (32-bit), double-precision (64-bit), single extended (≥ 43 bits) and double extended precisions (≥ 79 bits).

Under this standard, the floating point numbers have three components: a sign, an exponent and a mantissa. The mantissa has an implicit hidden leading hidden bit and the rest are fraction bits. The most used formats described by this standard are the single-precision and the double-precision floating-point number formats which are shown in Table 1. In each cell the first number indicates the number of bits used to represent each component, and the numbers in square brackets specify bit positions reserved for each component in the single-precision and double–precision numbers.

Table 1. Layouts for single and double precision numbers in IEEE 754 format.

Format	Sign	Exponent	Fraction / Mantissa	Bias
Single-precision	1 [31]	8 [30 – 23]	23 [22 – 0]	127
Double-precision	1[63]	11 [62 - 52]	52 [51 - 0]	1023

The Sign bit: A sign bit value of 0 is used to represent positive numbers and 1 is used to represent negative numbers

The Exponent: The exponent field has 8 bits in single-precision and 11 bits in double–precision. The value is stored in unsigned format and a *bias* is added to the actual exponent to get the stored exponent. For single-precision, the bias value is 127 and for double-precision it is 1023. Actual exponent = stored exponent – 127 for single-precision and it is equal to stored exponent – 1023 for double-precision. Denormalized numbers and zero have all zeroes in the exponent field. The infinite and Not a number values have all one's in the exponent field. The range of the exponent for single precision is from -126 to +127 and for double-precision it is -1022 to +1023.

The Mantissa: Apart from the sign and the exponent a floating-point number also has a magnitude part which is represented by the mantissa field. For single-precision the

number of mantissa bits is 23 and for double-precision it is 52. Each mantissa has a hidden bit which is not shown when the floating-point is represented in the IEEE format. This is because all the floating-point numbers are adjusted to have this hidden bit equal to 1 and so the fact that hidden bit is 1 is understood and so is not specified explicitly. Denormalized numbers have the hidden bit set to zero.

In general, floating-point numbers are stored in normalized form. This puts the radix point after the first non-zero digit. In normalized form, six is represented as $+ 6.0 \times 10^0$. In binary floating-point number representation, the radix point is placed after a leading 1. In this form six is represented as $+ 1.10 \times 2^2$. In general, a normalized floating-point number is represented as $\pm 1.f \times 2^e$. There is an implicit leading hidden 1 before the radix point and 23 visible bits after the radix point. The value of the IEEE 754 32-bit floating point number can be computed from the sign bit (s), 8-bit biased exponent field (e), and 23-bit fraction field (f) and arranging them as follows: Value = $(-1)^s 2^{e-127} \times 1.f$

When a nonzero number is being normalized, the mantissa is shifted left or right. Each time a left shift is performed, the exponent is decremented. In case the minimum exponent is reached but further reduction is still required, then the exponent value is taken 0 after biasing, such a number is a denormalized number. Hence a number having zeroes in its exponent field and at least a single 1 in its mantissa part is said to be a denormalized number. The IEEE 754 standard represents the denormalized number as follows:Value = $(-1)^s 2^{-126} \times 0.f$

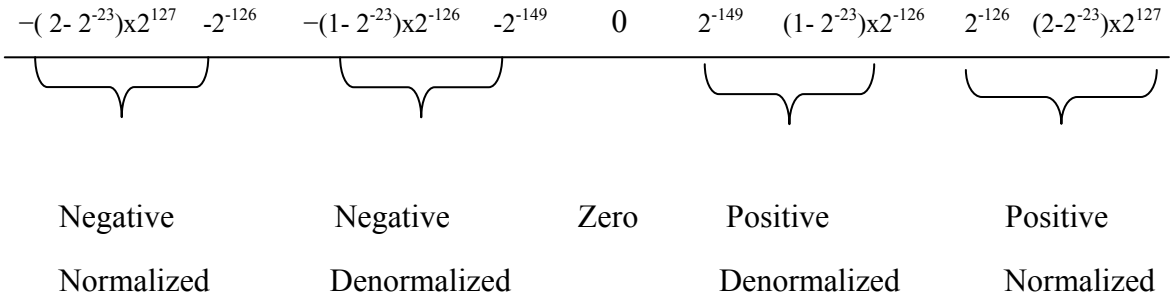


Figure 1. Number line showing the ranges of single-precision denormalized and normalized floating-point numbers in binary system.

Table 2. Representation of single-precision binary floating-point numbers

Sign	Exponent	Mantissa	Value
0	00000000	00000000000000000000000	+ 0
1	00000000	00000000000000000000000	-0
0	11111111	00000000000000000000000	+∞
1	11111111	00000000000000000000000	−∞
0	00000000	00000000000000000000001 to 11111111111111111111111	Positive Denormalized floating-point numbers
1	00000000	00000000000000000000001 to 11111111111111111111111	Negative Denormalized floating-point numbers
0	00000001 to 11111110	xxxxxxxxxxxxxxxxxxxxxxx	Positive Normalized floating-point numbers
1	00000001 to 11111110	xxxxxxxxxxxxxxxxxxxxxxx	Negative Normalized floating-point numbers
0/1	11111111	10000000000000000000000 to 11111111111111111111111	QNaN - Quiet Not a Number
0/1	11111111	00000000000000000000001 To 01111111111111111111111	SNaN – Signaling Not a Number

Exceptions

IEEE 754 floating-point standard defines five exceptions that are generally signaled using a separate flag. They are as follows:

1. Invalid Operation: Some operations like divide by zero, square root of a negative number or addition and subtraction from infinite values are invalid. The result of such invalid operation is NaN – Not a Number. NaNs are of two types: QNaNs, or Quiet NaNs, and SNaNs or signaling NaNs. Their formats are shown in table 2.

   The result of an invalid operation will result be a QNaN with a QNaN or SNaN exception. The SNaN can never be the result of any operation, only its exception can be signaled and this happens whenever one of the operands to a floating-point operation is SNaN. The SNaN exception can be used to signal operations with uninitialized operands, if we set the uninitialized operands to SNaN. The usage of SNaN is not subject to the IEEE 754 standard.

2. Inexact: This exception is signaled when the result of an arithmetic operation cannot be represented due to restricted exponent range or mantissa precision
3. Underflow: Two events cause that underflow exception to be signaled are tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between $\pm 2^{-126}$. Loss of accuracy is detected when the result is simply inexact or only when a denormalization loss occurs.
4. Overflow: The overflow exception is signaled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signaled when one of the operands is infinity, because infinity arithmetic is always exact.

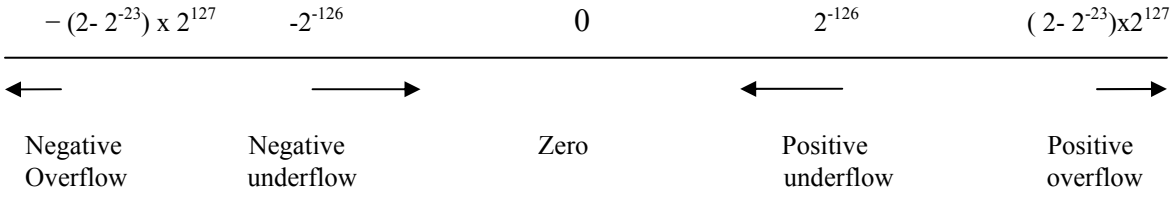


Figure 2. Ranges of overflow and underflow for single-precision floating-point numbers

Rounding modes

Precision is not infinite and sometimes rounding a result is necessary. To increase the precision of the result and to enable round-to-nearest-even rounding mode, three bits are added internally and temporally to the actual fraction: *guard*, *round*, and *sticky* bit. While guard and round bits are normal storage holders, the sticky bit is turned '1' whenever a '1' is shifted out of range.

As an example we take a 5-bit binary number: 1.1001. If we left-shift the number four positions, the number will be 0.0001, no rounding is possible and the result will not be accurate. Now, let's say we add the three extra bits. After left-shifting the number four positions, the number will be 0.0001 101 (remember, the last bit is '1' because a '1' was shifted out). If we round it back to 5-bits it will yield: 0.0010, giving a more accurate result.

The four specified rounding modes are:

1. Round to nearest even: This is the default rounding mode. The value is rounded to the nearest representable number. If the value is exactly halfway between two infinitely precise results or between two representable numbers, then it is rounded to the nearest infinitely precise even number. For example, in one digit base-10 floating-point arithmetic, 3.4 will be rounded to 3, 5.6 to 6, 3.5 to 4 and 2.5 to 2.
2. Round to zero: In this mode, the excess bits will simply get truncated. For example, in two digit base-10 floating-point arithmetic, 3.47 will be truncated to 3.4, and -3.47 will be rounded to -3.4.
3. Round up: In round up mode, a number will be rounded towards $+\infty$. For example, 3.2 will be rounded to 4, while -3.2 to -3.
4. Round down: The opposite of round-up, a number will be rounded towards $-\infty$. For example, 3.2 will be rounded to 3 while -3.2 to -4.

2.2. IEEE 754 Floating-point Arithmetic

The IEEE 754 standard apart from specifying the representation format, the rounding modes and the exceptions also defines the basic operations that can be performed on floating-point numbers.

The Floating-point addition requires the following steps:

1. Aligning the mantissa's to make the exponent of the two operands equal and calculating the sign based on two operands. This exponent becomes the output exponent unless it is changed in the Step 3.

2. The mantissa bits are added or subtracted depending on the signs of the operands.

3. The result from the addition has to be rounded and normalized in order to represent it correctly within the IEEE 754 floating-point format. These three steps are implemented in the floating-point unit three pipeline stages labeled prenormalization, addition unit and postnormalization. The three stages are explained in detail in Chapter 3. Subtraction is the same as addition except that the sign of the subtrahend is inverted before adding the two operands.

Floating-point multiplication also involves three steps:

1. Prenormalization: Multiplication does not require alignment of mantissa in order to make the exponents of the operands equal. In multiplication, the exponents are added and the mantissa bits are transferred to the multiplication stage. The sign of the product is also calculated in this stage.

2. Multiplication: In this stage, the mantissa bits are multiplied using a multiplication algorithm. The product has twice as many mantissa bits as the multiplicands.

3. Postnormalization: The result from the multiplication is rounded and normalized to represent in the given precision format while updating the output exponent when required.

Figure 3 shows the basic algorithm for addition or subtraction of two floating-point numbers.

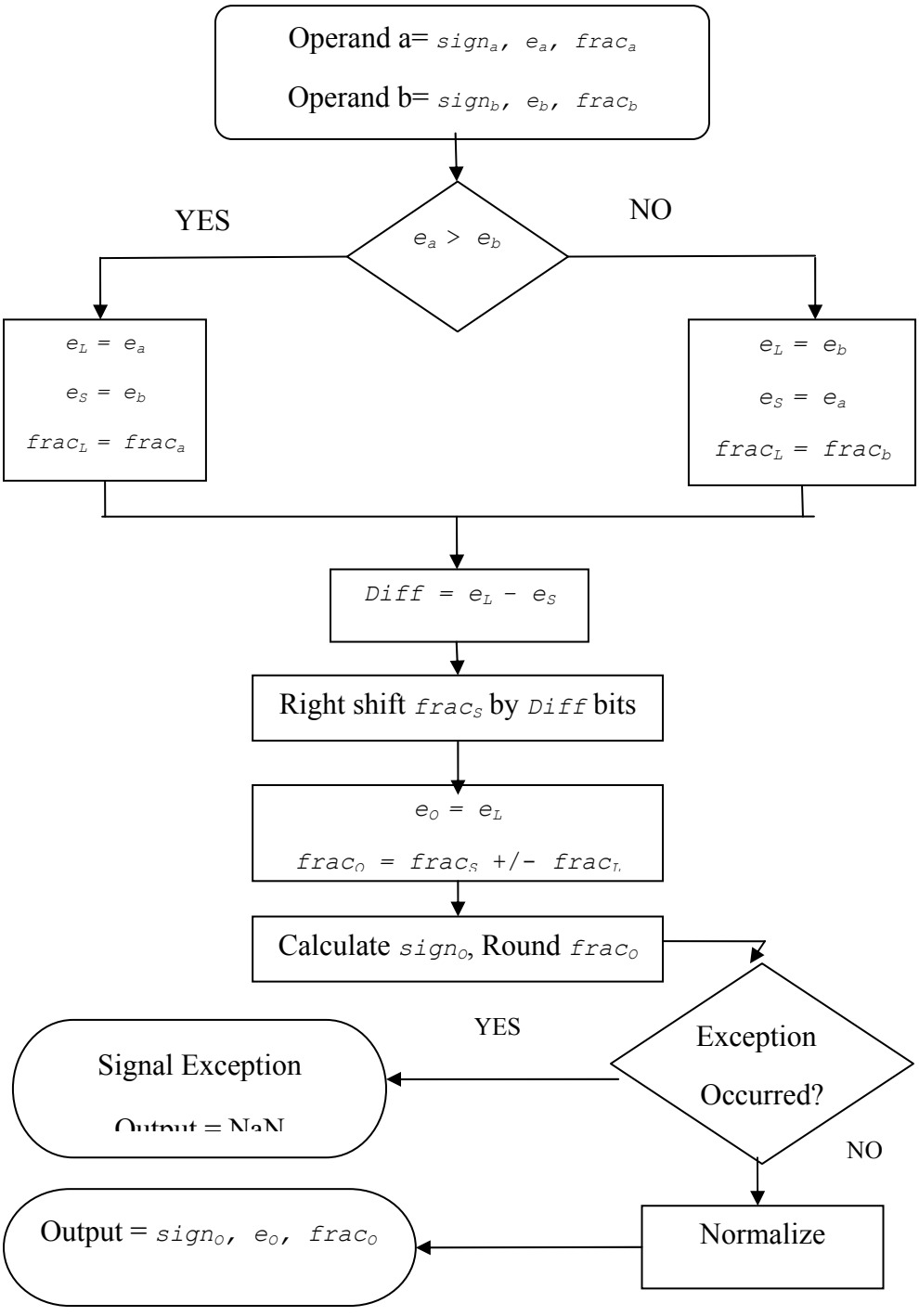


Figure 3. Basic floating-point addition algorithm

Consider a simple example for addition of two floating-point numbers:

Let's say we want to add two binary FP numbers with 5-bit mantissas:

A = 0|00000100|1001

$sign_a$= 0; e_a= 00000100; $frac_a$= 1001

B = 0|00000010|0010

$sign_b$= 0; e_b= 00000010; $frac_b$= 0100

1. Get the number with the larger exponent and subtract it from the smaller exponent.

   e_L = 4, e_S = 2 , so $diff$ = 4 -2 = 2.

2. Shift the fraction with the smaller exponent $diff$ positions to the right. We can now leave out the exponent since they are both equal.

   This gives us the following: 1.1001 000 + 0.0100 100

3. Add both fractions

   1.1001 000
   + 0.0100 100

   1.1101 100
   ----

4. Round to nearest even gives us 1.1110.

5. Result = 0|00000100|1110.

The basic algorithm for floating-point multiplication is shown in Figure 4.

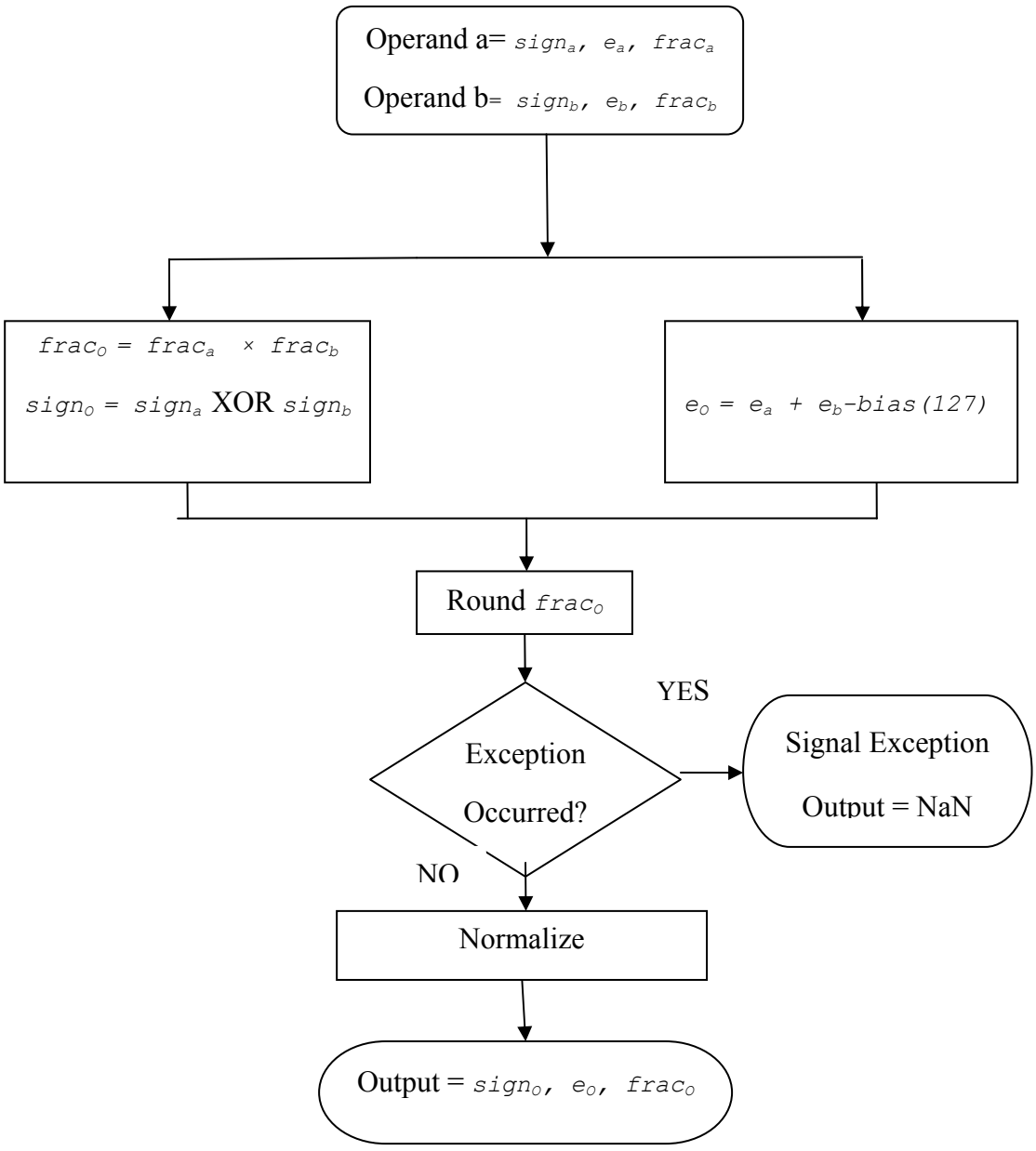


Figure 4. Basic Floating-point multiplication algorithm

Multiplication Example:

1. A = 001001001

   $sign_a$= 0; e_a= 01100100; $frac_a$= 1001

   B = 000100010

   $sign_b$= 0; e_b= 01101110; $frac_b$= 0100

2. 100 and 110 are the stored exponents; logical exponents are obtained by subtracting the bias of 127 from them.

   That is, the logical exponents in this case are 100-127and 110-127.

3. Multiply the fractions and calculate the

   1.1001
   × 1.0010

   1.11000010

   So $frac_O$= 1.11000010 and

   Output exponent: stored exponent = 100+110 and logical exponent = 100+110-127= 83

   e_O = 83

4. Round the fraction to nearest-even: $frac_O$= 1.1100

5. Result: 0|11010010|1100

2.3. History of Double-Double Arithmetic

Using single-length floating point arithmetic to describe or represent multi-length floating point arithmetic has been discussed and algorithms based on this approach were

described by T.J.Dekker in his research report [3]. The report represents a double length floating point number as sum of two single length floating point numbers, one of them being negligible in single length precision. It also discusses the algorithms for basic operations like addition, subtraction and multiplication in the ALGOL 60 language. The Fortran-90 double-double precision system developed by D.H.Bailey uses two 64-bit IEEE arithmetic values to represent quad-precision values in the Fortran 90 programming language [4]. "Implementation of float-float operators on graphics hardware" discusses the methods for improving of precision in floating-point arithmetic on GPUs. The paper discusses different algorithms by Dekker, Knuth and Sterbenz, and the results, performance, and accuracy of these methods [7]. It describes the framework for software emulation of float-float operators with 44 bits of accuracy and proves that these high-precision operators are fast enough to be used in real-time multi pass algorithms [7]**.** The residual register algorithms discussed by Dieter and Dietz [23] and this thesis can be used with these or other precision extending algorithms.

Native-pair arithmetic is a more general term for double-double encompassing precisions other than double. As with double-double, it uses an extra floating-point number to represent error residual term resulting from a floating-point operation. A native-pair value does not have exactly double the precision of the single native value due the occurrence of zeroes in between the two mantissas. These zeroes make the precision equal to the number of bits in the two mantissas plus the number of zeroes between the mantissas [23]. In this approach, a higher-accuracy value is spread across the mantissas of two native floating-point numbers and the exponent of the lower component is used to align the mantissas [23]. The high component, called `hi`, takes the top most significant bits and those that are left, also referred to as residual are represented using the low component, called `lo`. The exponent of `lo` will be less than that of exponent of `hi` by a minimum of N_m, where N_m is the number of mantissa bits in the native floating-point number. This means that if a higher precision value is spread over multiple native floating-point values, the exponents of consecutive `lo` components keep decreasing by N_m [1].

When considering a pair of native floating-point numbers and a 32-bit native mantissa being spread across them, the pair will have twice the precision of the mantissa being

spread only if the exponent of the `hi` is at least N_m greater than that of the native bottom of the exponent range [1]. That is the dynamic range of the exponent is reduced by N_m steps or 10 percent. In a single-precision or 32-bit floating point system, the precision is limited by the exponent range to less than 11 float values [1]. Also, when there are zeros in the top of the lower-half of the higher precision mantissa, then the exponent of `lo` part is further reduced by the number of zeros and the zeros are absorbed [2]. And if there are *K* zeros at the top of the lower-half then, the exponent of `lo` part is reduced by *K*. This has certain implications which are as follows:

- Some values requiring up to *K* bits more precision than twice the native mantissa can be represented precisely, as the *K* zeros that come between the top half and the lower-half are absorbed [1].

- If the adopted native floating-point does not represent denormalized numbers, the Low component may fall out of range sometimes. For example, if the High exponent was 24 above the minimum value and number of zeros $K = 1$, then the result has 25 bits only and not 48 bits as the stored exponent of Low would have to be -1, which is not representable in IEEE format [1].

2.4. The Residual Register

Native-pair arithmetic involves computing the error residual term from the floating point operation and using it to perform further computations. This error residual computation is the major overhead in the native-pair arithmetic. Dieter and Dietz proposed adding a residual register to save this left over information [23]. The residual register is only used to store the mantissa bits, exponent bits, the sign bit, and a complement flag. The value stored in the register need not be normalized immediately and has $N_m + 2$ mantissa bits with an implicit leading 1 bit. The same normalization hardware used for floating-point operations normalizes the residual value only when it is being moved into an architectural register. The complement flag indicates whether the residual value must be complemented before moving into the architectural register. Normalizing the residual register is done by giving a “MOVRR” instruction that copies the residual register value

into an architectural register after normalizing it into IEEE 754 format. Also each operation results in updating the residual register with a new error residual value.

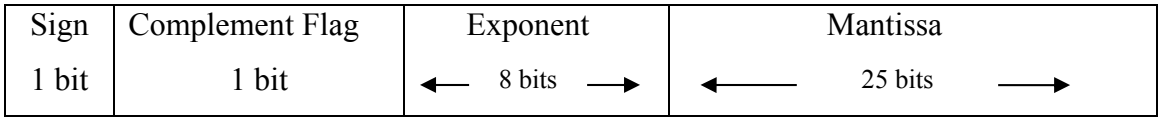


Figure 5. Residual register

Consider two floating-point numbers x, y and o be an operation such as +, -, or ×. Let $sign(x)$, $exp(x)$ and $mant(x)$, respectively denote the sign, exponent and mantissa of x. $Fl(x\ o\ y)$ denotes the primary result of a floating-point operation and $Res(x\ o\ y)$ be the residual of the floating-point operation. For operations discussed here namely addition, subtraction and multiplication the primary result and the residual are related as $x\ o\ y = Fl(x\ o\ y) + Res(x\ o\ y)$. This property holds true only for the round to nearest mode when IEEE 754 format is used. Depending on which rounding mode is used, the sign of the residual register value is set accordingly [23]. The residual logic only needs the information if the primary result is rounded up or down. Depending on this information the sign and the complement flag of the residual register is set as follows:

- When $Fl(x\ o\ y) = x\ o\ y$, the primary result is correct and the residual value is zero.
- When $Fl(x\ o\ y) < x\ o\ y$, the primary result p has been rounded down to the floating-point value with next lower magnitude. The residual r then takes the same sign as p to make $x\ o\ y = Fl(x\ o\ y) + Res(x\ o\ y)$.
- When $Fl(x\ o\ y) > x\ o\ y$, the primary result $Fl(x\ o\ y)$ is rounded up to the next larger magnitude value. The residual r then takes the opposite sign as $Fl(x\ o\ y)$ to make $x\ o\ y = Fl(x\ o\ y) - Res(x\ o\ y)$.

2.5. Native-pair Addition and Subtraction

Addition or subtraction of two floating-point numbers *'a'* and *'b'* with *'b'* being the smaller of the two, involves the shifting of the smaller number to align its radix point with that of the larger number. When the signs of the two numbers are the same, the numbers are added whereas in the case of opposite signs, the numbers are subtracted. The mantissa bits in the smaller number with significance less than $2^{exp(a)-(N_m+1)}$ are stored in the residual register with least significant bit in the rightmost position, and the exponent is set to $exp(b)$ when $exp(a)-exp(b) \geq N_m+1$ and the complement flag is not set. When $exp(a)-exp(b) < N_m+1$ or the complement flag is set, the residual register gets the bits in *b* with significance ranging from $exp(a)-N_m+1$ down to $exp(a)-2(N_m+1)$. That is, the residual register value is just below the primary output value. In this case, the exponent is set to $exp(a)-2(N_m+1)$ with the radix point assumed to be to the right of the least significant residual register bit. The sign and complement flag are set depending on the signs of *'a'* and *'b'*, and whether result *p* is rounded up or down. Four cases that arise depend on the signs of *'a', 'b'* and whether the primary result is rounded up or down, are shown in Table 3 below:

Table 3. Different cases for sign and complement flag of residual register

Case	**Sign of *a***	**Sign of *b***	**Rounded up / down**	**Complement flag**	**Sign of Residual register: *Sign(rr)***
Case 1	*Sign(a)*	*Sign(a)*	*Down*	Cleared	*Sign(a)*
Case 2	*Sign(a)*	*Sign(a)*	*Up*	Set	*Opposite of sign(a)*
Case 3	*Sign(a)*	*-Sign(a)*	*Down*	Set	*Sign(a)*
Case 4	*Sign(a)*	*-Sign(a)*	*Up*	Cleared	*Opposite of Sign(a)*

Native-pair Arithmetic Addition Algorithms

The algorithms that are discussed here are native-pair arithmetic algorithms for normalizing and adding two native-pair numbers. Each algorithm can be implemented with and without using the residual register.

Algorithm 1 shows the `nativepair_normalize` function adds two native floating-point numbers to produce a native-pair result. Given an unnormalized high and low pair of native numbers, the normalized native-pair is computed using this function. In general, the normalized native-pair is created without using the residual register.

Algorithm 1. Native-pair normalization algorithm without using the residual register:

```
   nativepair nativepair_normalize(native hi, native lo)
{
    nativepair r;
    native hierr;
    r.hi = hi + lo;
    hierr = hi – r.hi;
    r.lo = hierr + lo;
    return (r);
}
```

Algorithm 2 shows the use of the residual register in the nativepair_normalize function. The `hierr` variable denotes the error residual computed from hi component. The `getrr ( )` function is assumed to be an inline function that returns the residual register value using a single `MOVRR` instruction. Compared to the Algorithm 1, Algorithm 2 does not need to compute `hierr` and as a result, the number of instructions is reduced by one relative to Algorithm 1. Every basic operation ends by normalizing the result so this reduction decreases the instruction count for every native-pair operation.

Algorithm 2. Native-pair normalization algorithm using the residual register:

```
   nativepair nativepair_normalize (native hi, native lo)
{
    nativepair r;
```

```
    r.hi = hi + lo;
    r.lo = getrr ( );
    return (r);
}
```

Algorithm 3 describes the addition of `b` (native floating point number) to `a` (native-pair number). The algorithm adds `b` to `hi` component of a, computing the residual result and adding the residual result to `lo` component. It then normalizes the final result.

Algorithm 3. Addition of Native-pair number and a native number without residual register hardware.

```
   nativepair nativepair_native_add (nativepair a, native b)
{
    native hi = a.hi + b ;
    native bhi = hi - a.hi;
    native ahi = hi – bhi;
    native bhierr = b – bhi;
    native ahierr = a.hi – ahi;
    native hierr = bhierr + ahierr;
    native lo = a.lo + hierr;
    return (nativepair_normalize(hi,lo));
}
```

Algorithm 4 describes the same native-pair and native number addition with the use of residual register. This usage computes the `hierr` component using the `getrr ( )` inline function and so eliminates the use of `ahierr, bhierr` i.e., instructions to compute `ahi, bhi, ahierr, bhierr`. As a result, number of instructions is reduced by four when with respect to Algorithm 3 which does not use residual register.

Algorithm 4. Addition of Native-pair number and a native number with residual register hardware.

```
   nativepair nativepair_native_add (nativepair a, native b)
{
    native hi = a.hi + b;
    native hierr = getrr( );
    native lo = a.lo + hierr;
    return (nativepair_normalize(hi,lo));
}
```

Algorithm 5 shows addition of two native-pair numbers without using the residual register and Algorithm 6 adds two native-pair numbers using the residual register. In Algorithm 5, which shows addition without residual register, the residual from adding the two high components is stored in `ahierr` or `bhierr` depending on the values of `a` and `b`. When `a > b, bhierr` contains the residual and `ahierr` is zero and when `b > a, ahierr` contains the residual and `bhierr` is zero. Such a system of computing is faster than using a condition to decide which one to compute. The addition algorithm with residual register reduces the instruction count to 6 compared to Algorithm 5 which takes 11 instructions.

Algorithm 5. Addition of two Native-pair numbers without residual register hardware.

```
   nativepair nativepair_add (nativepair a, nativepair b)
{
    native hi = a.hi + b.hi;
    native lo = a.lo + b.lo;
    native bhi = hi - a.hi;
    native ahi = hi – bhi;
    native bhierr = b.hi – bhi;
```

```
    native ahierr = a.hi – ahi;
    native hierr = bhierr + ahierr;
    lo = lo + hierr;
    return (nativepair_normalize(hi,lo));
}
```

Algorithm 6. Addition of two Native-pair numbers with residual register hardware.

```
nativepair nativepair_add (nativepair a, nativepair b)
{
    native hi = a.hi + b.hi;
    native hierr = getrr( );
    native lo = a.lo + b.lo;
     lo = lo + hierr;
    return (nativepair_normalize(hi,lo));
}
```

Figure 6 shows the dataflow of the native-pair addition algorithm with and without residual register. Each ADD or SUB instruction typically would have a latency of 4 clock cycles. The MOVRR instruction is assumed to have a latency of 2 clock cycles as a worst case. Native-pair addition without residual register requires 9 instructions in its critical path and with a latency of 36 = 9 × 4 clock cycles. Addition with residual register requires 3 ADD/SUB instructions and 2 MOVRR instructions yielding to a total latency of 16 = 3 × 4 + 2 × 2 clock cycles. But this latency can be decreased without changing the critical path by delaying lo portions of an input to the algorithm in the dataflow. This reduces the latency to 28 = 36−8 cycles in native-pair addition without residual register and 14 =16−2 cycles in the native-pair addition with residual register. This results in exactly 2 × speedup over the algorithm not using residual register [23].

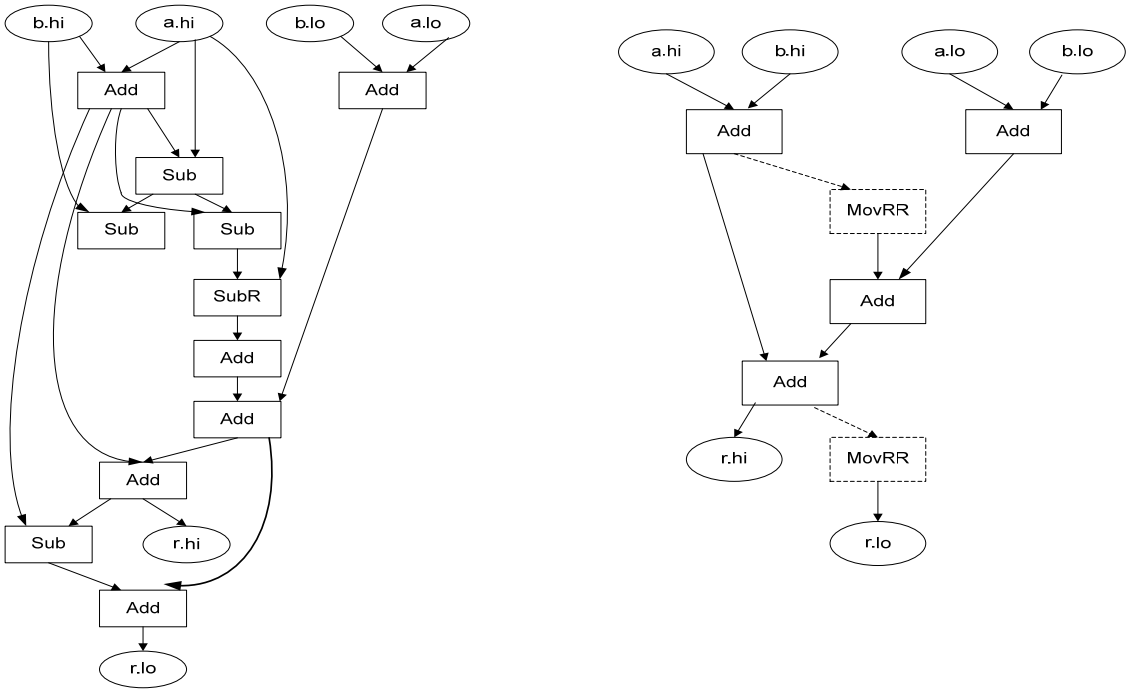


Figure 6. Native-pair addition data flow, conventional and residual algorithms.

2.6. Native-pair Multiplication

In multiplication of two floating-point numbers as opposed to addition, there is no shifting of the mantissa bits in order to make the exponents of the two numbers equal. Multiplication of two n-bit mantissa numbers produces a 2n-bit result and the exponents of the two numbers are simply added. The lower n-bits of the 2n-bit result are put into the residual register and its exponent is set to *exp (a) - (N_m+1)*. When the result is rounded down, the sign of the residual register is same as that of the result and the complement flag is cleared. On the other hand when the result is rounded up, the sign is set opposite to the sign of the result and the complement flag is set.

Table 4. Cases for complement flags and signs of residual register

Case	**Sign of product**	**Rounded up / down**	**Complement flag**	**Sign of Residual register : *Sign(rr)***
Case 1	*Sign(p)*	*Down*	Cleared	*Sign(p)*
Case 2	*Sign(p)*	*Up*	Set	*Opposite of Sign(a)*

Multiplication algorithms for native-pair multiplication

Algorithm 7 shows the multiplication of two native-pair numbers `a` and `b` without residual register hardware. The algorithm uses a `native_mul` function to multiply two high components of the two native-pair numbers. The multiplication of the high and low components also takes place producing three low components namely native_mul result low component, `a.hi × b.lo` and `b.hi × a.lo`. The fourth term, `a.hi × b.lo`, is too small to have an influence on the result. All the three low components are added to produce the final low component of the result. The `native_mul` function implementation is simplified if the processor has a fused multiply- subtract instruction that preserves the full precision of the product before addition. In such a case the residual value can be obtained by subtracting the rounded product from the full precision product. When such a provision is unavailable the `native_mul` function requires the entire component-wise multiplication of the high and low components.

Algorithm 7. Native-pair multiplication without residual register hardware

```
   nativepair nativepair_mul (nativepair a, nativepair b)
{
    nativepair tops = native_mul (a.hi, b.hi);
    native hiloa = a.hi * b.lo;
    native hilob = b.hi * a.lo;
    native hilo = hiloa + hilob;
    tops.lo = tops.lo + hilo;
     return (nativepair_normalize (tops.hi, tops.lo));
}
```

Algorithm 7.1. native_mul function

```
   #define NATIVEBITS 24
   #define NATIVESPLIT ((1<<(NATIVEBITS - (NATIVEBITS/2))) +
   1.0)
```

```
  nativepair native_mul (native a, native b)
{
    nativepair c;
#ifdef HAS_FUSED_MULSUB
    c.hi = a * b;
    c.lo = a * b – c.hi;
#else
    native asplit = a * NATIVESPLIT;
    native bsplit = b * NATIVESPLIT;
    native as = a – asplit;
    native bs = b – bsplit;
    native atop = as + asplit;
    native btop = b + bsplit;
    native abot = a – atop;
    native bbot = b – btop;
    native top = atop * btop;
    native mida = atop * bbot;
    native midb = btop * abot;
    native mid = mida+ midb;
    native bot = abot * bbot;
    c = nativepair_normalize (top, mid);
    c.lo = c.lo + bot;
#end if
    return(c) ;
}
```

When fused multiply-add is not available the residual register hardware simplifies the `native_mul` function from 17 instructions to two instructions. The Algorithm 8 shown below takes 8 instructions to perform the multiplication. Though the instruction count is

the same as the fused multiply-add implementation, the need for a wider adder is removed in the residual register implementation.

Algorithm 8. Native-pair multiplication using residual register hardware

```
   nativepair nativepair_mul (nativepair a, nativepair b)
{
     nativepair tophi = a.hi * b.hi;
     native toplo = getrr ( );
     native hiloa = a.hi * b.lo;
     native hilob = b.hi * a.lo;
     native hilo = hiloa + hilob;
     tops.lo = toplo + hilo;
     return (nativepair_normalize (tophi, toplo));
}
```

Nativepair multiplication has three data flow graphs: conventional, fused multiply-add and residual register implementation which are shown in Figure 7 in the next page. Depending on the latency of add and subtract operations in the critical path, the speed up resulting from the fused multiply-add implementation is 2.3 and that resulting from residual register implementation is 3 [23]. The residual register implementation also has an added advantage that the critical path can be implemented with only a 2-stage pipeline with careful instruction scheduling. Such improvisation is not possible in conventional and fused multiply-add implementations as they suffer from greater need for a larger pipeline [23].

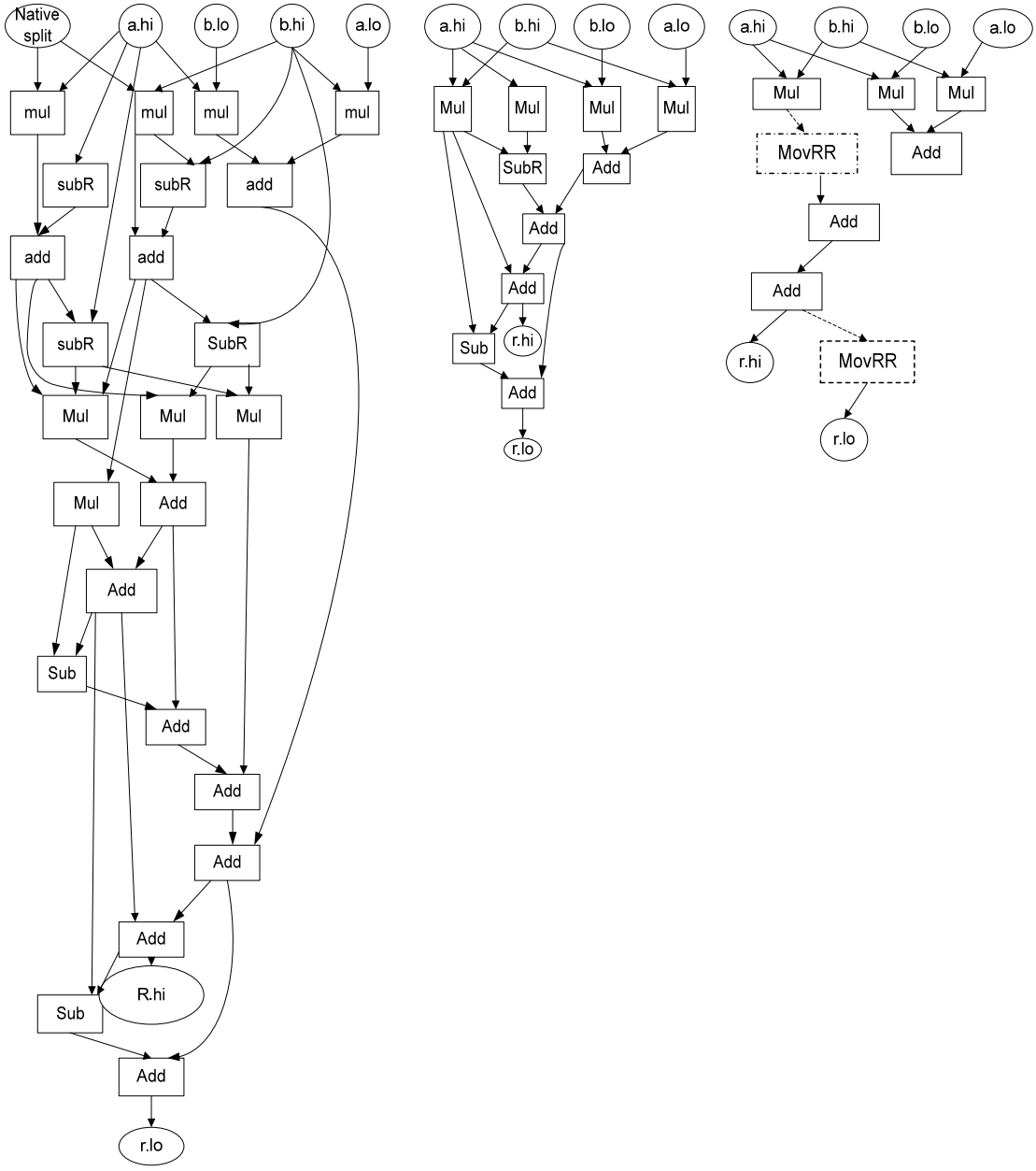


Figure 7. Native-pair multiplication data flow, conventional and residual algorithms

Chapter 3. Native-pair Floating-point Unit

This chapter describes the working of the native floating-point unit addition/subtraction and multiplication units, followed by the construction of the native-pair floating-point unit and usage of the residual register hardware.

3.1. Native Floating-Point Addition/Subtraction

The native floating-point addition/subtraction is subdivided into three steps: prenormalization, addition or subtraction and postnormalization.

3.1.1. Prenormalization

The input operands to the floating-point unit first go to the prenormalization unit. This unit finds the difference between the exponents of the two operands, shift the mantissa with lower exponent to make the two exponents equal and send the mantissa bits and the exponent o the addition stage.

Initially the two operands A and B are divided into sign, exponent and mantissa fields. After the last step the following fields or signals are obtained:

- Exp (A)
- Exp (B)
- Mant(A)
- Mant(B)

The exp (A) and exp (B) of all the input operands are checked for zero values to see if they are denormalized. If an operand is denormalized, its exponent is incremented by 1 to make the exponent equal to -126 after unbiasing. If exp(A) and exp(B) are non-zero values ,the corresponding operands are considered normalized. The fraction values are concatenated with 5 more bits – carry, hidden, guard, round and sticky bits. Carry and hidden bits are added as most significant bits. Initially the carry bit is 0 and the hidden bit is 0 if the operand is denormalized otherwise the hidden bit is 1. The guard, round and sticky bits are appended at the end of the fraction bits and are initially all zeroes. After this step fractions take the form of

- New Mant(B) = carry, hidden, mant(B), guard, round, sticky.
- New Mant(B) = carry, hidden, mant(B), guard, round, sticky.

A comparator COMP1 is used to check which exponent is greater and a multiplexer MUX1 is used to assign the greater exponent to the output exponent based on the comparator output signal. Multiplexer MUX2 is used to give the difference of the two exponents. If exp(A) > exp(B), then MUX2 gives the difference exp(A) – exp(B) if not it gives the difference exp(B) – exp(A). The fraction bits of the lower exponent operand's mantissa are shifted right as many bits as the difference obtained from the exponent difference. The sticky bit for the shifted mantissa is computed and updated. The two updated mantissas with the output exponent and other signals are sent to the next stage. Figure 8 shows the prenormalization process.

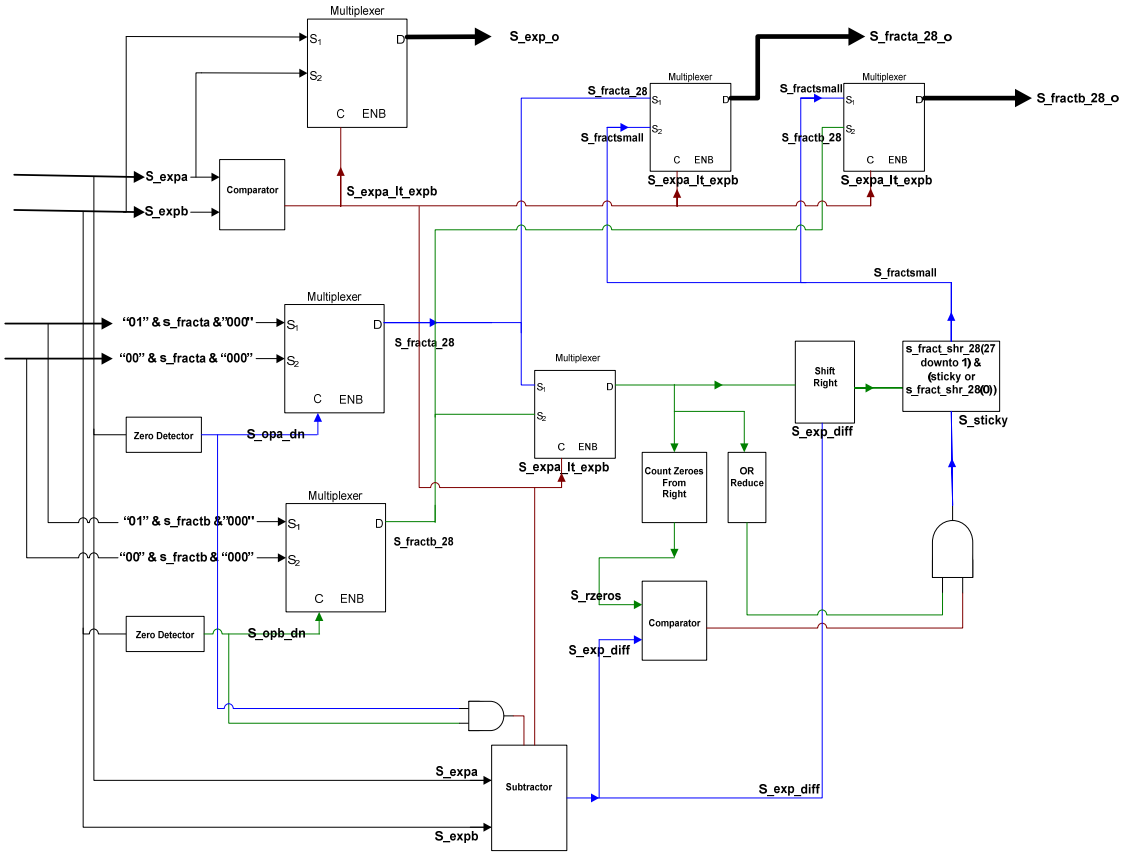


Figure 8. Prenormalization unit for Floating-point addition

3.1.2. Addition/Subtraction Stage

This stage has a simple functionality of computing the sign of the output and performing the addition or subtraction of mantissas based on the sign of the operands. The two mantissas are compared and the operation i.e., addition or subtraction to be performed is computed based on which mantissa is greater, signs of the two operands A and B and the opcode. Table 5 shows the different cases that arise. A and B having same signs with an opcode of 0, indicating addition is performed. If the opcode is a 1 then subtraction is performed. On the other hand A and B having opposite signs, for opcode of 0, subtraction is performed and for opcode of 1 addition is done.

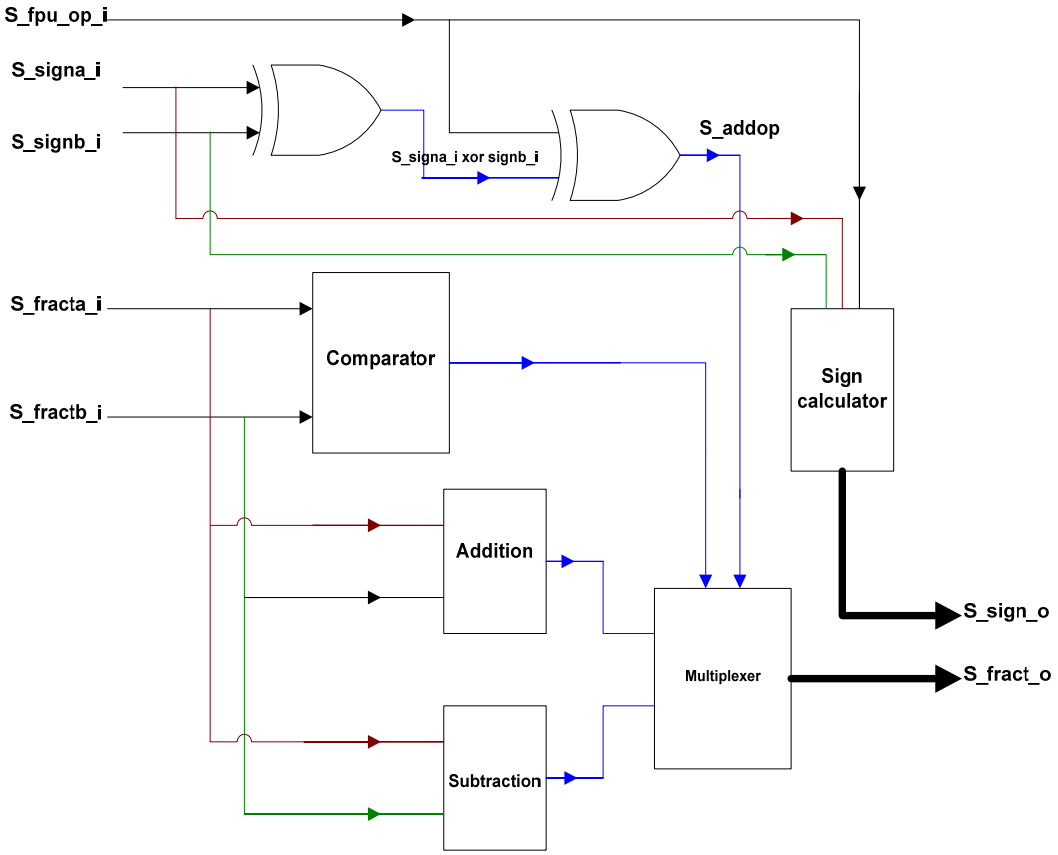


Figure 9. Addition unit for Floating-point addition

The sign of the output is computed based on the signs of the operands A and B, which mantissa is greater and the operation being performed. If the operation is an addition then the two mantissas are added and if it is a subtraction then the lower mantissa is subtracted from the higher mantissa. The output sign and mantissa are sent to the postnormalization

stage along with input operands passed by the prenormalization stage**.** Input operands are required in postnormalization for generation of exceptions.

Table 5. Addition or subtraction cases based on opcode and signs of the operands

Opcode	Sign of A	Sign of B	Operation
0	sign	sign	Addition
0	sign	~ sign	Subtraction
1	sign	sign	Subtraction
1	sign	~sign	Addition

3.1.3. Postnormalization

Postnormalization is the final stage of any floating-point operation. The inputs to this stage are the addition/subtraction unit output, the output exponent, the output sign and the rounding mode.

The postnormalization unit checks the result of the addition/subtraction stage for a carry. If the carry bit in the result is set then, shift the result right once and increase the output exponent by one. If the result has the hidden bit equal to zero then, the result must be left shifted until the hidden bit is one. To determine how far to shift the mantissa, the number of zeros starting from the most significant bit is counted. After the shift is performed, the exponent is decreased by the same number. Once again the sticky bit is checked to find if any bits were lost. Depending on the rounding mode and the sticky bits at different stages in the postnormalization, the result is rounded up or rounded down. The carry bit is checked again to see if carry occurred and if carry has occurred then the result is shift right once and the exponent is incremented by one. Finally, the result is checked for exceptions such as NaN, infinite, overflow, inexact result and depending on these values, the final result along with the exception flags are send to the output. The postnormalization unit is shown in Figure 10.

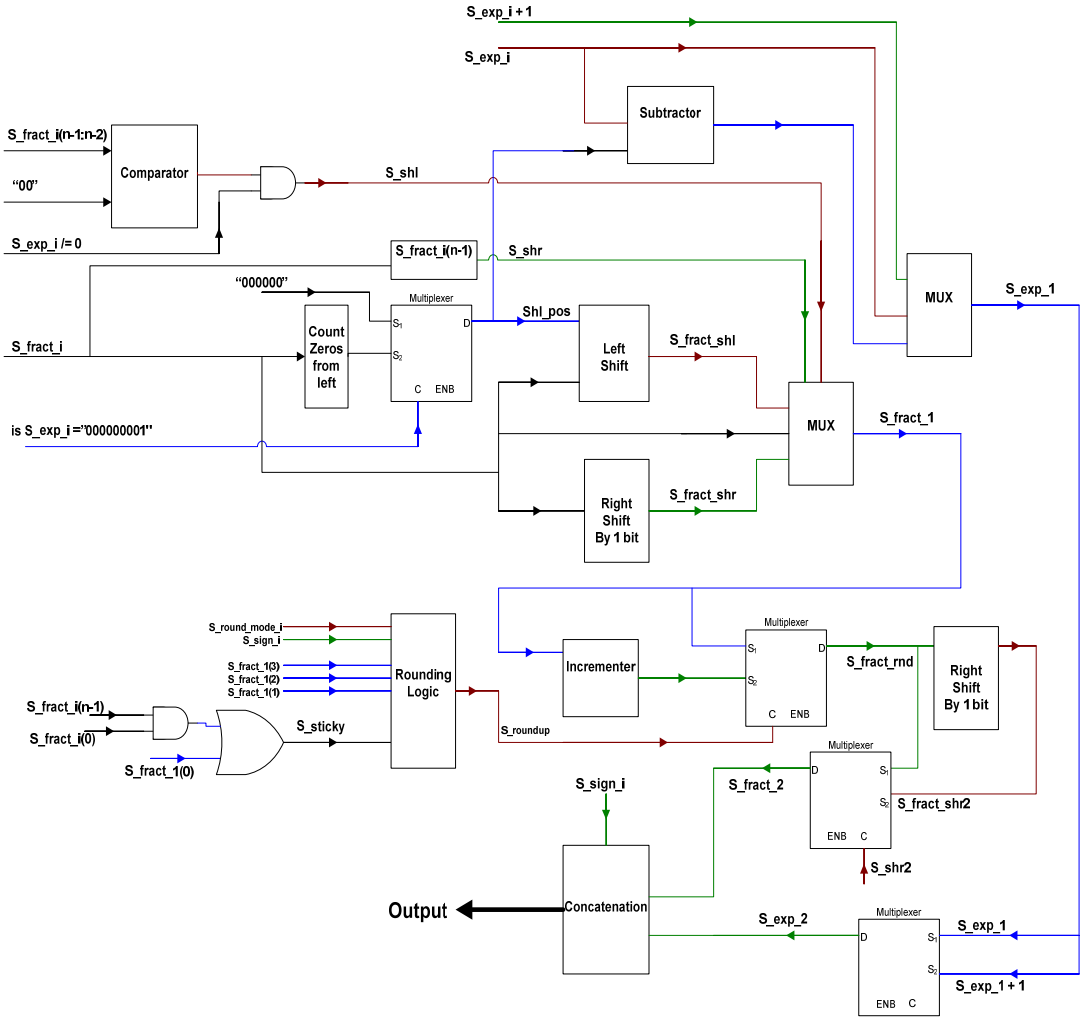


Figure 10. Postnormalization unit for Floating-point addition.

3.2. Native Floating-Point Multiplication

The native floating-point multiplication unit is also subdivided into three steps: prenormalization, multiplication and postnormalization.

3.2.1. Prenormalization

The input operands to the multiplication unit first go through the prenormalization unit. As compared to prenormalization in addition, the prenormalization in multiplication has less functionality. This unit checks if the operands A and B are denormalized, adds the exponents of A and B and transfers the mantissas to the multiplication stage.

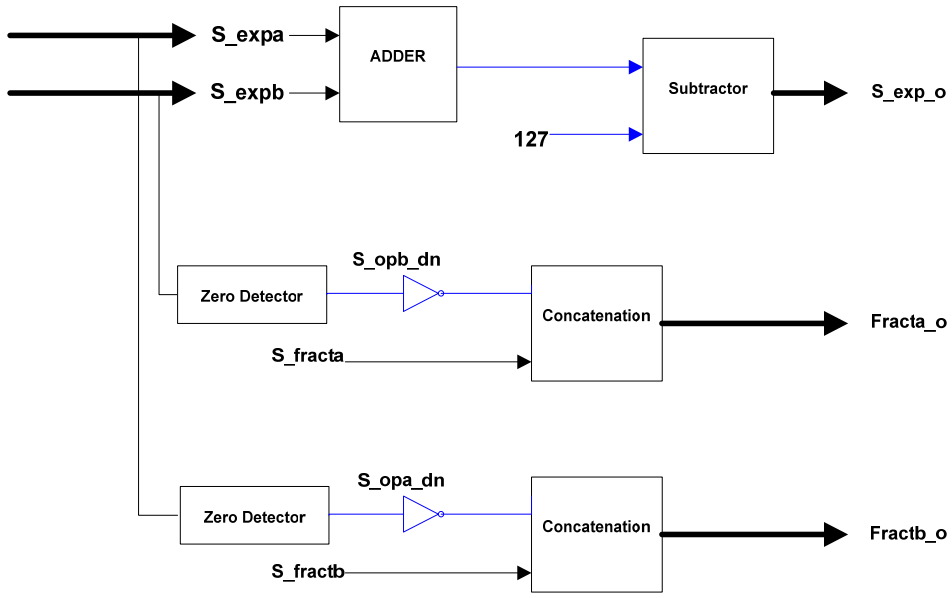


Figure 11. Prenormalization unit in floating-point multiplication.

Initially the two operands A and B are divided into sign, exponent and mantissa fields. After the last step the following fields or signals are obtained:

- Exp(A)
- Exp(B)
- Mant(A)
- Mant(B)

The exp (A) and exp (B) of the all the incoming operands are checked for zero values to see if they are denormalized. If an operand is denormalized, its exponent is incremented by 1 to make the exponent equal to -126 after unbiasing. The fraction values are appended with just 1 more bit, the hidden bit as the most significant bit. The hidden bit is 0 if the operand is denormalized otherwise the hidden bit is 1. After this step fractions take the form of

- New Mant(B) = hidden, mant(B)
- New Mant(B) = hidden, mant(B)

The exponents are added, but since the exponents are already biased i.e., we are baising the exponent twice and so 127 is subtracted from the sum. The two updated mantissas with the output exponent and other signals are sent to the next stage.

3.2.2. Multiplication Stage

This stage is used to multiply the two mantissas obtained from the previous stage. The multiplier used is a Booth's parallel multiplier model. An exclusive-or gate is used to obtain the sign of the product. The output sign s_sign_o as well as the product s_fract_o are transferred along with the other signals to the postnormalization stage.

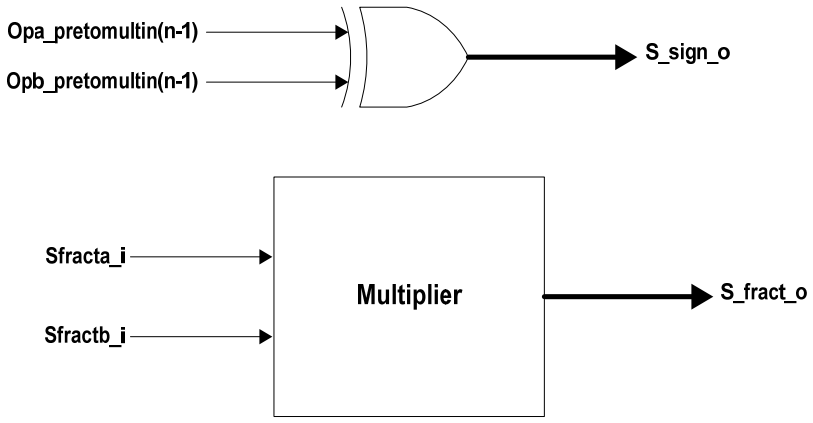


Figure 12. Multiplication unit for Floating-point multiplication

3.2.3. Postnormalization

The inputs to this stage are the multiplication unit output, the prenormalization exponent output, the multiplication sign output and the rounding mode. The postnormalization stage checks the multiplication output for a carry. If a carry has occurred, the multiplication output is shifted right once to normalize it. If the result has the hidden bit equal to zero then, the result must be left shifted until the hidden bit is one. For this, the number of zeros starting from the most significant bit is counted. After the shift is performed, the exponent is decreased by the same number. Once again the sticky bit is checked to find if any bits were lost. Depending on the rounding mode and the sticky bits at different stages in the postnormalization, the result is rounded up or rounded down. The carry bit is checked again to see if a carry occurred. If a carry has occurred then the

result is shift right once and the exponent is incremented by one. Finally, the result is checked for exceptions such as NaN, infinite, overflow, inexact result and depending on these values, the final result along with the exception flags are send to the output.

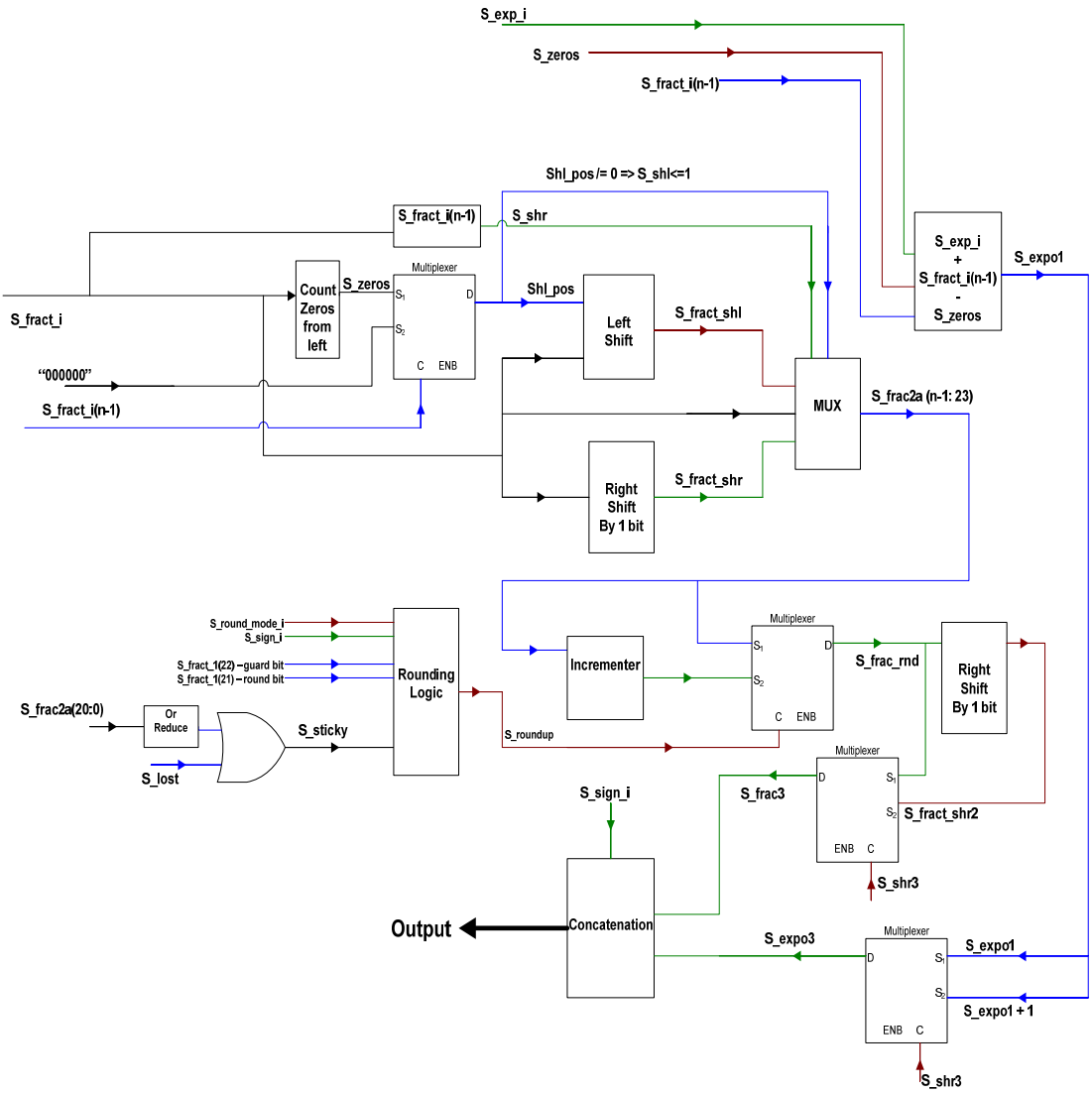


Figure 13. Postnormalization unit for Floating-point multiplication

3.3. Native-pair Floating-point Addition/subtraction

Native floating-point addition/subtraction has been discussed in Section 3.1.1. This section discusses the 32-bit native-pair floating-point addition/subtraction and the extra hardware added to the native floating-point addition/subtraction unit to make it work as a native-pair floating-point addition/subtraction unit. Native-pair addition/subtraction also

is subdivided into three steps: prenormalization, addition or subtraction and postnormalization. Each step and its hardware addition are discussed in detail below.

3.3.1. Prenormalization

The native-pair prenormalization unit, apart from doing the normal operation of making the exponents equal and aligning the mantissa' s, also includes the first of the residual register operations.

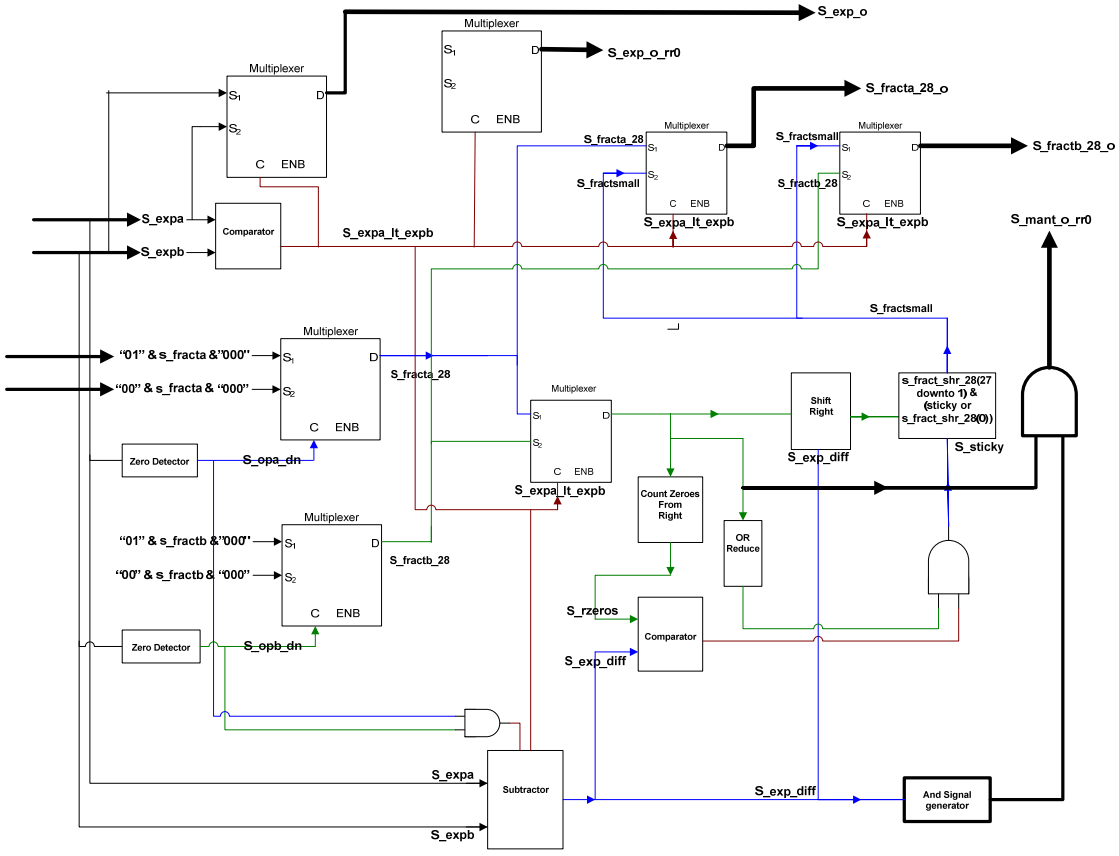


Figure 14. Prenormalization unit for Native-pair Addition using Residual register

For a better understanding the steps of the native normalization are again repeated. Initially the two operands A and B are divided into sign, exponent and mantissa fields. After the last step the following fields or signals are obtained:

- Exp(A)
- Exp(B)
- Mant(A)

- Mant(B)

The exponents exp(A) and exp(B) of all the input operands are checked for zero values to see if they are denormalized. The exponent is incremented by 1 to make the exponent equal to -126 after unbiasing. The fraction values are concatenated with 5 more bits – carry, hidden, guard, round and sticky bits. Carry and hidden bits are added as most significant bits. Initially the carry bit is 0 and the hidden bit is 0 if the operand is denormalized else hidden bit is 1. The guard, round and sticky bits are appended at the end of the fraction bits and are initially all zeroes. After this step fractions take the form of

- New Mant(B) = carry, hidden, mant(B), guard, round, sticky.
- New Mant(B) = carry, hidden, mant(B), guard, round, sticky.

A comparator COMP1 is used to check which exponent is greater and a multiplexer MUX1 is used to assign the greater exponent to the output exponent based on the comparator output signal. Multiplexer MUX2 is used to give the difference of the two exponents. If exp(A) > exp(B), then MUX2 gives the difference exp(A) – exp(B) if not it gives the difference exp(B) – exp(A). An '`andsignal`' is generated which is a 25 bit signal consisting of zeroes and ones in till the position of the exponent difference (`s_exp_diff-1`) i.e. if the exponent difference is 4 then the andsignal is "00000000000000000000001111". The fraction bits of the lower exponent operand's mantissa s_fract_small are shifted right as many bits as the difference obtained from the exponent difference. A bit-wise AND operation is performed between the smaller mantissa s_fract_small and the andsignal, the result is the initial mantissa part for the residual register. The bits that are being shifted out are stored in the mantissa of the residual register. The exponent of the residual register is set to the exponent of the lower mantissa. The sticky bit for the shifted mantissa is computed and updated. One other signal that is generated here is exp_greater_24 which indicates if the exponent difference is greater than 24. The two updated mantissas with the output exponent, residual register exponent, mantissa and other signals are sent to the next stage.

3.3.2. Addition/subtraction Stage

There is no change in the functionality of the addition/subtraction unit. It takes in the mantissas and the operand signs as inputs. After logically generating which operation has to be performed, it performs that operation i.e., addition or subtraction. Output sign is generated based on the operation, the signs of the operands and which operand is greater. The operation output, the output sign and other inputs such as the residual register values from the prenormalization stage etc., are all passed to the postnormalization stage.

3.3.3. Postnormalization

Postnormalization in the Section 3.3.3 involves all the main operation surrounding the residual register hardware operation. The inputs to this stage are the addition/subtraction unit output, prenormalization output exponent, addition unit output sign, rounding mode, residual register values from the prenormalization stage. The following are the steps involved in the postnormalization stage:

Check the result of the addition/subtraction stage for a carry. If carry bit in the result is set then, shift the result right by once and increase the output exponent by one. If the result has the hidden bit equal to zero then, the result must be left shifted until the hidden bit is one. For this, the number of zeros starting from the most significant bit is counted. After the shift is performed, the exponent is decreased by the same number.

As the result is shifted, one bit before the guard bit is lost and this bit has to be prepended to the residual register. This bit has to be prepended before the bits that were inserted in the prenormalization stage. For this purpose a decoder is used which whose output d1 has a value in the position which corresponds to the exponent difference. D1 is logically ANDed with the output of the addition/subtraction and then ORed with the mantissa from the prenormalization stage to get the new updated mantissa.

```
s_mant_rr2_br <= ('0' & mant_i_rr2) or (d1 and s_fract_28_i (27 downto 3));
```

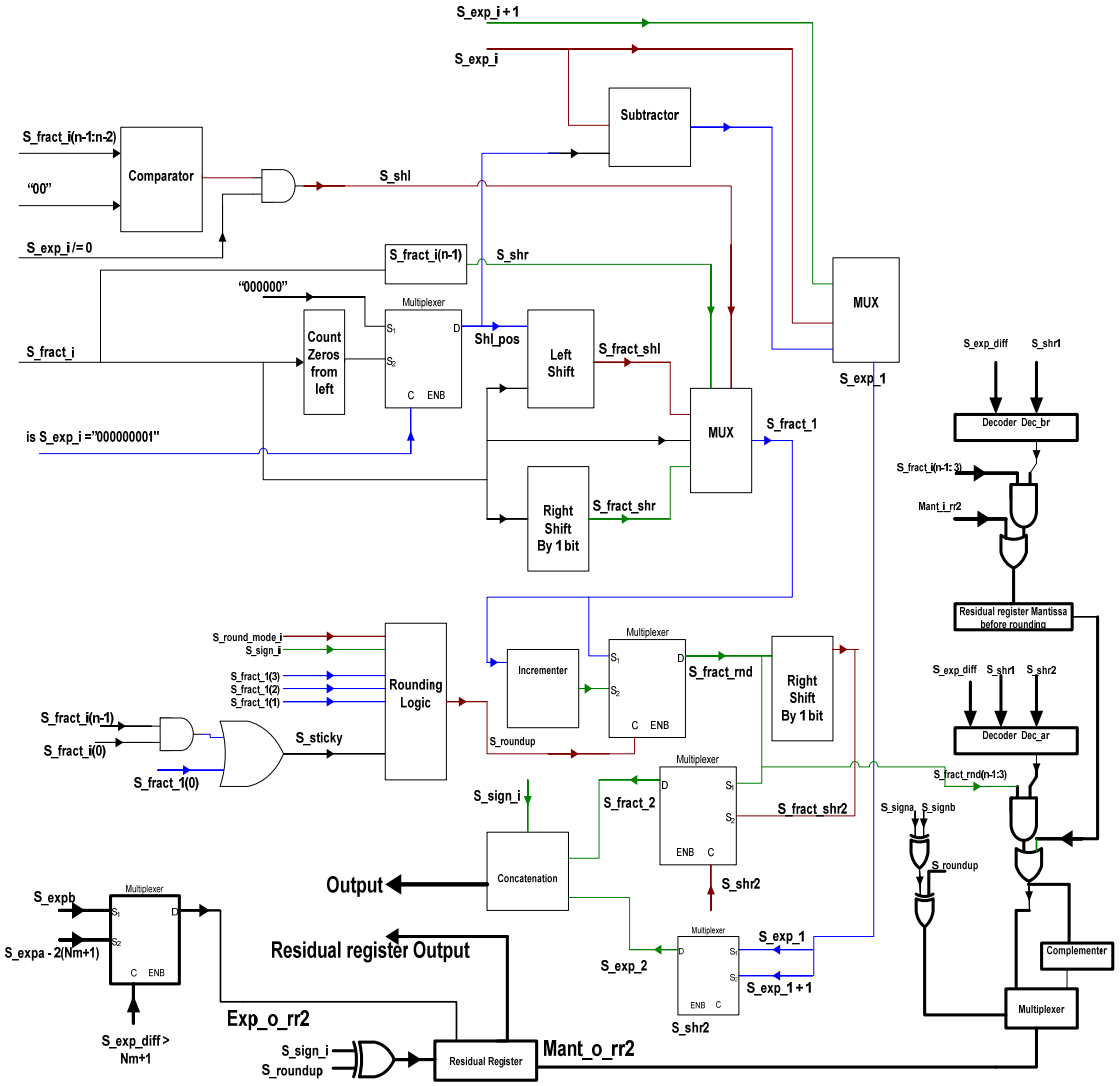


Figure 15. Postnormalization unit for Native-pair addition with Residual register

Once again the sticky bit is checked to find if any bits were lost. Depending on the rounding mode and the sticky bits at different stages in the postnormalization, the result is rounded up or rounded down. The carry bit is checked again to see if carry occurred and if carry has occurred then the result is shift right once and the exponent is incremented by one.

As the result is shifted right again, one bit before the guard bit is lost and this bit has to be added to the residual register. This bit has to be added before the bit that was added after the right shift performed before rounding. For this purpose another decoder is used whose output D2 has a value '1' in the position which next to '1' in D1. D2 is logically

ANDed with the output of the after the rounded result is shifted right and then ORed with the mantissa s_mant_rr2_br to get the new updated mantissa.

s_mant_rr2_ar <= ('0' & s_mant_rr2_ar) or (d2 and s_fract_rnd (27 downto 3));

Suppose that the exponent difference in the prenormalization was greater than 24, and all the bits of the smaller mantissa are shifted into the residual register. Now in the postnormalization stage if the result was shifted right twice once before rounding and once after rounding, then 2 bits must be stuck on the So. In total the residual register mantissa temporarily can have 27 bits and then the 25 most significant bits are stored as final residual value.

The sign of the residual register and the complement flag are also generated in this stage. If the complement flag is set, then residual value is complemented before it is stored in an architectural register. The signal exp_greater_24 that was generated in the prenormalization stage to check if the exponential difference was greater than the number of mantissa bits + 1 is used here.

If the signal is set, then the exponent of the residual is set to higher exponent – $2(N_m+1)$ else exponent is set to lower exponent, where N_m is the number of mantissa bits.

Finally, the result is checked for exceptions such as NaN, infinite, overflow, inexact result and depending on these values, the final result along with the exception flags are send to the output.

The next instruction is to normalize the residual register value which happens with the MOVRR signal going high. During this stage, residual register value is concatenated with the guard, round and the sticky bits in the end to make it 28 bits and this value is directly sent into the postnormalization input for normalization. This normalized residual register value is later used in computation related to native-pair algorithms.

3.4. Native-pair Floating-point Multiplication

This section discusses the 32-bit native-pair floating-point multiplication and the extra hardware added to the native floating-point multiplication unit to make it work as a native-pair floating-point multiplication unit. The residual register hardware in a multiplication is less complex when compared to addition. Since there is no shifting of mantissas in multiplication, there is no residual register functionality in the prenormalization. So the entire residual register operation takes place only in the postnormalization stage. Hence only the changes and the steps involved to the postnormalization are discussed here.

3.4.1. Postnormalization

The inputs to this stage are the multiplication unit output, the prenormalization output exponent, the multiplication sign output and the rounding mode. The following are the steps involved in the native-pair multiplication postnormalization stage

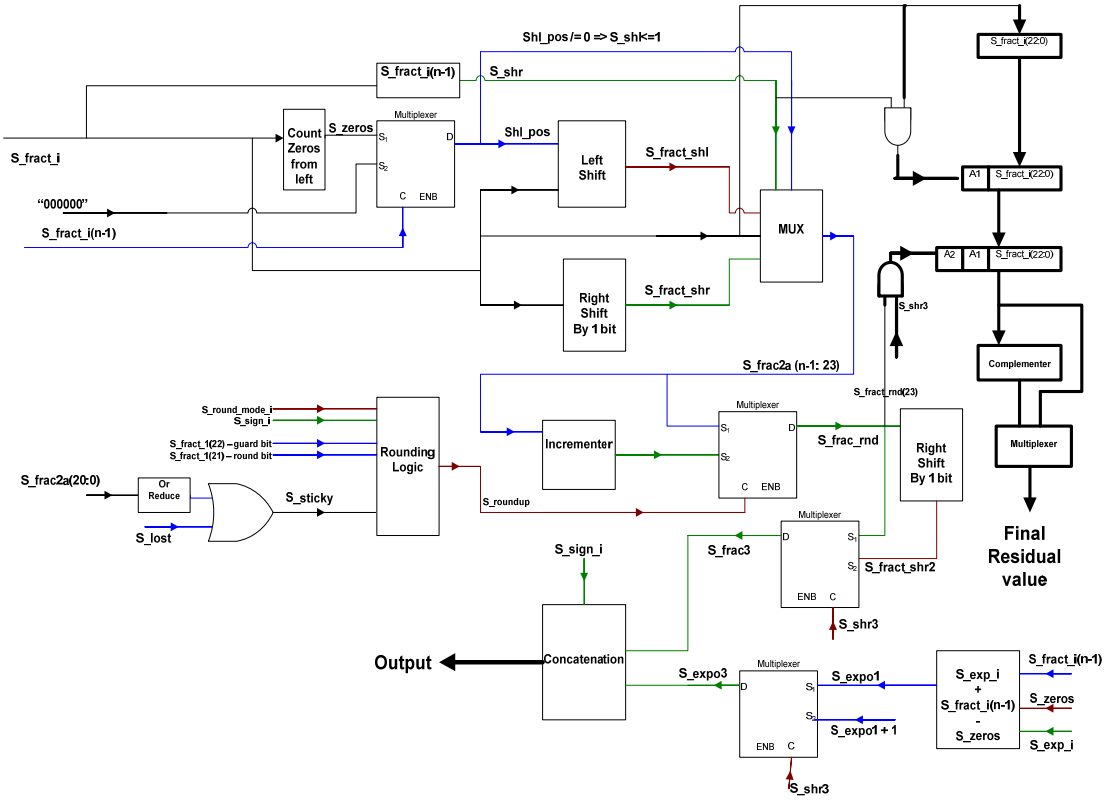


Figure 16. Postnormalization unit for Native-pair multiplication with Residual register

Check the result of the multiplication stage for a carry. If carry bit in the result is set then, shift the result right by once and increase the output exponent by one. In postnormalization, the 25 most significant bits are taken into consideration for the final output, hence initially the residual register consists of the 23 least significant bits.

If a carry had occurred then, the result would be shifted right once, and the bit that comes out is stored into the residual register. Compared to the addition, multiplication does not involve shifting of mantissa in the prenormalization stage and so no decoder is required here to append the discarded bit into the residual register. If the result has the hidden bit equal to zero then, the result must be left shifted until the hidden bit is one. For this, the number of zeros starting from the most significant bit is counted. After the shift is performed, the exponent is decreased by the same number.

The sticky bit is checked to find if any bits were lost. Depending on the rounding mode and the sticky bits at different stages in the postnormalization, the result is rounded up or rounded down. The carry bit is checked again to see if carry occurred and if carry has occurred then the result is shift right once and the exponent is incremented by one. When a carry occurs second time and the result is shifted again, the discarded bit is again appended as the most significant bit into the residual register, this becomes the 25^{th} bit.

The complement flag and the sign flag are generated. When he complement flag is set, then the final residual value being stored into the residual register is complemented. The exponent of the residual is set to higher exponent – (N_m+1) to align the residual register mantissa with the result, again N_m denotes the number of mantissa bits.

Finally, the result is checked for exceptions such as NaN, infinite, overflow, inexact result and depending on these values, the final result along with the exception flags are send to the output. The MOVRR signal can be used to re-route the residual register values into the postnormalization for getting the normalized value of the residual.

3.5. Debugging FPU Unit

The adopted FPU – floating-point unit suffered from some architectural errors related to the routing of signals through the pipelines, carrying of input signals to the various stages of the pipeline. This section discusses the changes made to the original Floating-point unit in order to enable its proper functioning in pipelined fashion. The floating-point unit pipeline consists of four stages: prenormalization stage, addition/multiplication stage, postnormalization stage and final output stage. All of these are instantiated in the FPU module. The clock, the input operands, the movrr signal, the rounding mode and the opcode are the inputs to the FPU module. These inputs are sent into the various stages depending on their usage. Apart from these primary inputs, at each stage intermediate outputs such as the exponents, mantissas, signs, operation results are generated and need to be carried to the later stages. The following sections discuss the changes made to the original FPU architecture.

In pipelined operation, the inputs change every clock cycle. Also different instruction or function or set of parallel instructions are executed in each clock cycle in different stages of the pipeline. Hence, an operation performed in the third clock cycle might need an input given in an earlier clock cycle. For this purpose, the needed inputs must be propagated through each stage or each clock cycle using registers until it is used.

The input operands are required in the prenormalization stage to generate the sign, exponent and mantissa bits. The input operands are also required in the postnormalization stage to generate the NaN – Not a number signals. Similarly the FPU operation signal fpu_op_i and the rounding mode signal s_rmode_i are required in addition/multiplication stage and the postnormalization stage. All these signals have to be propagated from prenormalization through addition/multiplication stage to postnormalization.

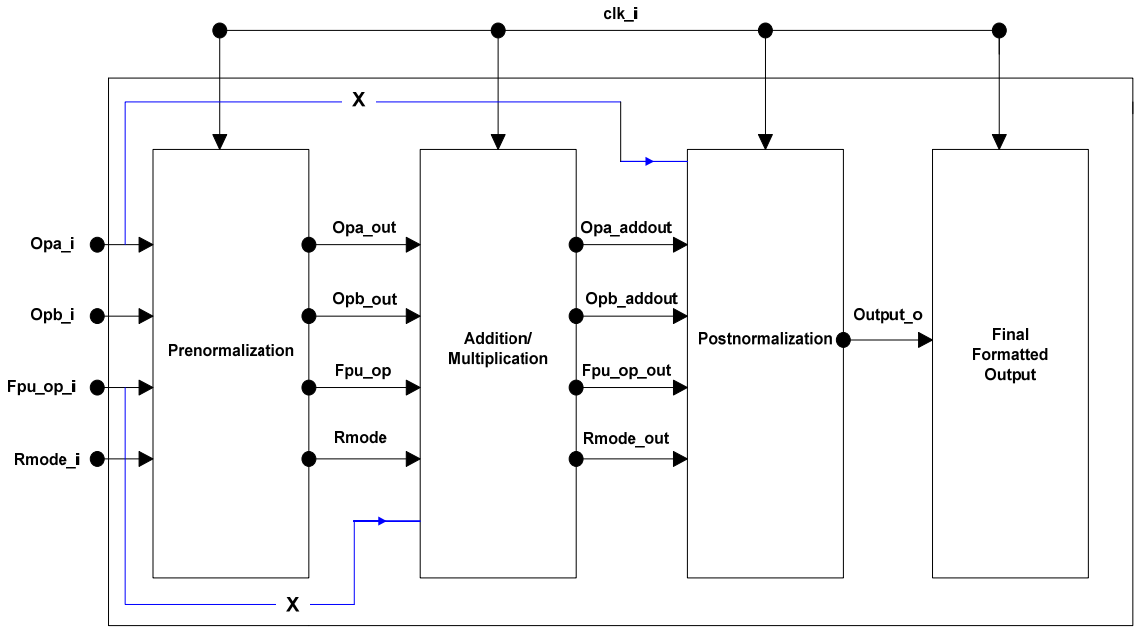


Figure 17. Floating point arithmetic unit pipeline

Figure 17 shows the four stages of the floating-point pipeline: prenormalization, arithmetic core, postnormalization and formatting output. In the formatted output pipeline stage, changes will be made in the output with respect to exceptions [19]. The right way of propagating signals is through the pipelines stages and not those marked X in the figure 17**.** Supposing the inputs to the FPU are opa_i, opb_i, fpu_op_i and rmode_i, fpu_op_i is used in addition/multiplication stage and the opa_i is used in the postnormalization stage. The operation is performed in the second clock-cycle and the postnormalization in the third clock-cycle. When performing the operation, the input through the pipeline is fpu_op whereas the other input could be fpu_op_i. In the second clock cycle, the value of the opcode fpu_op_i can change and so a wrong operation is performed. Fpu_op on the other hand was assigned original fpu_op_i at the end of the clock cycle and so its value does not change and is the opcode for correct operation. Similarly, opa_i changes value until it reaches postnormalization, hence it is propagated through the pipeline stages via opa_out and opa_addout. All the signals that are added or modified to fix this problem are added with a comment "propagated input through pipeline register" in Modified FPU VHDL code given in Appendix B.

When a value or a signal generated in one pipeline stage is required in another stage, then that signal also has to be propagated through the pipeline stages. For example, the output exponent or the larger exponent output of the prenormalization stage is required in the postnormalization stage and hence this has to be taken through the addition/multiplication unit to the postnormalization stage. This is done using prenorm_addsub_exp signal to take the exponent from prenormalization to addition stage and then by using exp_o_addsubpost signal to take it from addition to postnormalization. These modified signals are commented as “intermediate outputs through pipeline register in the modified code.

All the pipeline stages should consume same number of clock cycles to produce the outputs of a particular stage. If different pipeline stages consume different number of clock cycles to produce the outputs of corresponding pipelines, then all the pipelines stages should wait until all the pipelines stages are done with producing their outputs to ensure the correct functioning of the pipelined system. This decreases the throughput of the system as outputs will be produced at a reduced frequency than that of the clock frequency. New output will be produced for every n clock cycles where n is the number of clock cycles consumed by the pipeline stage that consumes highest number of clock cycles to produce its output.

In the original adopted FPU [19], prenormalization takes two clock cycles, arithmetic core takes one clock cycle, postnormalization takes three clock cycles and formatting output takes one clock cycle to give their outputs. All the pipeline stages have been modified such that each pipeline consumes only one clock cycle. For example, two sequential process blocks used in Postnormalization caused two extra clock cycles to get the output of that stage. The signal that is computed in the first sequential process block is needed to compute the signals in second sequential process block and hence needs two clock cycles to get the output of that stage. Those two sequential process blocks are replaced by combinational logic as explained below to reduce the number of clock cycles required by postnormalization pipeline stage to one.

When the hidden bit and carry bit of the arithmetic result are zeros, then the mantissa has to be left-shifted to normalize it. The following sequential process block is used in postnormalization unit of FPU [19] to compute the number of positions by which the mantissa has to be shifted left.

Listing 1. Process to count zeroes from the left.

```
process(clk_i)
begin
if rising_edge(clk_i) then
-- count the leading zeros of fraction, needed for left-shift
s_zeros <= count_l_zeros(s_fract_28_i(26 downto 0));
end if;
end process;
```

The above process block is replaced by the following line of combinational logic.

```
s_zeros <= count_l_zeros(s_fract_28_i(26 downto 0));
```

This change reduced the number of clock cycles required by the postnormalization unit to two. After the above mentioned sequential process block, combinational logic is used in the FPU to compute the left shifted mantissa (s_fract_shl) and corresponding decremented exponent (s_exp_shl) using the count of leading zeros of fraction (s_zeros) computed in above process. After the combinational logic, the following sequential process blocks are used in FPU to compute the normalized fraction and corresponding exponent using left shifted mantissa (s_fract_shl) and decremented exponent (s_exp_shl) computed using combinational logic.

Listing 2. Process to compute normalized fraction

```
process (clk_i)
begin
if rising_edge(clk_i) then
if s_shr1='1' then -- if carry bit is set, then right shift
s_fract_1 <= s_fract_shr1; -- assign right shifted fraction
elsif s_shl='1' then -- if carry bit and hidden bits are zeros, then left shift
s_fract_1 <= s_fract_shl; -- assign right shifted fraction
else
s_fract_1 <= s_fract_28_i; -- assign already normalized fraction
end if;
```

```
end if;
end process;


process (clk_i) -- process to compute normalized exponent
begin
if rising_edge(clk_i) then
if s_shr1='1' or s_shr1e='1' then -- if carry bit is set,
then right shift
s_exp_1 <= s_exp_shr1; -- assign incremented exponent
elsif s_shl='1' then -- if carry bit and hidden bits are
zeros, then left shift
s_exp_1 <= s_exp_shl; -- assign decremented fraction
else
s_exp_1 <= s_exp_i; -- assign already normalized exponent
end if;
end if;
end process;
```

The above two process blocks are used replaced with the following combinational logic.

```
s_fract_1 <= s_fract_shr1 when s_shr1='1' else
                    s_fract_shl when s_shl='1' else
                    s_fract_28_i;

s_exp_1 <= s_exp_shr1 when s_shr1='1' or s_shr1e='1' else
                   s_exp_shl when s_shl='1' else
                   s_exp_i;
```

Such similar changes have been made to the entire floating-point unit to enable its proper functioning. This conversion of sequential logic to combination logic increases the clock frequency. Postnormalization for multiplication is subdivided into more pipelines internally and care is taken to see that no branch prediction hazards occur. The size of the pipeline is influences the hardware cost; greater the size more is the cost. In the FPU, the 32-bit operands given to prenormalization are taken as inputs to postnormalization also to find whether inputs are infinities (two 1-bit signals) or NaNs (two 1-bit signals). Changes have been made to check the operands for infinity and SNaN in prenormalization unit itself. If operands are checked for infinity and SNaN in prenormalization, then 6 bits (four 1-bit signals indicating whether inputs are infinities are not and two 1-bit signals

indicating whether inputs are NaNs are not) can be carried across the pipeline registers instead of 64-bits (two 32-bit operands). This checking of the inputs for infinity and NaN does not increase the length of the critical path in prenormalization.
With all the changes made, the modified FPU was thoroughly tested by sending Gaussian distributed synthetic test data inputs using a VHDL test bench for full pipelined operation and this FPU has been later used for construction of native-pair FPU.

3.6. Examples

Based on the steps and the circuitry described for performing floating-point arithmetic in Chapter 2 the following examples are worked out in a step by step fashion. Each sub-section covers two examples. The same test cases are using for operation without residual register and for operation with residual register. For example, the operands used in addition are again used in addition with residual register to clearly differentiate the functioning of the two approaches.

3.6.1. IEEE 754 Floating-point addition examples

Example 1:

A = 0x (4171999A) = 01000001011100011001100110011010

B = 0x (3FC147AE) = 00111111110000010100011110101110

Step 1: Prenormalization:

Sign (A) = 0; Exp (A) = 10000010 = 130 – 127 = 3; Mantissa (A) = 11100011001100110011010;

Append≠ mantissa (A) with carry, hidden, guard, round and sticky bit. Hidden = 1 if Exp ≠ “00000000” that is number is not a denormalized number.

Mantissa (A) = 01|11100011001100110011010|000

Sign (B) = 0; Exp (B) = 01111111 = 127 – 127 = 0; Mantissa (B) = 10000010100011110101110;

Append mantissa (B) with carry, hidden, guard, round and sticky bit.

Mantissa (B) = 01|10000010100011110101110|000

Exp difference = Exp (A) – Exp (B) = 3 – 0 = 3

Exp (A) > Exp (B) = 1

Smaller mantissa = Mantissa (B) = 01|10000010100011110101110|000

Larger mantissa = Mantissa (A) = 01|11100011001100110011010|000

Shift smaller mantissa right by Exp difference.

RS (Right shifted) Smaller Mantissa = 00|00110000010100011110101|110 000

Exponent to Postnormalization = 3

Step 2: Addition

Output sign = sign (A) if Exp (A) > Exp (B) = 1 else Sign (B) = '0'

Larger mantissa = 01|11100011001100110011010|000

RS Smaller Mantissa = 00|00110000010100011110101|110

Sum = 10|00010011100001010001111|110

Step 3: Postnormalization

Sum = 10|00010011100001010001111|110

Exponent to Postnormalization = 3

Carry =1 => right shift sum once

Right shifted Sum = 01|00001001110000101000111|111 0

Exponent = 3+1 = 4

Sum is rounded up assuming round-to-nearest mode.

Rounded Sum = Right shifted Sum + 1 = 01|00001001110000101001000|111

Carry bit =0 =>No right shift.

Concatenating: Rounded Sum (excluding carry, hidden, guard, round and sticky bits) with Sign and Exponent

0 | 10000011 |00001001110000101001000 = 0x (4184E148)

Example 2:

A = 0x (501502F9) = 01010000000101010000001011111001

B = 0x (219392EF) = 00100001100100111001001011101111

Step 1: Prenormalization

Sign (A) = 0; Exp (A) = 10100000 = 160 – 127 = 33; Mantissa (A) = 00101010000001011111001;

Append mantissa (A) with carry, hidden, guard, round and sticky bit. Hidden = 1 if Exp ≠ "00000000" that is number is not a denormalized number.

Mantissa (A) = 01|00101010000001011111001|000

Sign (B) = 0; Exp (B) = 01000011 = 67 – 127 = -60; Mantissa (B) = 00100111001001011101111;

Append mantissa (B) with carry, hidden, guard, round and sticky bit.

Mantissa (B) = 01|00100111001001011101111|000

Exp difference = Exp (A) – Exp (B) = 33 – (-60) = 93

Exp (A) > Exp (B) = 1

Smaller mantissa = Mantissa (B) = 01|00100111001001011101111|000

Larger mantissa = Mantissa (A) = 01|00101010000001011111001|000

Shift smaller mantissa right by Exp difference.

RS (Right shifted) Smaller Mantissa = 00|000000000000000000000000|001

Discarded bits - 011110001100110011010

Exponent to Postnormalization = 33

Step 2: Addition

Output sign = sign (A) if Exp (A) > Exp (B) = 1 else Sign (B) = '0'

Larger mantissa = 01|00101010000001011111001|000

RS Smaller Mantissa = 00|000000000000000000000000|001

Sum = 01|00101010000001011111001|001

Step 3: Postnormalization

Sum = 01|00101010000001011111001|001

Exponent to Postnormalization = 33

Carry =0=> no right shift sum.

Sum = 01|00101010000001011111001|001

Exponent = 33

Sum is rounded down assuming round-to-nearest mode.

Rounded Sum = Sum = 01|00101010000001011111001|001

Carry bit =0 =>No right shift.

Concatenate: Rounded Sum (excluding carry, hidden, guard, round and sticky bits) with Sign and Exponent

0 | 10100000 |00101010000001011111001 = 0x (501502F9)

3.6.2. Addition with Residual Register examples:

Example 1:

A = 0x (4171999A) = 01000001011100011001100110011010

B = 0x (3FC147AE) = 00111111110000010100011110101110

Step 1: Prenormalization

Sign (A) = 0; Exp (A) = 10000010 = 130 – 127 = 3; Mantissa (A) = 11100011001100110011010;

Append mantissa (A) with carry, hidden, guard, round and sticky bit. Hidden = 1 if Exp ≠ "00000000" that is number is not a denormalized number.

Mantissa (A) = 01|11100011001100110011010|000

Sign (B) = 0; Exp (B) = 01111111 = 127 – 127 = 0; Mantissa (B) = 10000101000011110101110;

Append mantissa (B) with carry, hidden, guard, round and sticky bit.

Mantissa (B) = 01|10000101000011110101110|000

Exp difference = Exp (A) – Exp (B) = 3 – 0 = 3

Exp (A) > Exp (B) = 1

Smaller mantissa = Mantissa (B) = 01|10000101000011110101110|000

Larger mantissa = Mantissa (A) = 01|11100011001100110011010|000

Shift smaller mantissa right by Exp difference.

RS (Right shifted) Smaller Mantissa = 00|00110000010100011110101|110

Discarded bits – 110 go into residual register

Mantissa (RR) = 00000000000000000000000**110**

Exponent to Postnormalization = 3

Step 2: Addition

Output sign = sign (A) if Exp (A) > Exp (B) = 1 else Sign (B) = '0'

Larger mantissa = 01|11100011001100110011010|000

RS Smaller Mantissa = 00|00110000010100011110101|110

Sum = 10|00010011100001010001111|110

Step 3: Postnormalization

Sum = 10|00010011100001010001111|110

Exponent to Postnormalization = 3

Carry =1 => right shift sum once

Right shifted Sum = 01|00001001110000101000111|111

Discarded bit = 1; Goes into residual register.

Mantissa (RR) before rounding = 00000000000000000000000|**1**|110

Exponent = 3+1 = 4

Sum is rounded up assuming round-to-nearest mode.

Rounded Sum = Right shifted Sum + 1 = 01|00001001110000101001000|111

Carry bit =0 =>No right shift.

Mantissa (RR) after rounding = 00000000000000000000001110

Concatenate: Rounded Sum (excluding carry, hidden, guard, round and sticky bits) with Output Sign and Output Exponent.

0 | 10000011 |00001001110000101001000 = 0x (4184E148)

Complement flag for residual register = sign (a) XOR sign (b) XOR Roundup = 1

Residual register sign = Output Sign XOR Roundup = 1

Hence, complement the added bits in residual: 00000000000000000000000**1110**

2's complement (1110) - 0010

Final residual mantissa = 00000000000000000000000**0010**

Exponent (RR) = Exp (A) – $2(N_m + 1)$ when (exponent difference > $N_m + 1$) else Exp (B).

Where N_m is number of mantissa bits in the native-precision floating-point number. For Single-precision N_m =23.

Accordingly, Exponent (RR) = Exp (B) = 0

Final un-normalized residual value = Sign (RR) | Exponent (RR) | Final residual mantissa

Final outputs:

Output = 0 | 10000011 |00001001110000101001000 = 0x (4184E148)

Sign (RR) = 1

Exponent (RR) = "01111111"

Mantissa (RR) = "00000000000000000000000010"

MOVRR = 1

Step 4: Normalization of residual value

Sum = Mantissa (RR) & 000 = 00|00000000000000000000010|000

Exponent = Exponent (RR) = 01111111

Sign = Sign (RR) =0 => left shift till hidden bit =1

22 left shifts. Exponent = Exponent (RR) – 22 = 127 – 22 = 105 = 01101001

Output mantissa = 01|00000000000000000000000|000

Normalized Residual register value = 1|01101001|00000000000000000000000

= 0x (B4800000)

Example 2:

A = 0x (501502F9) = 01010000000101010000001011111001

B = 0x (219392EF) = 00100001100100111001001011101111

Step 1: Prenormalization

Sign (A) = 0; Exp (A) = 10100000 = 160 – 127 = 33;

Mantissa (A) = 00101010000001011111001

Append mantissa (A) with carry, hidden, guard, round and sticky bit. Hidden = 1 if Exp ≠ "00000000" that is number is not a denormalized number.

Mantissa (A) = 01|00101010000001011111001|000

Sign (B) = 0; Exp (B) = 01000011 = 67 – 127 = -60;

Mantissa (B) = 00100111001001011101111

Append mantissa (B) with carry, hidden, guard, round and sticky bit.

Mantissa (B) = 01|00100111001001011101111|000

Exp difference = Exp (A) – Exp (B) = 33 – (-60) = 93

Exp (A) > Exp (B) = 1

Smaller mantissa = Mantissa (B) = 01|00100111001001011101111|000

Larger mantissa = Mantissa (A) = 01|00101010000001011111001|000

Shift smaller mantissa right by Exp difference.

RS (Right shifted) Smaller Mantissa = 00|000000000000000000000000|001

Discarded bits - 010010011100100101111011111 go into residual register

Mantissa (RR) = **01001001110010010111011111**

Exponent to Postnormalization = 33

Step 2: Addition

Output sign = sign (A) if Exp (A) > Exp (B) = 1 else Sign (B) = '0'

Larger mantissa = 01|00101010000001011111001|000

RS Smaller Mantissa = 00|000000000000000000000000|001

Sum = 01|00101010000001011111001|001

Step 3: Postnormalization

Sum = 01|00101010000001011111001|001

Exponent to Postnormalization = 33

Carry =0=> no right shift sum.

Sum = 01|00101010000001011111001|001

Exponent = 33

Mantissa (RR) before rounding = Mantissa (RR) = 01001001110010010111011111

Sum is rounded down assuming round-to-nearest mode.

Rounded Sum = Sum = 01|00101010000001011111001|001

Carry bit =0 =>No right shift.

Mantissa (RR) after rounding = Mantissa (RR) before rounding = 01001001110010010111011111

Concatenate: Rounded Sum (excluding carry, hidden, guard, round and sticky bits) with Sign and Exponent

0 | 10100000 |00101010000001011111001 = 0x (501502F9)

Complement flag for residual register = sign (a) XOR sign (b) XOR Roundup = 0

Residual register sign = Output Sign XOR Roundup = 0

Final residual mantissa = 01001001110010010111011111

Exponent (RR) = Exp (A) – 2(N_m +1) when (exponent difference > N_m +1) else Exp (B).

Where N_m is number of mantissa bits in the native-precision floating-point number. For Single-precision N_m =23.

Accordingly, Exponent (RR) = Exp (A) – 2(N_m +1) = 65 = 01000011

Final un-normalized residual value = Sign (RR) | Exponent (RR) | Final residual mantissa

Final outputs:

Output = 0 | 10100000 |00101010000001011111001 = 0x (501502F9)

Sign (RR) = 0

Exponent (RR) = “01000011”

Mantissa (RR) = “01001001110010010111011111”

MOVRR = 1

Step 4: Normalization of residual value

Sum = Mantissa (RR) & 000 = 01001001110010010111011111|000

Hidden bit = 1 so no shifting Exponent = Exponent (RR) = 01000011

Output mantissa = 01|00000000000000000000000|000

Normalized Residual register value = 0|01000011|00100111001001011101111

= 0x (219392EF)

3.6.3. IEEE 754 Floating-point Multiplication Examples:

Example 1:

A = 0x (4171999A) = 01000001011100011001100110011010

B = 0x (40000011) = 01000000000000000000000000010001

Step 1: Prenormalization:

Sign (A) = 0; Exp (A) = 10000010 = 130 – 127 = 3; Mantissa (A) = 11000110011001100110011010;

Prepend mantissa (A) with a hidden bit. Hidden = 1 if Exp ≠ "00000000" that is number is not a denormalized number.

Mantissa (A) = 1|11100011001100110011010 -- 0x (F1999A)

Sign (B) = 0; Exp (B) = 10000000 = 128-127 = 1; Mantissa (B) = 10000010100011110101110;

Prepend mantissa (B) with a hidden bit.

Mantissa (B) = 1|00000000000000000010001 -- 0x (800011)

Exp (O) – exponent to the postnormalization = 130 + 128 -127 = 131 = 4

Step 2: Multiplication

Output sign = sign (A) XOR Sign (B) = '0' XOR '0' ='0'

Mantissa (A) = 1|11100011001100110011010 – 0x (F1999A)

Mantissa (B) = 1|00000000000000000010001 – 0x (800011)

Product (48 bits) = 01|111000110011001101110100001011001100111010

Step 3: Postnormalization

Product[47:0]= 01|111000110011001101110100001011001100111010

Exponent to Postnormalization = Exp (O) = 4

Carry =0 => No right shifting product[47:0]

Product 2[47:0] = **01**| 11100011001100110111010 |**00**| 01011001100110011101O

01 – carry and hidden bits.

00 – guard and round bits.

Sticky = OR (Product 2 [20:0]) = 1

Roundup = guard and ((round or sticky) or Product 2(23) = 0

Based on Rounding logic, product is rounded down

Rounded product = product 2 [47:23] + 1 = 01| 11100011001100110111010

Lower 23 bits discarded - 00| 01011001100110011101O

Carry bit =0 =>No right shift.

Concatenate: Rounded Sum (excluding carry and hidden bits) with Sign and Exponent

0 | 10000011 |11100011001100110111010 = 0x (41F199BA)

Example 2:

A = 0x (501502F9) = 01010000000101010000001011111001

B = 0x (41A77700) = 01000001101001110111011100000000

Step 1: Prenormalization

Sign (A) = 0; Exp (A) = 10100000 = 160 – 127 = 33;

Mantissa (A) = 00101010000001011111001;

Prepend mantissa(A) with a hidden bit. Hidden = 1 if Exp ≠ “00000000” that is number is not a denormalized number.

Mantissa (A) = 1|00101010000001011111001

Sign (B) = 0; Exp (B) = 10000011 = 131 – 127 = 4;

Mantissa (B) = 01001110111011100000000;

Prepend mantissa(B) with a hidden bit.

Mantissa (B) = 1|01001110111011100000000

Exp (A) > Exp (B) = 1

Mantissa (B) = 1|01001110111011100000000

Mantissa (A) = 1|00101010000001011111001

Exponent to Postnormalization = Exp (O) =160 + 131 -127 = 164 = 37 = 10100100

Step 2: Multiplication

Output sign = sign (A) XOR Sign (B) = ‘0’ XOR ‘0’ = ‘0’

Mantissa (A) = 1|00101010000001011111001

Mantissa (B) = 1|01001110111011100000000

Product (48 bits) = 01|100001011110100011010011010000101111110000000

Step 3: Postnormalization

Product[47:0]= 01|100001011110100011010011010000101111110000000

Exponent to Postnormalization = Exp (O) = 37

Carry =0=> no right shift product[47:0]; Exp (O) = 37

Product 2[47:0] = **01** |10000101111010001101001| **10** |10000101111110000000

01 – carry and hidden bits.

10 – guard and round bits.

Sticky = OR (Product 2 [20:0]) = 1

Roundup = guard and ((round or sticky) or Product 2(23) = 1

Product is rounded up assuming round-to-nearest mode.

Rounded Product = Product 2 [47:23] + 1 = 01|100001011110100011010 01 + 1

= 01|100001011110100011010 10

Discarded bits – Product 2 [23:0] - 10 |10000101111100000000

Carry bit =0 =>No right shift.

Exp (O) = 37 = 164 (without bias) = 10100100

Concatenate: Rounded Sum (excluding carry and hidden bits) with Sign and Exponent

0 | 10100100 |10000101111010001101010 = 0x (5242F46A)

3.6.3. Multiplication with Residual Register Examples

Example 1:

A = 0x (4171999A) = 01000001011100011001100110011010

B = 0x (40000011) = 01000000000000000000000000010001

Step 1: Prenormalization:

Sign (A) = 0; Exp (A) = 10000010 = 130 – 127 = 3; Mantissa (A) = 11000110011001100110 10;

Prepend mantissa (A) with a hidden bit. Hidden = 1 if Exp ≠ “00000000” that is number is not a denormalized number.

Mantissa (A) = 1|11100011001100110011010 -- 0x (F1999A)

Sign (B) = 0; Exp (B) = 10000000 = 128-127 = 1; Mantissa (B) = 10000010100011110101110;

Prepend mantissa (B) with a hidden bit.

Mantissa (B) = 1|00000000000000000010001 -- 0x (800011)

Exp (O) – exponent to the postnormalization = 130 + 128 -127 = 131 = 4

Step 2: Multiplication

Output sign = sign (A) XOR Sign (B) = '0' XOR '0' ='0'

Mantissa (A) = 1|11100011001100110011010 – 0x (F1999A)

Mantissa (B) = 1|00000000000000000010001 – 0x (800011)

Product (48 bits) = 01|111000110011001101110100001011001100111010

Step 3: Postnormalization

Product[47:0]= 01|111000110011001101110100001011001100111010

Exponent to Postnormalization = 4

Carry =0 => No right shift product[47:0]

Product 2 [47:0] = **01**| 11100011001100110111010 |**00**| 01011001100110011101O

01 – carry and hidden bits.

00 – guard and round bits.

Sticky = OR (Product 2 [20:0]) = 1

Roundup = guard and ((round or sticky) or Product 2(23) = 0

Based on rounding logic, product is rounded down.

Rounded product = product 2 [47:23] + 1 = 01| 11100011001100110111010

Lower 23 bits discarded go into the residual register - 00| 01011001100111010

Residual register mantissa = Mantissa (RR) = 00| 01011001100111010

Carry bit =0 =>No right shift. Nothing goes into residual register.

Mantissa (RR) = 000| 01011001100111010

Exponent (RR) = Output Exponent – 24 = 131-24 = 107 = 01101011

Complement (RR) = roundup =’0’

Sign (RR) = sign (O) XOR roundup = ‘0’

Concatenate: Rounded Sum (excluding carry and hidden bits) with Sign and Exponent

0 | 10000011 |11100011001100110111010 = 0x (41F199BA)

Outputs:

Output = 0 | 10000011 |11100011001100110111010 = 0x (41F199BA)

Mantissa (RR) = 00000101100110011001110 10; Exponent (RR) = 01101011; Sign (RR) = ‘0’;

MOVRR =1

Step 4: Normalization of Residual Register Value

Input = Product[47:0] = Mantissa (RR) & “000000000000000000000000”

= 00| 00010110011001100111010000000000000000000000000

Exponent to postnormalization = 107 = 01101011

Sign (RR) = ‘0’

Hidden bit = 0 => count zeros from left starting from hidden bit or Product[46] = 2

Shift left product 2 times.

Shifted product = Product 2 [46] =
00001|01100110011001110100000|00|00000000000000000

Exponent (RR) = 107 – 5 = 102

01 – carry and hidden bits.

00 – guard and round bits.

Sticky = OR (Product 2 [20:0]) = 0

Roundup = guard and ((round or sticky) or Product 2(23) = 0

Shifted product is rounded down based on the rounding logic.

Rounded product = 000101100111001100111010

Carry bit = 0 => no right shift.

Output = Sign (RR) | Exponent (RR)| Rounded Product (excluding carry and hidden bits)

Normalized Residual value = 0|01100110|00010110011001100111010 = 0x (330B333A)

Example 2:

A = 0x (501502F9) = 01010000000101010000001011111001

B = 0x (41A77700) = 01000001101001110111011100000000

Step 1: Prenormalization

Sign (A) = 0; Exp (A) = 10100000 = 160 – 127 = 33;

Mantissa (A) = 00101010000001011111001;

Prepend mantissa (A) with a hidden bit. Hidden = 1 if Exp ≠ “00000000” that is number is not a denormalized number.

Mantissa (A) = 1|00101010000001011111001

Sign (B) = 0; Exp (B) = 10000011 = 131 – 127 = 4;

Mantissa (B) = 01001110111011100000000;

Prepend mantissa (B) with a hidden bit.

Mantissa (B) = 1|01001110111011100000000

Exp (A) > Exp (B) = 1

Mantissa (B) = 1|01001110111011100000000

Mantissa (A) = 1|00101010000001011111001

Exponent to Postnormalization = Exp (O) = 160 + 131 -127 = 164 = 37(with bias) = 10100100

Step 2: Multiplication

Output sign = sign (A) XOR Sign (B) = '0' XOR '0' = '0'

Mantissa (A) = 1|00101010000001011111001

Mantissa (B) = 1|01001110111011100000000

Product (48 bits) = 01|100001011110100011010011010000101111100000000

Step 3: Postnormalization

Product [47:0]= 01|100001011110100011010011010000101111100000000

Exponent to Postnormalization = Exp (O) = 37

Carry =0=> no right shift product [47:0]; Exponent = Exp (O) = 37

Product 2 [47:0] = **01** |100001011110100011010 01| **10** |10000101111100000000

01 – carry and hidden bits.

10 – guard and round bits.

Sticky = OR (Product 2 [20:0]) = 1

Roundup = guard and ((round or sticky) or Product 2(23) = 1

Product is rounded up assuming round-to-nearest mode.

Rounded Product = Product 2 [47:23] + 1 = 01|10000101111010001101001 + 1

= 01|10000101111010001101010

Discarded bits – Product 2 [23:0] - 10 |10000101111111000000000 go into the residual register

Residual Register Mantissa = 10100001011111100000000

Carry bit =0 =>No right shift. So nothing is added into the residual register.

Concatenate: Rounded product (excluding hidden bit) with Sign and Exponent

Product output = 0 | 10100100 |10000101111010001101010 = 0x (5242F46A)

Mantissa (RR) = 00|10100001011111100000000

Exponent (RR) = Exp (O) – 24 = 164 – 24 = 140 (without bias) = 10001100

Sign (RR) = Sign (O) XOR roundup = ‘0’ XOR ‘1’ = ‘1’

Complement (RR) = roundup = ‘1’ => bits added in Mantissa (RR) are complemented.

Mantissa (RR) = 00 & (~ Mantissa (RR))

= 00 | 01011110100000011111111

Outputs:

Product output = 01010010010000101111010001101010 = 0x (5242F46A)

Mantissa (RR) = 00 | 01011110100000011111111

Exponent (RR) = 10001100; Sign (RR) = ‘1’

MOVRR = 1

Step 4: Normalization of Residual register value

Input = Product [47:0] = Mantissa (RR) & "00000000000000000000"

= 00 | 01011110100000011111111000000000000000000000000

Exponent to postnormalization = 140 = 10001100

Sign (RR) = '1'

Hidden bit = 0 => count zeros from left starting from hidden bit or Product[46] = 2

Shift left product 3 times.

Shifted product product2 [47:0] =
01|01111010000001111111100000000000000000000000|00

Exponent (RR) = 140 – 3 = 137

01 – carry and hidden bits.

00 – guard and round bits.

Sticky = OR (Product 2 [20:0]) = 0

Roundup = guard and ((round or sticky) or Product 2(23) = 0

Shifted product is rounded down based on the rounding logic.

Rounded product = 01|01111010000001111111100

Carry bit = 0 => no right shift.

Output = Sign (RR) | Exponent (RR)| Rounded Product (excluding carry and hidden bits)

Normalized Residual value = 1|10001001|01011110100000011111111 = 0x (C4AF40FF)

Chapter 4. Testing and Results

The Native-pair FPU adder and multiplier have been verified using 6,300 test cases. Both behavioral and post-route simulations were run using Modelsim for this purpose. A set of Gaussian distributed data was used for the purpose of thorough testing of both the adder and the multiplier. This Gaussian sequence was a sequence 6300 million randomized numbers with mean value of zero and variance value of 1. The sequence values were stored into a text file in IEEE 754 format. A VHDL test bench was written to read this data from this text file, generate results. These same test cases have been used to test both the adder and multiplier and the results so obtained were compared to the results by the software program. The VHDL test benches and the residual.cpp codes are given in the Appendix B

Appendix A describes in detail the place and route simulation output waveforms.

Chapter 5. Estimation of Hardware Cost and Performance

The Floating-point unit with residual register needed to perform Native-pair floating-point arithmetic has been implemented in VHDL. The floating-point adder and multiplier with residual register have been synthesized to evaluate the hardware complexity and speed of the design. Implementation costs of these designs are compared with those of 32-bit FPU and 64-bit FPU without the residual register hardware. Xilinx 9.1 ise tool is used to generate post-route synthesis reports and Modelsim is used to generate place and route simulation waveform. The three designs – 32-bit FPU with residual register, 32-bit FPU without residual register, 64-bit FPU without residual register are targeted to Xilinx Virtex 4 FPGA xc4vlx25 device. Also the individual stages of the floating-point unit adder and multiplier i.e., prenormalization unit, addition unit, multiplication unit and postnormalization unit are also synthesized in order to analyze the residual register hardware needed to support the Native-pair floating-point arithmetic in a native 32-bit FPU.

5.1. Adder Implementation

Table 6. Comparison of Implementation cost and delay for Adders

ADDER	Implementation cost		Minimum period	
	Slices	% Increase	(ns)	% Increase
32-bit FPU adder without residual register	1437	0.0	20.924	0.0
32-bit FPU adder with residual register	1674	16.5	24.971	19.34
64-bit FPU adder without residual register	2272	56.4	62.1	197.8

Table 6 above gives the number of slices used and the minimum period of the critical path obtained from the post-route synthesis reports of the floating-point adders. The Implementation cost column is divided into the absolute utilization and relative utilization. The absolute utilization gives the exact number of slices needed for each

design and the relative utilization indicates the percentage increase in number of slices for each design relative to 32-bit FPU adder without residual register. The second column gives the minimum period or the delay in the critical path of the three adders and their relative increases taking 32-bit FPU adder without residual register as the mean. The observations drawn from Table 6 are:

- The percentage increase in hardware from using residual register for 32-bit FPU adder instead of 32- FPU adder without residual register is 16.5% and the percentage increase jumps 56.4% when 64-bit FPU adder without residual register is used

- The minimum period increased by 4.67 ns for 32-bit FPU adder with residual register compared to the increase of 41 ns for 64-bit FPU adder hardware. The relative increase in the minimum period changes from 19.34% to 197.8%.

- Comparing only 32-bit FPU adder and 64-bit FPU adder, the relative hardware cost increases by a factor of 3.4 and minimum period increases by a factor of 10.

The hardware cost of 64-bit FPU adder is due to increase in the use of resources and increase in the size of pipeline. Table 7 shows the comparison of 32-bit FPU adders with and without residual register hardware. This rise in the hardware cost involves addition of no new logic but only increasing the size of all the combinational and sequential logic within the 32-bit FPU adder hardware. But the increase in the hardware cost of 32-bit FPU adder with residual register is due to addition of new logic which is needed in the residual value computation.

Table 7. Comparison of device utilization reports of Prenormalization unit for 32-bit FPU adder with and without residual register hardware

ADDER - Prenormalization	32-bit FPU adder without residual register	32-bit FPU adder with residual register
Number of Slices	498	492
Number of Slice Flip Flops	16	64
Number of 4 input LUTs	889	880
Minimum period	1.10 ns	1.648 ns

Table 7 shows the device utilization summary of the prenormalization unit for 32-bit FPU adder with and without the residual register. The overall hardware needed for adder with residual register increased due the addition of residual register and the storing it with the discarded bits after a right-shift is performed in prenormalization. This extra hardware shown in bold in Figure 14, adds sequential logic in the critical data path which becomes the cause for the delay and the increase in the minimum period.

Table 8. Comparison of device utilization reports of Postnormalization unit for 32-bit FPU adder with and without residual register hardware

ADDER - Postnormalization	32-bit FPU adder with residual register	32-bit FPU adder without residual register
Number of Slices	522	639
Number of 4 input LUTs	932	1130

Table 9 shows the extra hardware in the Postnormalization unit of the FPU adder with residual register. The extra hardware shown in Figure 15 is due to the appending of the discarded bits in to the residual register value that comes from the prenormalization unit, the computation of the sign, the complement flag and the exponent value, computing the 2's complement of the residual value based on the complement flag and storing all this into a residual register. As there is no delay in the critical datapath as the extra hardware does not involve any sequential logic in the datapath.

5.2. Multiplier Implementation

Table 10 shows the results of the place-route synthesis reports. The increase in number of slices for 32-bit FPU multiplier with residual register is 330 and it is 4497 for 64-bit FPU multiplier. The minimum period increases by 11.8% and 61.7% respectively for 32-bit FPU multiplier with residual register hardware and 64-bit FPU multiplier.

Table 9. Comparison of Implementation cost and delay of Multipliers

MULTIPLIER	Implementation cost		Minimum period	
	Slices	% Increase	(ns)	% Increase
32-bit FPU multiplier without residual register	2703	0.0	34.754	0.0
32-bit FPU multiplier with residual register	3033	12.2	38.875	11.8
64-bit FPU multiplier without residual register	7200	166.3	55.7	61.74

The inferences that can be drawn from table 10:

- The extra hardware needed for 32-bit FPU multiplier with residual register increases by 12.2% and for 64-bit FPU multiplier the hardware cost increases by 166%.
- The minimum period increases by 4 ns for 32-bit FPU multiplier with residual register hardware and 21 ns for 64-bit FPU multiplier.
- Hardware cost (64-bit FPU multiplier) = 13.6 × hardware cost (32-bit FPU multiplier with residual register).
- Minimum period increase (64-bit FPU multiplier) = 5.23 × minimum period increase (32-bit FPU multiplier with residual register).

Similar to the adders, the hardware increase in 64-bit multiplier is primarily due to increase in the size of the resources and the pipeline. From Table 9 it can also be observed that the increase in the hardware when residual register is used for multiplier is less than the increase for adder with residual register. Floating-point multiplication does not have any shifting or discarding of bits in prenormalization but floating-point addition involves shifting of mantissa in the prenormalization unit. The residual register hardware and the related logic are present only in the postnormalization unit for a FPU multiplier.

Table 10. Comparison of device utilization reports of Postnormalization unit for 32-bit FPU multiplier with and without residual register hardware

Multiplier - Postnormalization	32-bit FPU adder with residual register	32-bit FPU adder without residual register
Number of Slices	1817	1807
Number of Slice flip flops	112	146
Number of 4 input LUTs	3322	3311
Minimum period	8.332 ns	7.637 ns

Table 12 shows the device utilization of postnormalization unit in FPU multiplier with residual register and FPU multiplier without residual register. All the extra logic resulting in extra hardware is due to setting of the residual mantissa before and after rounding, computing the residual register exponent, the sign and complement flag and complementing the residual register value if the complement flag is set. The delay in the critical path increases by 0.7 ns due to this extra hardware.

Conclusion

Most processors in video game consoles and graphics hardware widely use 32-bit or single precision floating-point hardware which is available at low cost. To harness this hardware for scientific computing, the intermediate results in these processors require precision higher than 32-bits. Usage of double precision or 64-bit floating-point arithmetic is not justifiable for this purpose because the scientific computing market is too small to justify the added expense. Using 32-bit Native-pair arithmetic increases the accuracy of these applications close to that offered when double precision arithmetic is used but at a fraction of the cost of 64-bit floating point.

The FPU unit [19] used for this thesis has been debugged extensively before being used to implement the residual register hardware needed for adder and multiplier. The signals between modules have been routed correctly to enable pipelined operation. The input signals were carried through various stages in the pipeline or till where ever they were needed. The pipelines were balanced to attain maximum operable frequency. Signals present in the last stage and that required wider operands to be enabled have been computed in the earlier stage of the pipeline. There by the requirement for wider operands to be routed through the pipeline was negated and the width of the pipeline was reduced. The debugged FPU gave addition outputs every 3 clock cycles and multiplication output every 5 clock cycles. The residual register hardware was added to the FPU adder and FPU multiplier and its proper functioning has been implemented. Both the FPU adder and the FPU multiplier have been thoroughly tested by performing post-route simulations and also performing test-bench analysis using the synthetic data generated by the test code. The synthesis reports after the placement and routing have been obtained and a detailed analysis has been show in Chapter 5.

As can be seen from the synthesis results in Table 6, the increase in the hardware cost of due to residual register hardware is 15.4% for adders and 12.2% for multipliers. The increase in the hardware for 64-bit floating-point hardware is 55% for adders and 166% for multipliers. When comparing just 64-bit floating-point hardware and the 32-bit residual register hardware, there is a cost increases by a factor of 3.6 for adders and 13.6

for multipliers. The minimum period comparison as obtained from the Table 6 shows that the period increases for adder with residual register by 37% where as in multiplier with residual register the increase is just 11.8%. In comparison, the 64-bit adder has a decrease in performance by 226% and 64-bit multiplier has a decrease by 61%.

These results prove that with a minimal increase in hardware cost and a moderate slow down in performance, the native-pair arithmetic can be used to increase the accuracy of floating-point computations rather than going for the high cost double precision hardware. The residual register arithmetic unit performance can be enhanced greatly through the use of speculation. Using the native-pair hardware only when the speculation software detects loss of information above a certain limit will certainly result in floating-point arithmetic with higher precision, better accuracy and improved performance [31].

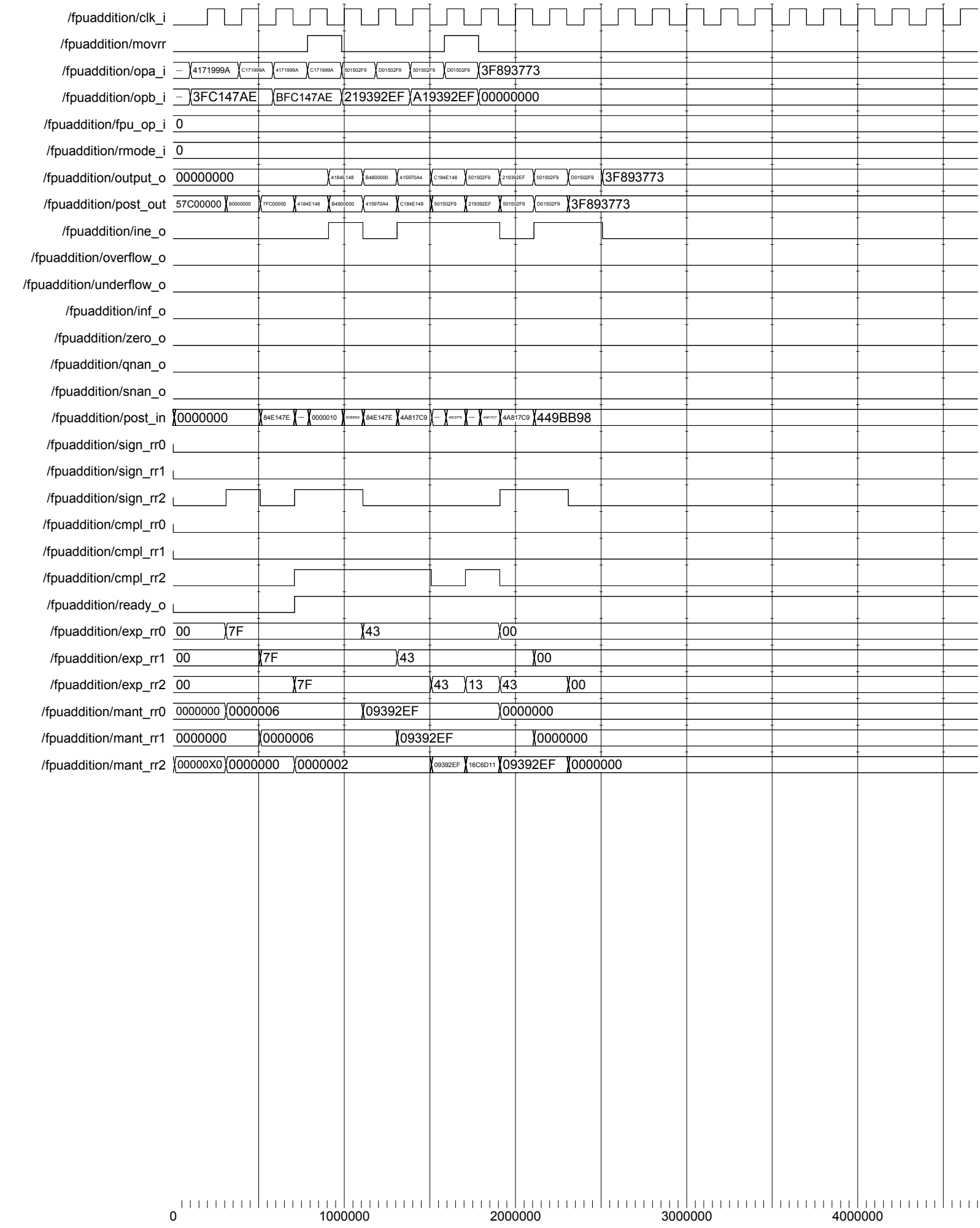
Appendix A

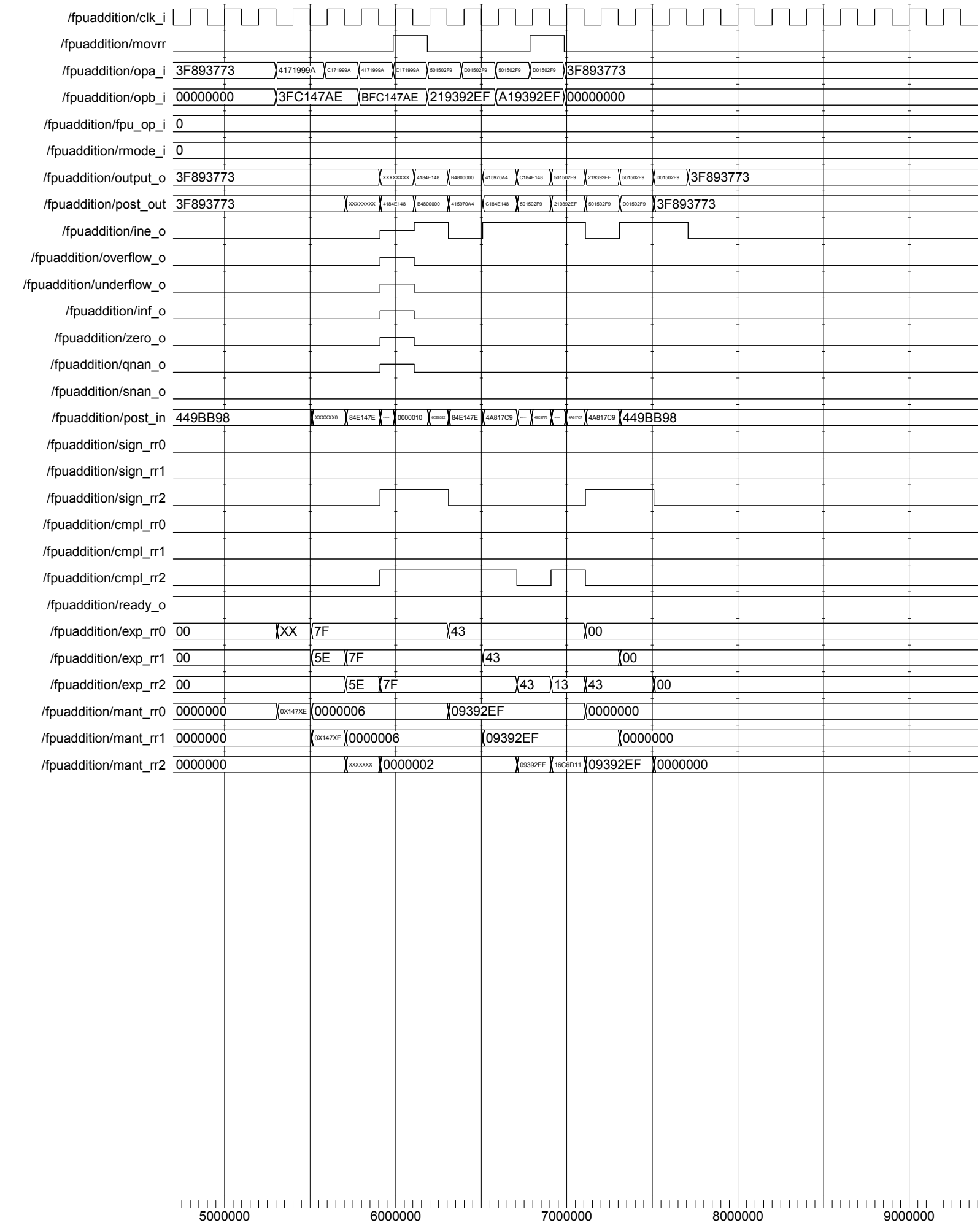
Post-route simulations

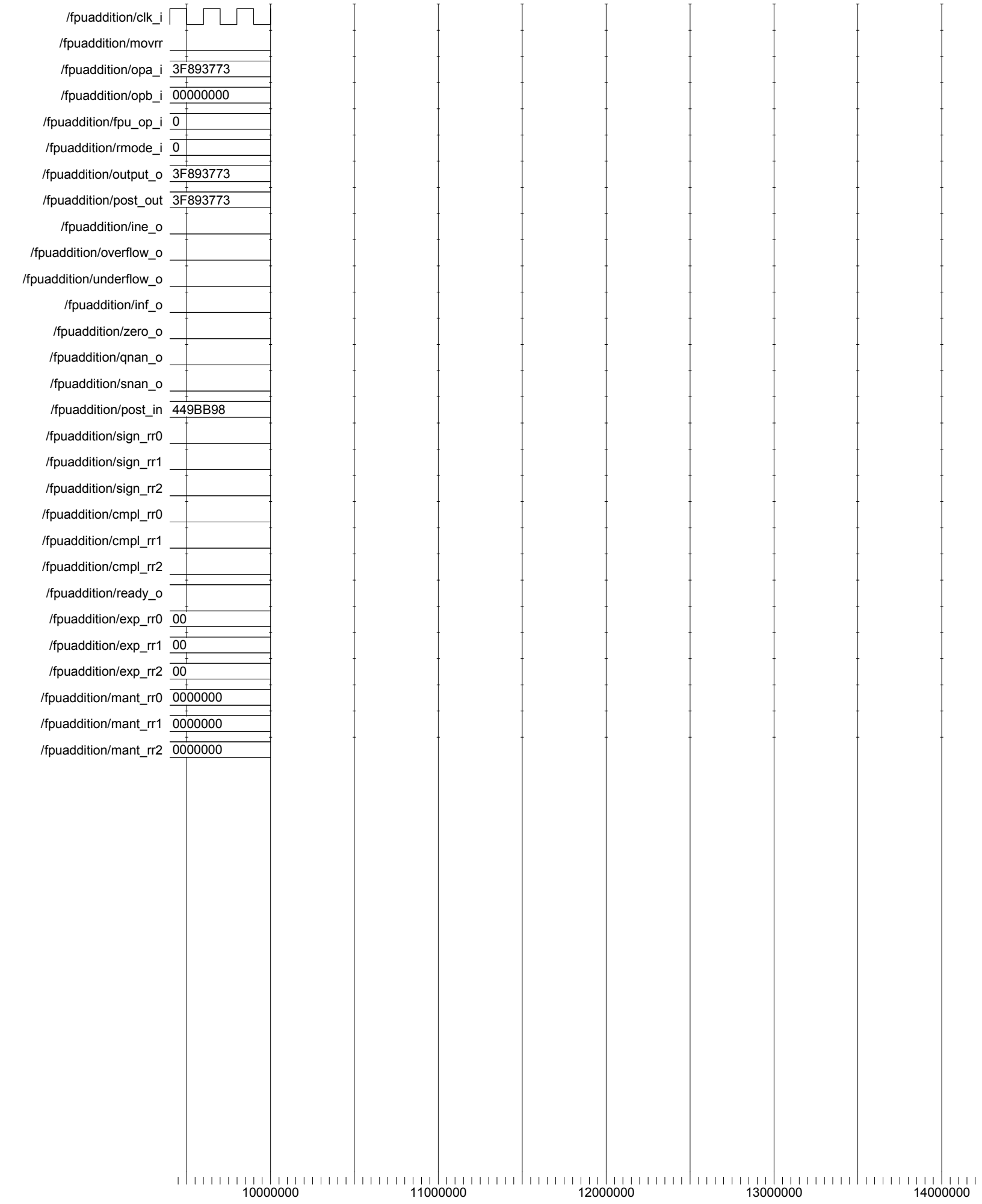
The simulation reports for the addition with residual register are presented in the next two pages. The next page shows the post-route simulation for 8 pairs of operands inputs were given in continuous clock cycles. The 1st signal is the clock input; the 2nd signal is the MOVRR input; 3rd and 4th signals are the operands A and B represented as opa_i and opb_i; 5th signal is the opcode 000-addition, 001-subtraction; the 6th signal is the rounding mode; the 7th signal is the output of the FPU, it could be the result or the normalized residual value depending on the MOVRR signal; 8th and 16th signals are the output of the postnormalization and the input to the postnormalization. These signals have been used to check if the right residual value was going into the postnormalization unit when MOVRR goes high; the 17th signal is the final sign of the residual register value from the postnormalization unit; the 20th signal is the complement flag output of the residual register in the postnormalization unit and it is used to keep track of when the residual value is being complemented; signals 24,25 and 26 are the exponent values of residual registers in the prenormalization unit, addition/subtraction unit and postnormalization unit. But as the final exponent is set only in the postnormalization unit only the 26th signal can be considered important; the 27th signal, 28th signal and 29th signal are residual register mantissa outputs from prenormalization unit, addition/subtraction unit and postnormalization unit. Since the mantissa is set in both prenormalization and postnormalization, signals 27 and 29 are important. Apart from these, signals 9 to 15 give the inexact, overflow and the exception outputs. 17th signal is the post_in signal and it is used to see if the residual register mantissa mant_rr2, exponent exp_rr2 and sign_rr2 obtained in the previous clock cycle is sent as input into the postnormalization when the MOVRR signal is one. 23rd signal is the ready signal, valid postnormalization unit outputs are sent to the FPU output only after this signal goes high. The FPU addition takes place in 3 clock cycles, one clock cycle each for prenormalization, addition and postnormalization.

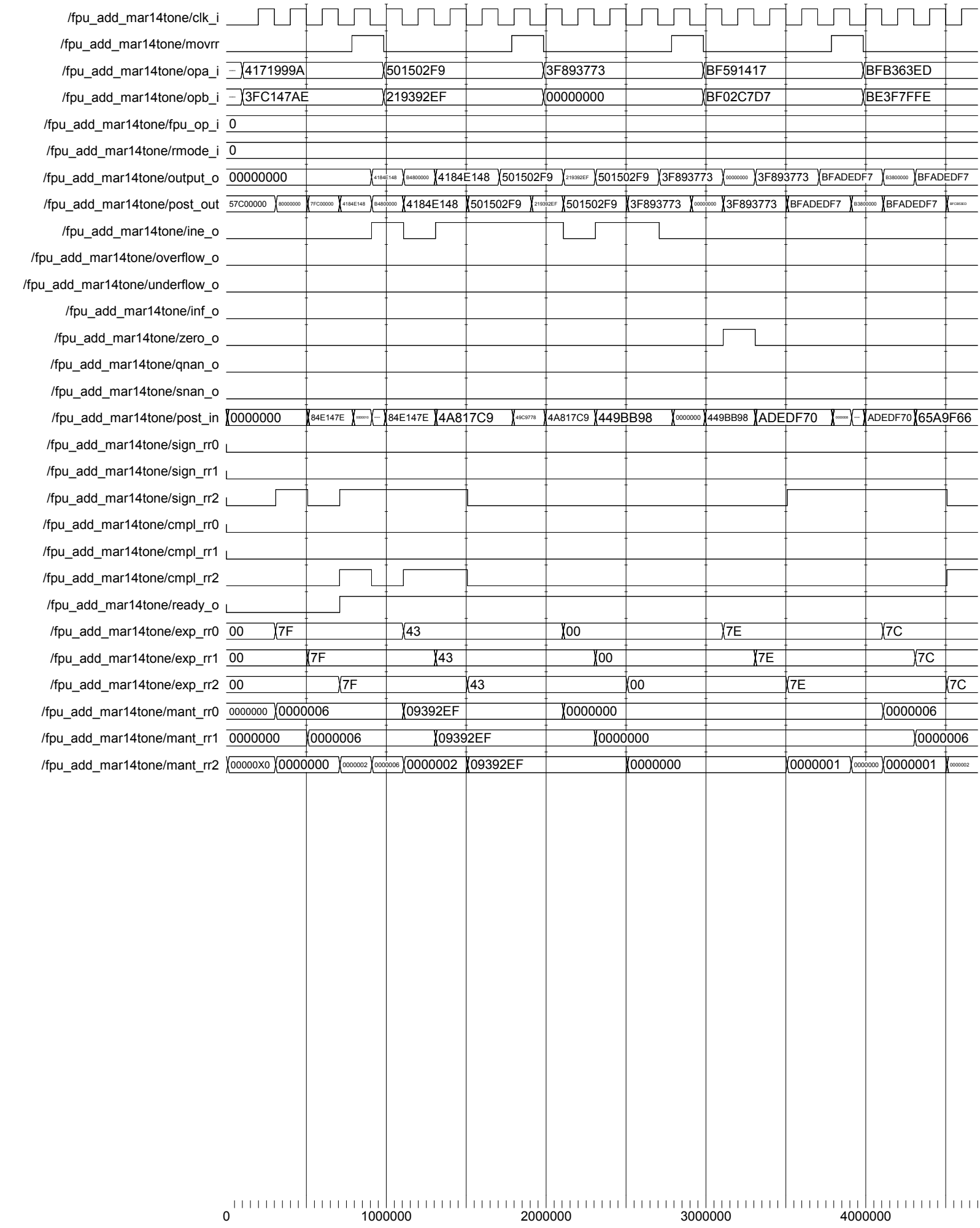
In the wave form shown in the next page, signal 3 indicates the operand A - opa_i; it is 4171999A in the 1st clock cycle, C17199A in the 2nd, 4171999A in the 3rd clock cycle, C17199A in the 4th, 501502F9 in the 5th and 7th, D01502F9 in the 6th and 8th and 3F893773 in the 9th clock cycle. 4th signal is operand B – opb_i which takes value 3FC147AE in 1st and 2nd clock cycles, BFC147AE in 3rd and 4th clock cycles, 219392EF in the 5th and 6th clock cycles, A19392EF in the 7th and 8th clock cycles and 00000000 in the 9th clock cycle. The output for the inputs in the 1st clock cycle that is opa_i = 4171999A and opb_i = 3FC147AE comes in the 4th clock cycle with the falling edge of the clock and its value is shown by the 7th signal output_o = 4184E148. The outputs for the other inputs come along the consequent clock cycles. MOVRR signal goes high at the end of the 3rd clock cycle, at this time the post_out = 4184E148 and post_in becomes the residual register mantissa value obtained in the previous clock cycle. The normalized residual value to be stored in the architectural register B48000000 is obtained in the 5th clock cycle with the falling edge of the clock. Similarly MOVRR again goes high in the 8th clock cycle to give the normalized residual register value 219392EF as output in the 9th clock cycle.

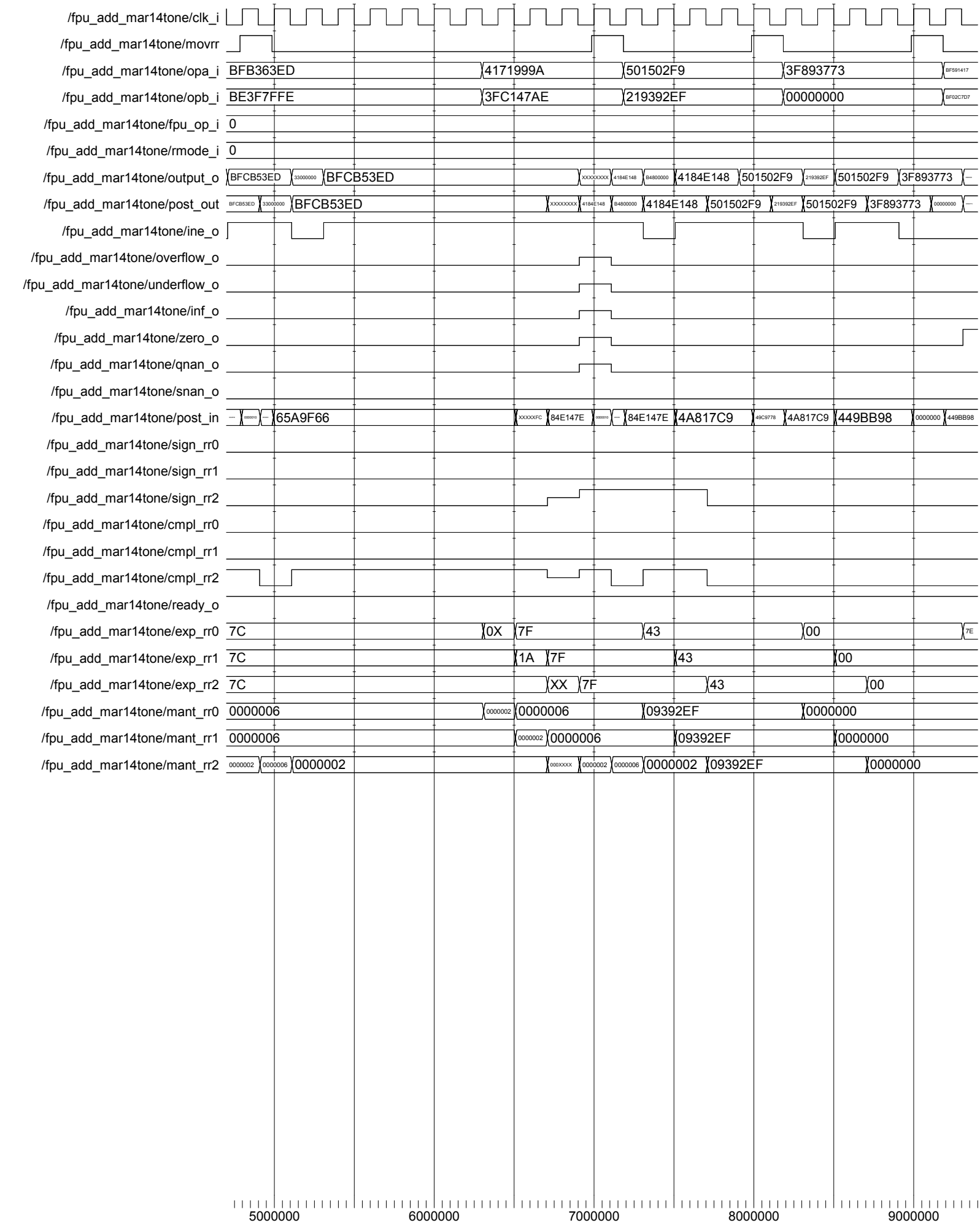
The second wave form has been run to check if the all the 8 input vectors are giving the correct values of output and residual register values. As can be seen in the wave form MOVRR signal is made to go high after every 3 clock cycles for this purpose.


/fpuaddition/clk_i
/fpuaddition/movrr
/fpuaddition/opa_i
4171999A
C171999A
4171999A
C171999A
501502F9
D01502F9
501502F9
D01502F9
3F893773
/fpuaddition/opb_i
3FC147AE
BFC147AE
219392EF
A19392EF
00000000
/fpuaddition/fpu_op_i
0
/fpuaddition/rmode_i
0
/fpuaddition/output_o
00000000
4184E148
B4800000
415970A4
C184E148
501502F9
219392EF
501502F9
D01502F9
3F893773
/fpuaddition/post_out
57C00000
80000000
7FC00000
4184E148
B4800000
415970A4
C184E148
501502F9
219392EF
501502F9
D01502F9
3F893773
/fpuaddition/ine_o
/fpuaddition/overflow_o
/fpuaddition/underflow_o
/fpuaddition/inf_o
/fpuaddition/zero_o
/fpuaddition/qnan_o
/fpuaddition/snan_o
/fpuaddition/post_in
0000000
84E147E
0000010
84E147E
4A817C9
4A817C9
449BB98
/fpuaddition/sign_rr0
/fpuaddition/sign_rr1
/fpuaddition/sign_rr2
/fpuaddition/cmpl_rr0
/fpuaddition/cmpl_rr1
/fpuaddition/cmpl_rr2
/fpuaddition/ready_o
/fpuaddition/exp_rr0
00
7F
43
00
/fpuaddition/exp_rr1
00
7F
43
00
/fpuaddition/exp_rr2
00
7F
43
13
43
00
/fpuaddition/mant_rr0
0000000
0000006
09392EF
0000000
/fpuaddition/mant_rr1
0000000
0000006
09392EF
0000000
/fpuaddition/mant_rr2
00000X0
0000000
0000002
09392EF
16C6D11
09392EF
0000000
0
1000000
2000000
3000000
4000000


/fpuaddition/clk_i
/fpuaddition/movrr
/fpuaddition/opa_i
3F893773
4171999A
C171999A
4171999A
C171999A
501502F9
D01502F9
501502F9
D01502F9
3F893773
/fpuaddition/opb_i
00000000
3FC147AE
BFC147AE
219392EF
A19392EF
00000000
/fpuaddition/fpu_op_i
0
/fpuaddition/rmode_i
0
/fpuaddition/output_o
3F893773
XXXXXXXX
4184E148
B4800000
415970A4
C184E148
501502F9
219392EF
501502F9
D01502F9
3F893773
/fpuaddition/post_out
3F893773
XXXXXXXX
4184E148
B4800000
415970A4
C184E148
501502F9
219392EF
501502F9
D01502F9
3F893773
/fpuaddition/ine_o
/fpuaddition/overflow_o
/fpuaddition/underflow_o
/fpuaddition/inf_o
/fpuaddition/zero_o
/fpuaddition/qnan_o
/fpuaddition/snan_o
/fpuaddition/post_in
449BB98
XXXXXX0
84E147E
—
0000010
4C38532
84E147E
4A817C9
—
49C9F78
—
4A817C7
4A817C9
449BB98
/fpuaddition/sign_rr0
/fpuaddition/sign_rr1
/fpuaddition/sign_rr2
/fpuaddition/cmpl_rr0
/fpuaddition/cmpl_rr1
/fpuaddition/cmpl_rr2
/fpuaddition/ready_o
/fpuaddition/exp_rr0
00
XX
7F
43
00
/fpuaddition/exp_rr1
00
5E
7F
43
00
/fpuaddition/exp_rr2
00
5E
7F
43
13
43
00
/fpuaddition/mant_rr0
0000000
0X147XE
0000006
09392EF
0000000
/fpuaddition/mant_rr1
0000000
0X147XE
0000006
09392EF
0000000
/fpuaddition/mant_rr2
0000000
XXXXXXX
0000002
09392EF
16C6D11
09392EF
0000000
5000000
6000000
7000000
8000000
9000000


/fpuaddition/clk_i
/fpuaddition/movrr
/fpuaddition/opa_i 3F893773
/fpuaddition/opb_i 00000000
/fpuaddition/fpu_op_i 0
/fpuaddition/rmode_i 0
/fpuaddition/output_o 3F893773
/fpuaddition/post_out 3F893773
/fpuaddition/ine_o
/fpuaddition/overflow_o
/fpuaddition/underflow_o
/fpuaddition/inf_o
/fpuaddition/zero_o
/fpuaddition/qnan_o
/fpuaddition/snan_o
/fpuaddition/post_in 449BB98
/fpuaddition/sign_rr0
/fpuaddition/sign_rr1
/fpuaddition/sign_rr2
/fpuaddition/cmpl_rr0
/fpuaddition/cmpl_rr1
/fpuaddition/cmpl_rr2
/fpuaddition/ready_o
/fpuaddition/exp_rr0 00
/fpuaddition/exp_rr1 00
/fpuaddition/exp_rr2 00
/fpuaddition/mant_rr0 0000000
/fpuaddition/mant_rr1 0000000
/fpuaddition/mant_rr2 0000000
10000000
11000000
12000000
13000000
14000000

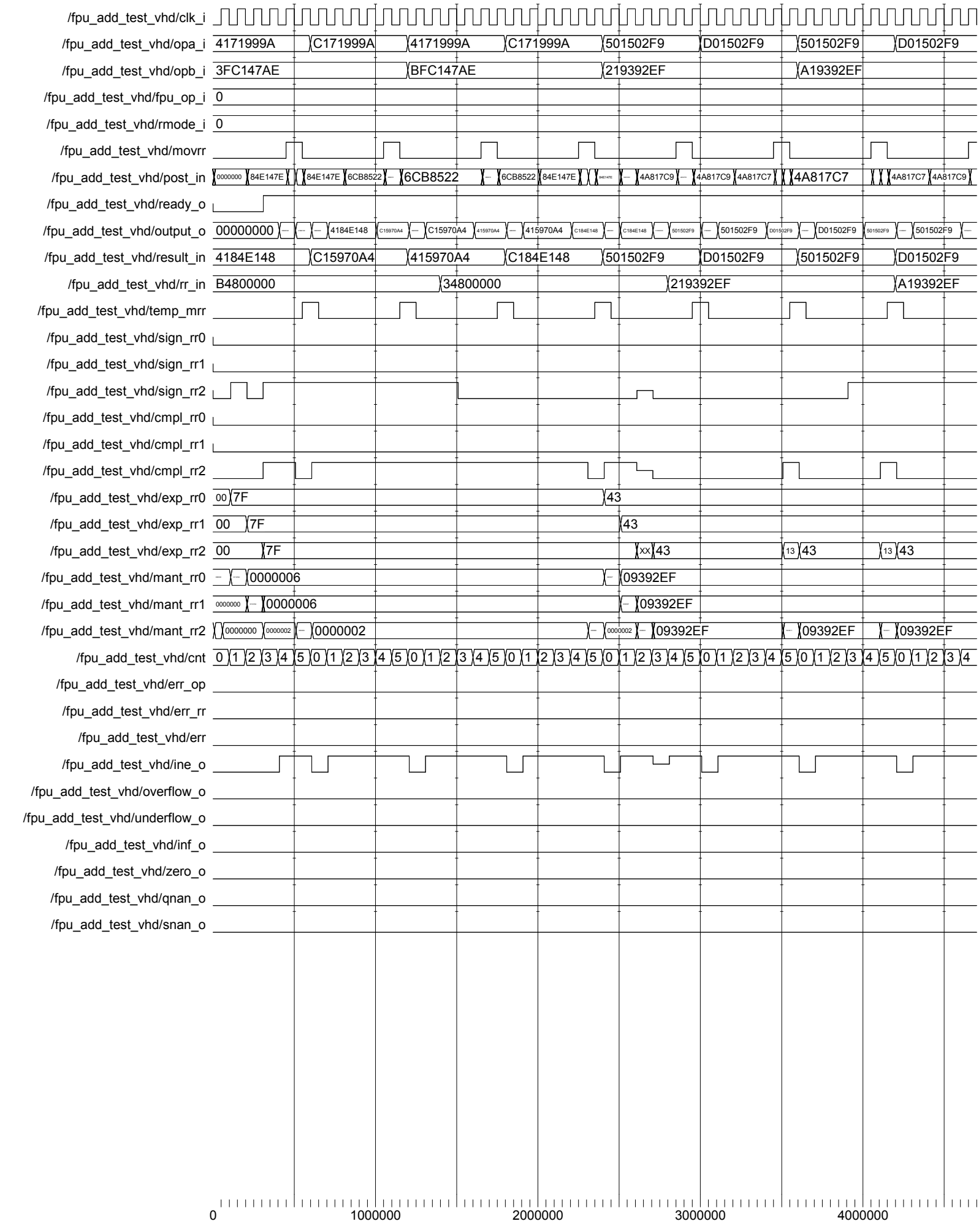

/fpu_add_mar14tone/clk_i
/fpu_add_mar14tone/movrr
/fpu_add_mar14tone/opa_i
4171999A
501502F9
3F893773
BF591417
BFB363ED
/fpu_add_mar14tone/opb_i
3FC147AE
219392EF
00000000
BF02C7D7
BE3F7FFE
/fpu_add_mar14tone/fpu_op_i
0
/fpu_add_mar14tone/rmode_i
0
/fpu_add_mar14tone/output_o
00000000
4184E148
501502F9
3F893773
BFADEDF7
/fpu_add_mar14tone/post_out
57C00000
4184E148
501502F9
3F893773
BFADEDF7
/fpu_add_mar14tone/ine_o
/fpu_add_mar14tone/overflow_o
/fpu_add_mar14tone/underflow_o
/fpu_add_mar14tone/inf_o
/fpu_add_mar14tone/zero_o
/fpu_add_mar14tone/qnan_o
/fpu_add_mar14tone/snan_o
/fpu_add_mar14tone/post_in
0000000
84E147E
4A817C9
449BB98
ADEDF70
65A9F66
/fpu_add_mar14tone/sign_rr0
/fpu_add_mar14tone/sign_rr1
/fpu_add_mar14tone/sign_rr2
/fpu_add_mar14tone/cmpl_rr0
/fpu_add_mar14tone/cmpl_rr1
/fpu_add_mar14tone/cmpl_rr2
/fpu_add_mar14tone/ready_o
/fpu_add_mar14tone/exp_rr0
00
7F
43
7E
7C
/fpu_add_mar14tone/exp_rr1
/fpu_add_mar14tone/exp_rr2
/fpu_add_mar14tone/mant_rr0
0000006
09392EF
/fpu_add_mar14tone/mant_rr1
/fpu_add_mar14tone/mant_rr2
00000X0
0000002
0000001
0
1000000
2000000
3000000
4000000


/fpu_add_mar14tone/clk_i
/fpu_add_mar14tone/movrr
/fpu_add_mar14tone/opa_i
/fpu_add_mar14tone/opb_i
/fpu_add_mar14tone/fpu_op_i
/fpu_add_mar14tone/rmode_i
/fpu_add_mar14tone/output_o
/fpu_add_mar14tone/post_out
/fpu_add_mar14tone/ine_o
/fpu_add_mar14tone/overflow_o
/fpu_add_mar14tone/underflow_o
/fpu_add_mar14tone/inf_o
/fpu_add_mar14tone/zero_o
/fpu_add_mar14tone/qnan_o
/fpu_add_mar14tone/snan_o
/fpu_add_mar14tone/post_in
/fpu_add_mar14tone/sign_rr0
/fpu_add_mar14tone/sign_rr1
/fpu_add_mar14tone/sign_rr2
/fpu_add_mar14tone/cmpl_rr0
/fpu_add_mar14tone/cmpl_rr1
/fpu_add_mar14tone/cmpl_rr2
/fpu_add_mar14tone/ready_o
/fpu_add_mar14tone/exp_rr0
/fpu_add_mar14tone/exp_rr1
/fpu_add_mar14tone/exp_rr2
/fpu_add_mar14tone/mant_rr0
/fpu_add_mar14tone/mant_rr1
/fpu_add_mar14tone/mant_rr2
BFB363ED
4171999A
501502F9
3F893773
BE3F7FFE
3FC147AE
219392EF
00000000
0
BFCB53ED
4184E148
B4800000
501502F9
65A9F66
84E147E
4A817C9
449BB98
0000000
7C
0X
7F
43
00
1A
XX
0000006
0000002
09392EF
5000000
6000000
7000000
8000000
9000000

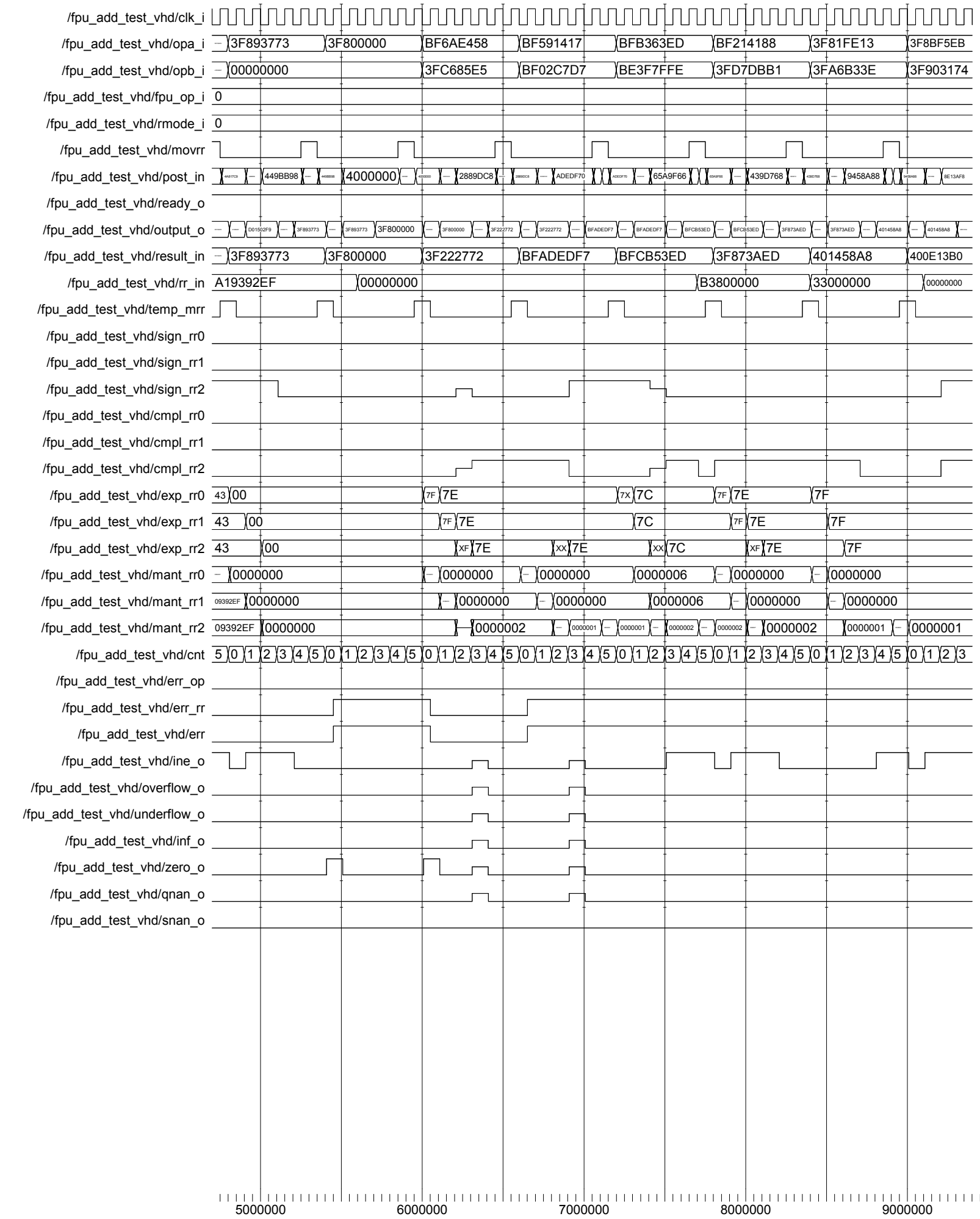
Entity:fpu_add_mar14tone Architecture:testbench_arch Date: Wed Mar 19 1:28:31 AM Eastern Daylight Time 2008 Row: 3 Page: 3

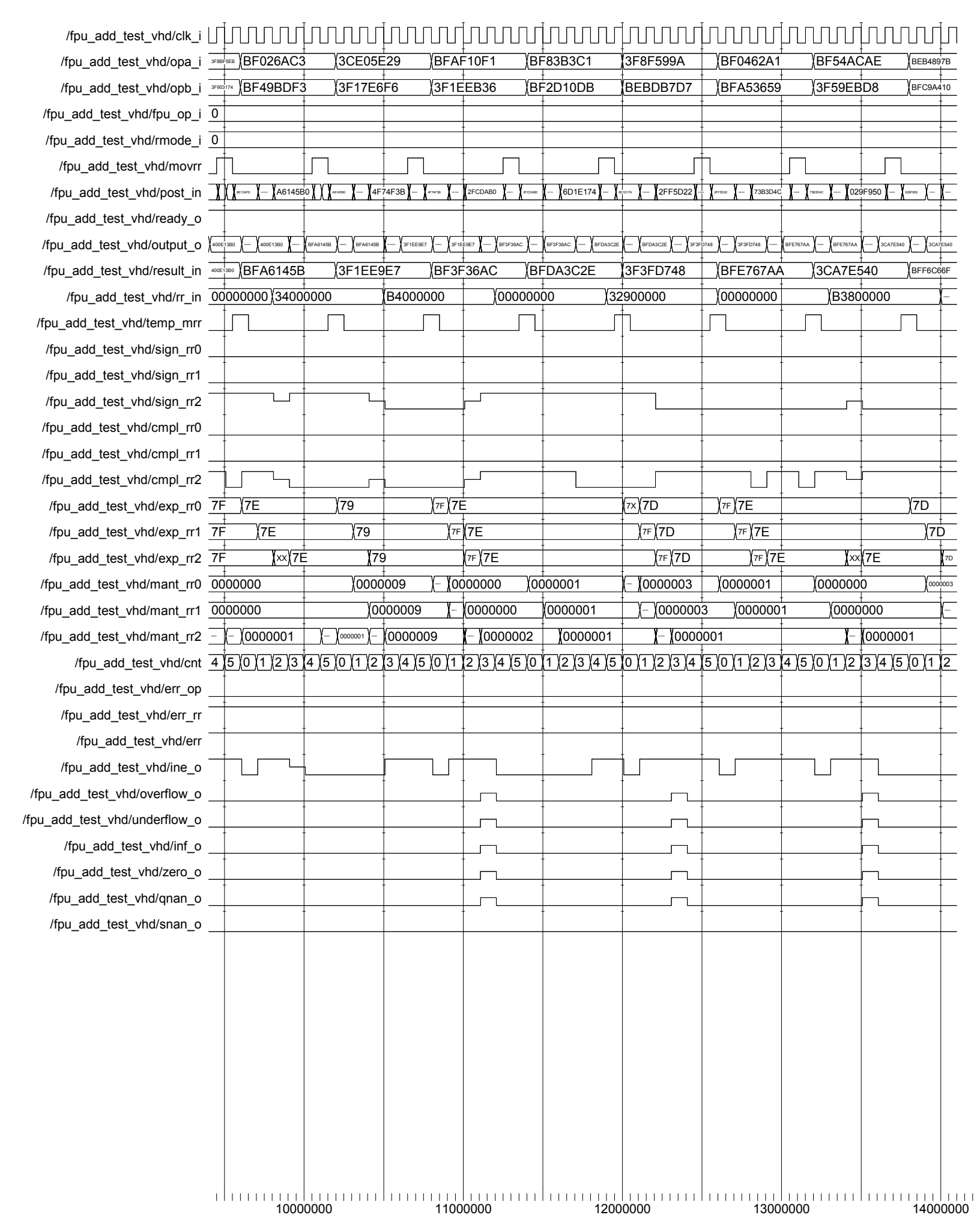
The wave form shown in the next page is the wave form of the test bench file that is used to test the FPU adder with the synthetic data. The MOVRR signal is made to periodically go high, one clock cycle before the addition output appears. The inputs from the text file are given periodically to the operands, and the outputs from the text file are read with the rising and falling edge of temp_mrr signal. This signal has been generated so that the outputs from the MUT - module under test and outputs from the text file match i.e., they correspond to the same operands.

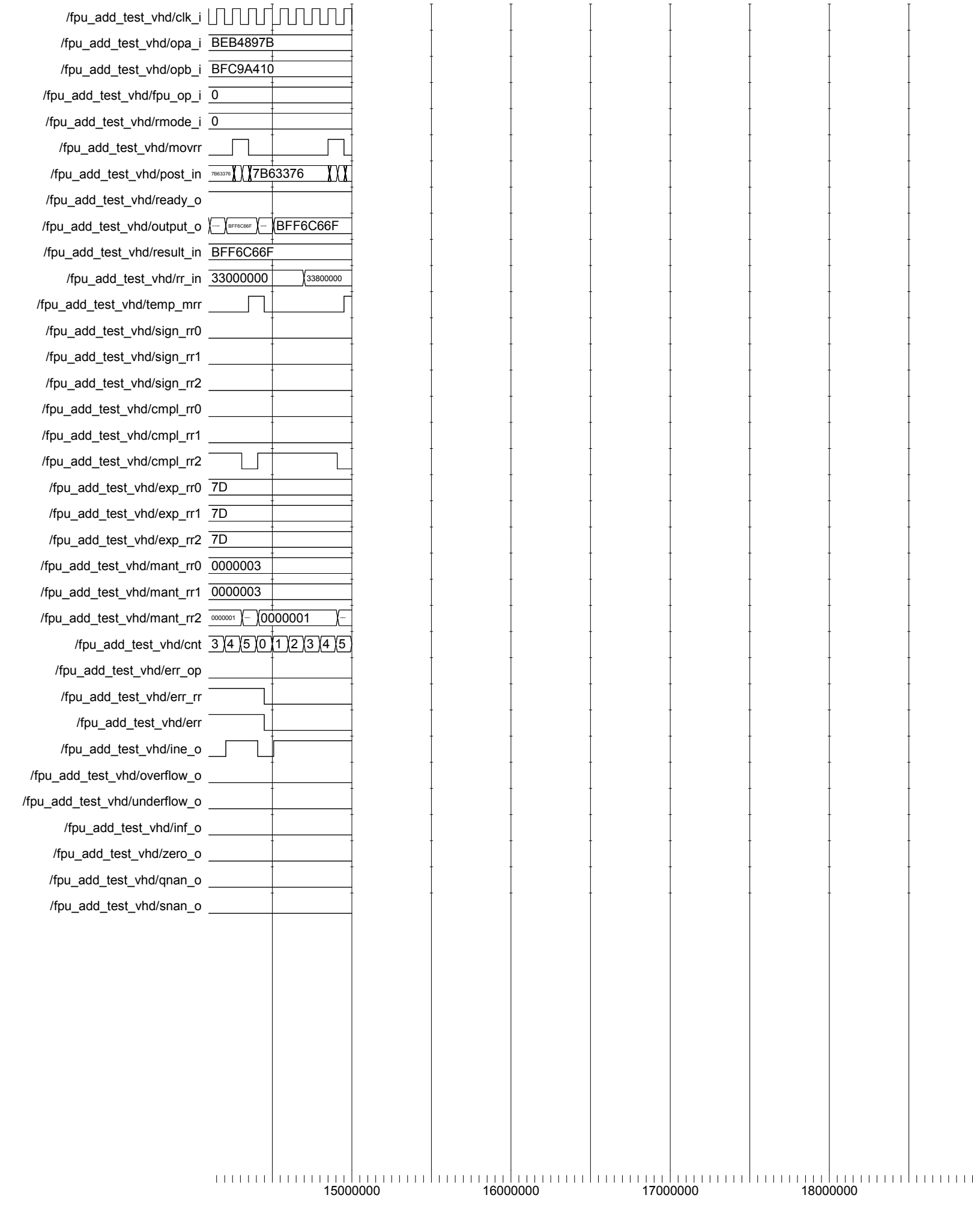
The 1st signal is the clock input;; 2nd and 3rd signals are the operands A and B represented as opa_i and opb_i; the 6th signal is the MOVRR input; 4th signal is the opcode 000-addition, 001-subtraction; 5th signal is the rounding mode; 9th signal is the output of the FPU, it could be the result or the normalized residual value depending on the MOVRR signal; 10th and 11th signals are the output and the residual register values read from the test data file. 7th signal is the input to the postnormalization. 26th, 27th and 28th signals are the error signals generated after comparing the outputs of MUT – module under test and the output values read from the test data file.



Entity:fpu_add_test_vhd Architecture:behavior Date: Wed Mar 19 2:11:58 AM Eastern Daylight Time 2008 Row: 1 Page: 1


/fpu_add_test_vhd/clk_i
/fpu_add_test_vhd/opa_i
/fpu_add_test_vhd/opb_i
/fpu_add_test_vhd/fpu_op_i
/fpu_add_test_vhd/rmode_i
/fpu_add_test_vhd/movrr
/fpu_add_test_vhd/post_in
/fpu_add_test_vhd/ready_o
/fpu_add_test_vhd/output_o
/fpu_add_test_vhd/result_in
/fpu_add_test_vhd/rr_in
/fpu_add_test_vhd/temp_mrr
/fpu_add_test_vhd/sign_rr0
/fpu_add_test_vhd/sign_rr1
/fpu_add_test_vhd/sign_rr2
/fpu_add_test_vhd/cmpl_rr0
/fpu_add_test_vhd/cmpl_rr1
/fpu_add_test_vhd/cmpl_rr2
/fpu_add_test_vhd/exp_rr0
/fpu_add_test_vhd/exp_rr1
/fpu_add_test_vhd/exp_rr2
/fpu_add_test_vhd/mant_rr0
/fpu_add_test_vhd/mant_rr1
/fpu_add_test_vhd/mant_rr2
/fpu_add_test_vhd/cnt
/fpu_add_test_vhd/err_op
/fpu_add_test_vhd/err_rr
/fpu_add_test_vhd/err
/fpu_add_test_vhd/ine_o
/fpu_add_test_vhd/overflow_o
/fpu_add_test_vhd/underflow_o
/fpu_add_test_vhd/inf_o
/fpu_add_test_vhd/zero_o
/fpu_add_test_vhd/qnan_o
/fpu_add_test_vhd/snan_o
3F893773
3F800000
BF6AE458
BF591417
BFB363ED
BF214188
3F81FE13
3F8BF5EB
00000000
3FC685E5
BF02C7D7
BE3F7FFE
3FD7DBB1
3FA6B33E
3F903174
449BB98
4000000
2889DC8
ADEDF70
65A9F66
439D768
9458A88
BE13AF8
D01502F9
3F222772
BFADEDF7
BFCB53ED
3F873AED
401458A8
400E13B0
A19392EF
B3800000
33000000
00000000
09392EF
0000000
0000002
0000001
0000006
5000000
6000000
7000000
8000000
9000000


/fpu_add_test_vhd/clk_i
/fpu_add_test_vhd/opa_i
/fpu_add_test_vhd/opb_i
/fpu_add_test_vhd/fpu_op_i
/fpu_add_test_vhd/rmode_i
/fpu_add_test_vhd/movrr
/fpu_add_test_vhd/post_in
/fpu_add_test_vhd/ready_o
/fpu_add_test_vhd/output_o
/fpu_add_test_vhd/result_in
/fpu_add_test_vhd/rr_in
/fpu_add_test_vhd/temp_mrr
/fpu_add_test_vhd/sign_rr0
/fpu_add_test_vhd/sign_rr1
/fpu_add_test_vhd/sign_rr2
/fpu_add_test_vhd/cmpl_rr0
/fpu_add_test_vhd/cmpl_rr1
/fpu_add_test_vhd/cmpl_rr2
/fpu_add_test_vhd/exp_rr0
/fpu_add_test_vhd/exp_rr1
/fpu_add_test_vhd/exp_rr2
/fpu_add_test_vhd/mant_rr0
/fpu_add_test_vhd/mant_rr1
/fpu_add_test_vhd/mant_rr2
/fpu_add_test_vhd/cnt
/fpu_add_test_vhd/err_op
/fpu_add_test_vhd/err_rr
/fpu_add_test_vhd/err
/fpu_add_test_vhd/ine_o
/fpu_add_test_vhd/overflow_o
/fpu_add_test_vhd/underflow_o
/fpu_add_test_vhd/inf_o
/fpu_add_test_vhd/zero_o
/fpu_add_test_vhd/qnan_o
/fpu_add_test_vhd/snan_o
BF026AC3
3CE05E29
BFAF10F1
BF83B3C1
3F8F599A
BF0462A1
BF54ACAE
BEB4897B
BF49BDF3
3F17E6F6
3F1EEB36
BF2D10DB
BEBDB7D7
BFA53659
3F59EBD8
BFC9A410
A6145B0
4F74F3B
2FCDAB0
6D1E174
2FF5D22
73B3D4C
029F950
BFA6145B
3F1EE9E7
BF3F36AC
BFDA3C2E
3F3FD748
BFE767AA
3CA7E540
BFF6C66F
00000000
34000000
B4000000
00000000
32900000
00000000
B3800000
10000000
11000000
12000000
13000000
14000000



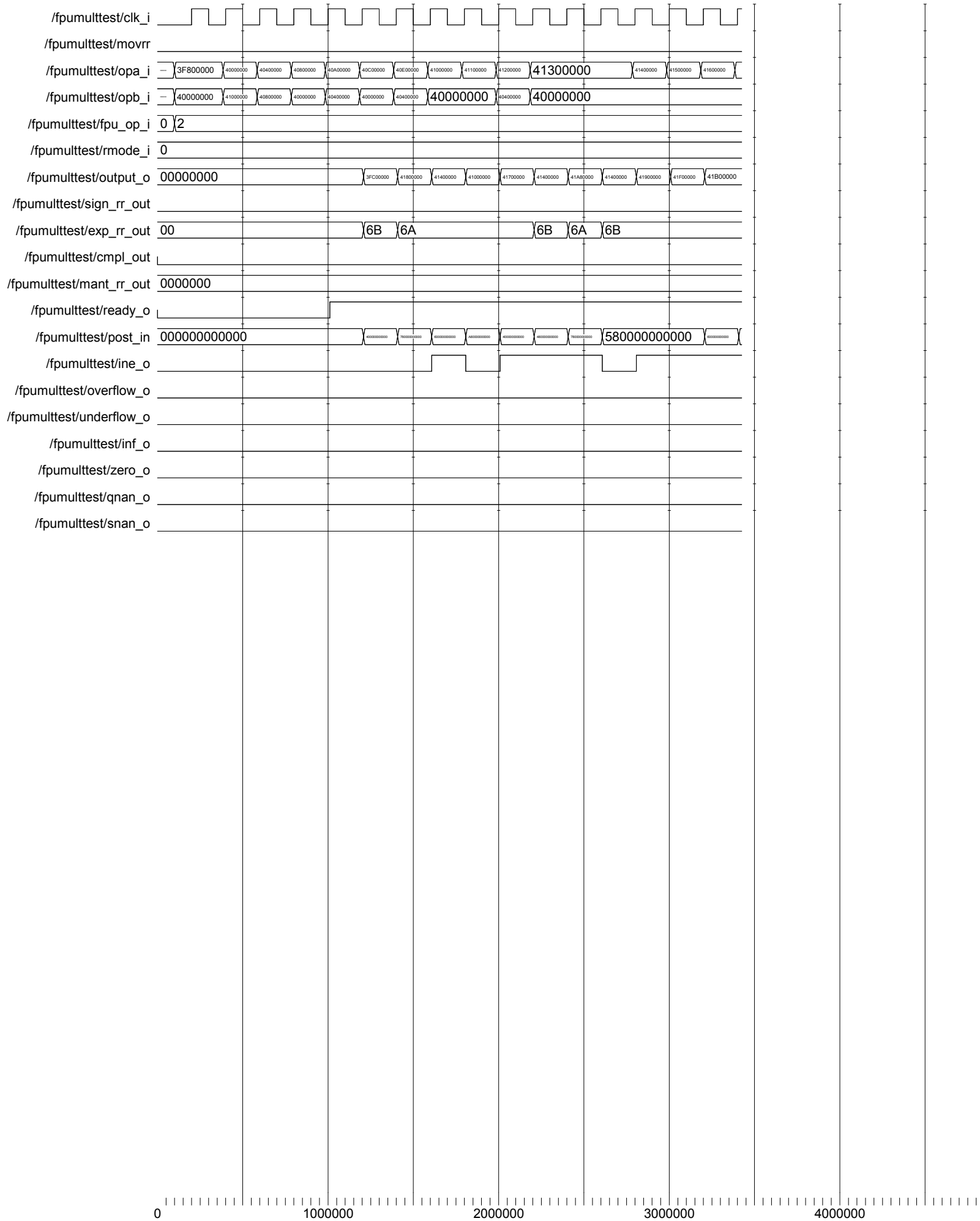
Entity:fpu_add_test_vhd Architecture:behavior Date: Wed Mar 19 2:11:58 AM Eastern Daylight Time 2008 Row: 4 Page: 4

The simulation reports for the Native-pair multiplication are presented in the next two pages. The next page shows the post-route simulation for 8 pairs of operands inputs were given in continuous clock cycles. The 1st signal is the clock input; the 2nd signal is the MOVRR input; 3rd and 4th signals are the operands A and B represented as opa_i and opb_i; 5th signal is the opcode 010 - multiplication; 6th signal is the rounding mode; 7th signal is the output of the FPU, it could be the result or the normalized residual value depending on the MOVRR signal; 8th and 9th signals are the sign output and exponent output of the residual register. 10th signal is the complement flag output and is used to check the proper functioning of the residual register hardware. 11th signal is the residual register mantissa output. 12th signal is the ready signal used to indicate the valid output of the FPU multiplier. 12th signal is the post_in signal and it used to check if the right residual value was going into the postnormalization unit when MOVRR goes high; Apart from these, signals from 13th to 19th give the inexact, overflow and the exception outputs. The FPU multiplication takes place in 5 clock cycles, one clock cycle each for prenormalization and multiplication, two clock cycles for postnormalization and once clock cycle for formatting output.

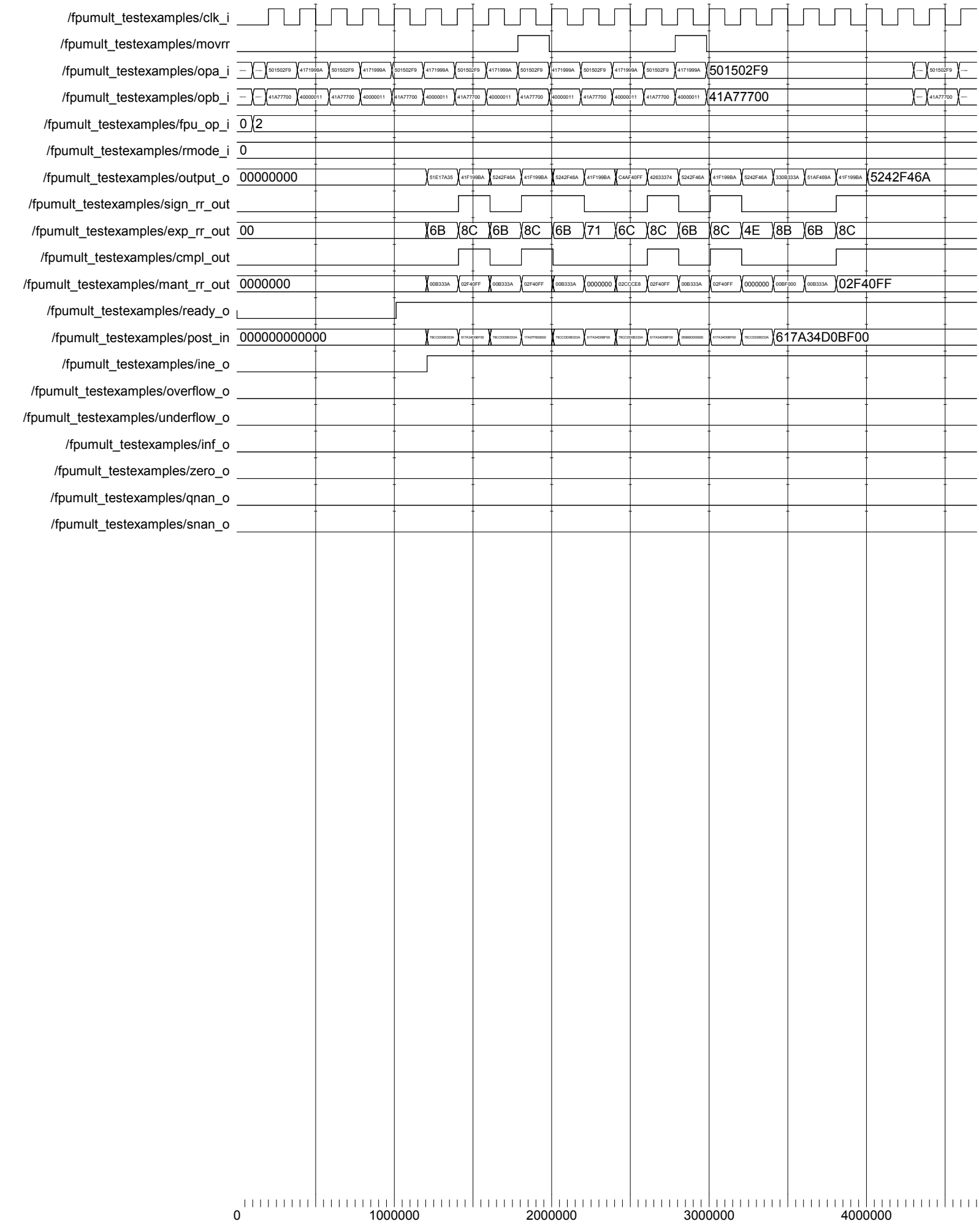
In the wave form shown in the next page, the signal 3 indicates the operand A - opa_i; it is 501502F9 in the 1st clock cycle, 417199A in the 2nd and is periodically repeated. 4th signal is operand B – opb_i which takes value 41A77700 in 1st cycle, 400000011 in 2nd clock cycle and there after repeats itself alternatively with 41A77700 and 40000011. This has been done in order to show the residual register value resulting from multiplication of these operands. The outputs are obtained from 6th clock cycle onwards. Consider the inputs in the 2nd clock cycle that is opa_i = 4171999A and opb_i = 40000011 and product is shown by the 7th signal output_o = 41F199BA obtained in the 7th clock cycle with the rising edge of the clock. The outputs for the other inputs come along the consequent clock cycles. MOVRR signal goes high at the end of the 7th clock cycle, at this time the post_in becomes the residual register mantissa value obtained in the previous clock cycle. The normalized residual value to be stored in the architectural register C4AF40FF is obtained in the 10th clock cycle with the rising edge of the clock. This is the residual

register value for the inputs opa_i = 501502F9 and opb_i = 41A77700. Similarly MOVRR again goes high in the 12^{th} clock cycle to give the normalized residual register value 330B333A as output in the 15^{th} clock cycle which is the residual value for opa_i =4171999A and opb_i = 40000011. The output of opa_i = 501502F9 and opb_i = 41A77700 can be observed in 8^{th} clock cycle with the input being given in the 3^{rd} clock cycle.

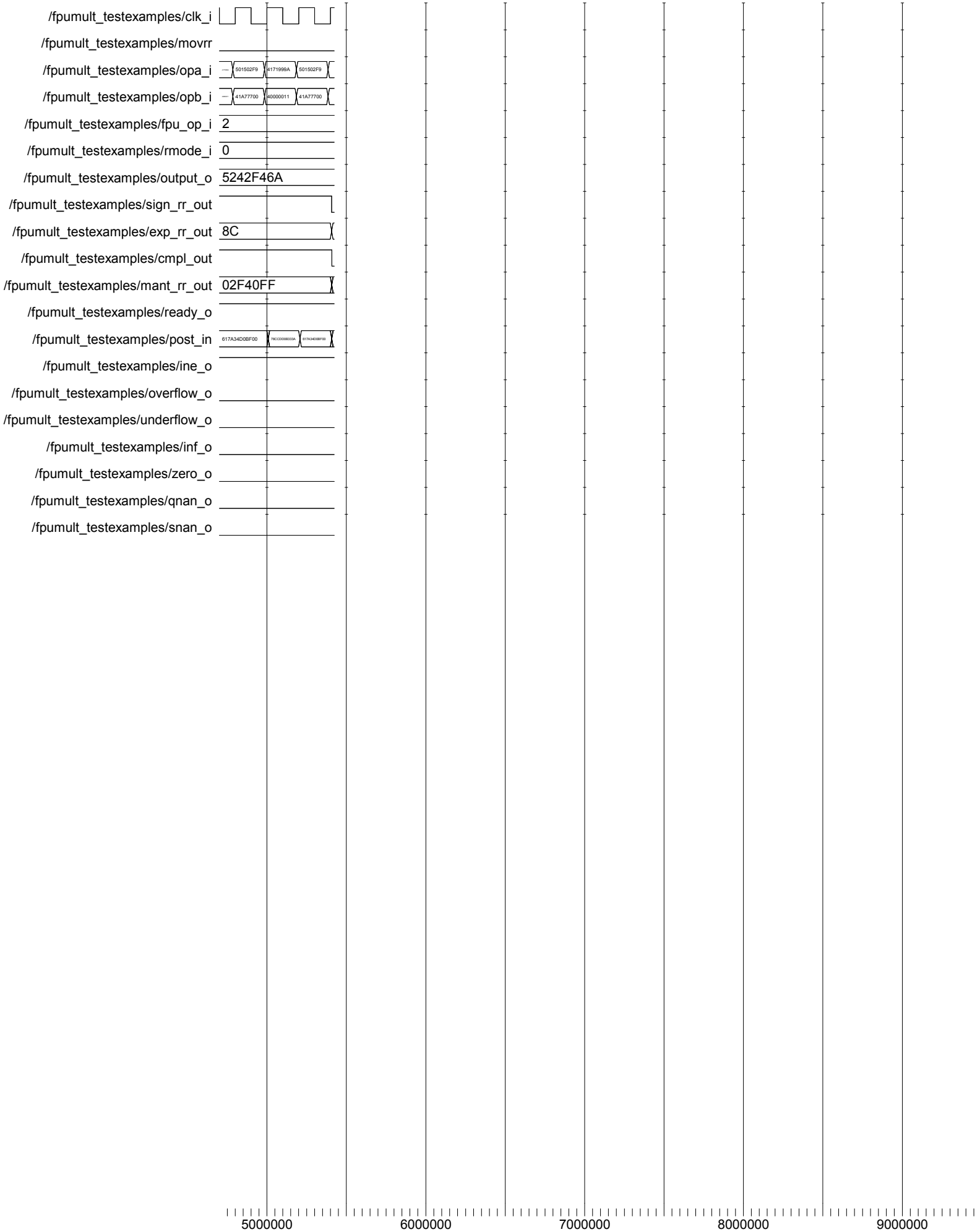
The second wave form has been run to check if the all the proper functioning of the FPU multiplier unit.



Entity:fpumulttest Architecture:testbench_arch Date: Mon Mar 24 2:45:40 AM Eastern Daylight Time 2008 Row: 1 Page: 1



Entity:fpumult_testexamples Architecture:testbench_arch Date: Mon Mar 24 5:57:02 AM Eastern Daylight Time 2008 Row: 1 Page: 1


/fpumult_testexamples/clk_i
/fpumult_testexamples/movrr
/fpumult_testexamples/opa_i
501502F9
4171999A
501502F9
/fpumult_testexamples/opb_i
41A77700
40000011
41A77700
/fpumult_testexamples/fpu_op_i
2
/fpumult_testexamples/rmode_i
0
/fpumult_testexamples/output_o
5242F46A
/fpumult_testexamples/sign_rr_out
/fpumult_testexamples/exp_rr_out
8C
/fpumult_testexamples/cmpl_out
/fpumult_testexamples/mant_rr_out
02F40FF
/fpumult_testexamples/ready_o
/fpumult_testexamples/post_in
617A34D0BF00
/fpumult_testexamples/ine_o
/fpumult_testexamples/overflow_o
/fpumult_testexamples/underflow_o
/fpumult_testexamples/inf_o
/fpumult_testexamples/zero_o
/fpumult_testexamples/qnan_o
/fpumult_testexamples/snan_o
5000000
6000000
7000000
8000000
9000000

Appendix B

High-level Schematics

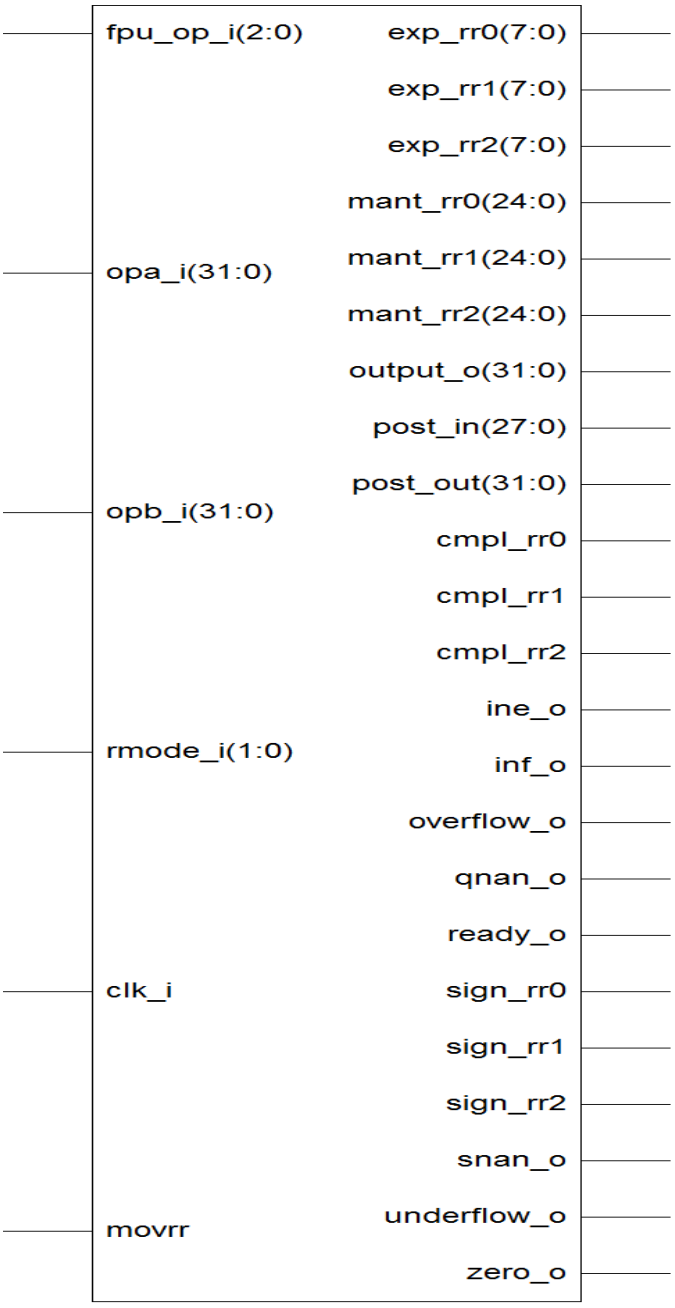


Figure 18: High-level schematic of FPU Adder

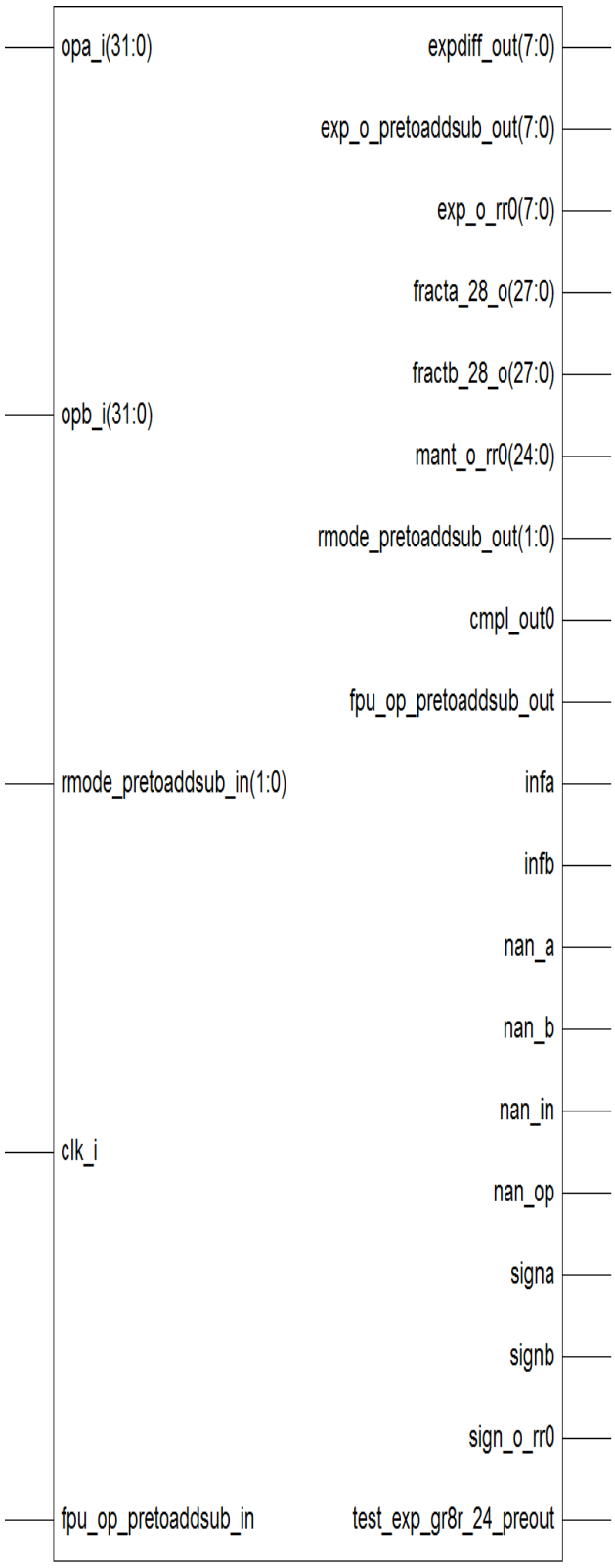


Figure 19. High-level schematic Prenormalization unit used in Floating-point addition.

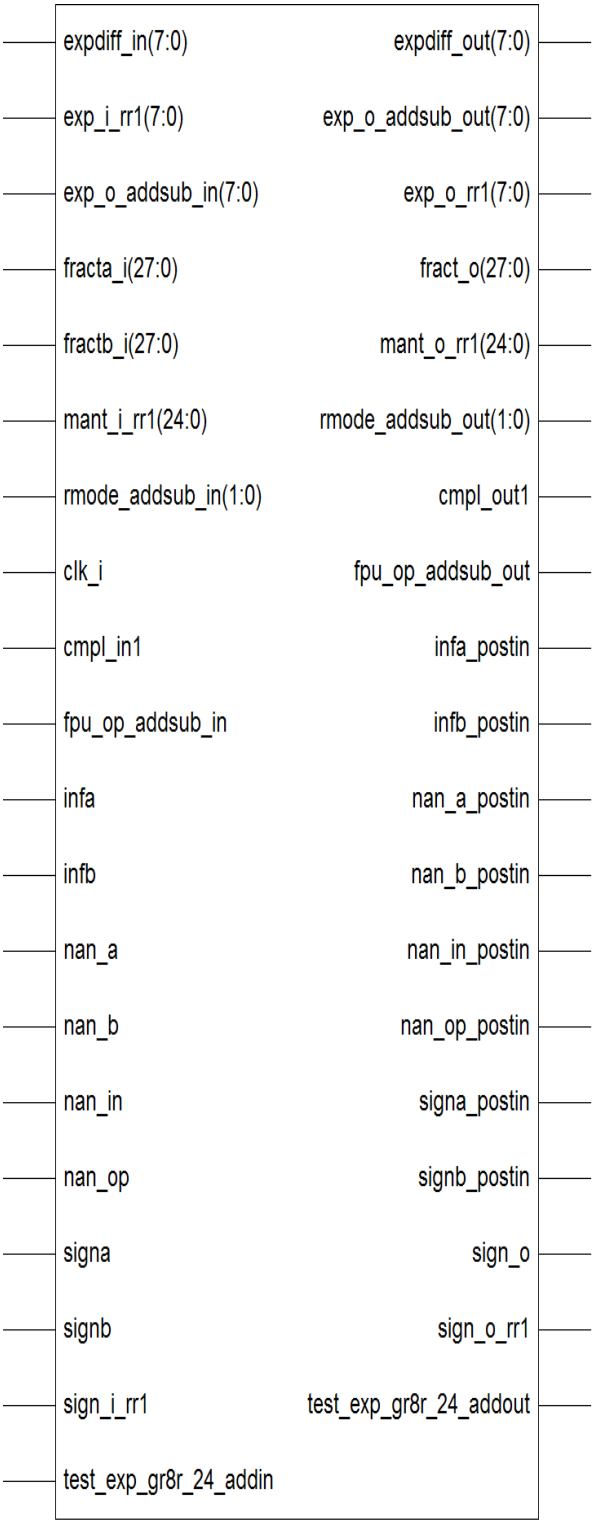


Figure 20. High-level schematic of Addition unit used in Floating-point addition

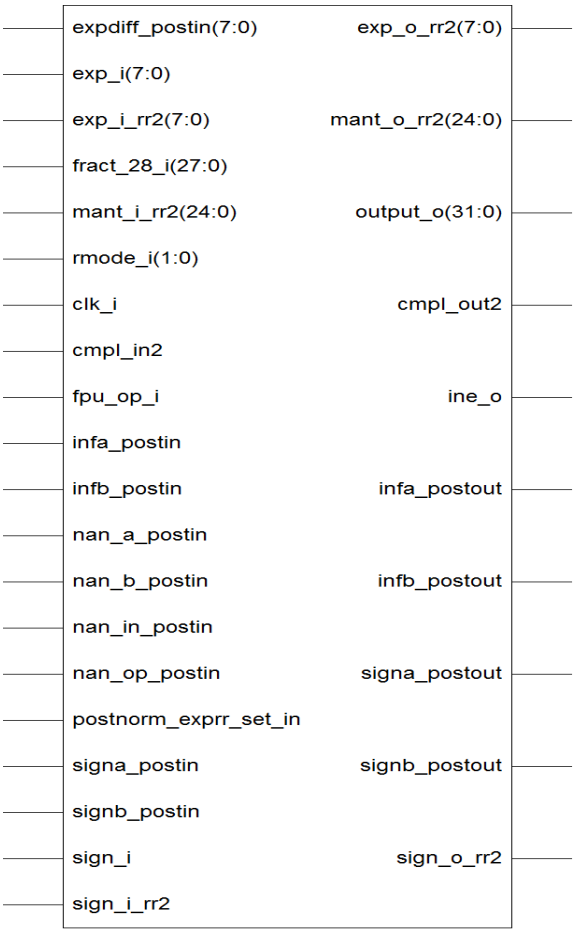


Figure 21. High-level schematic of Postnormalization Unit used in Floating-point addition

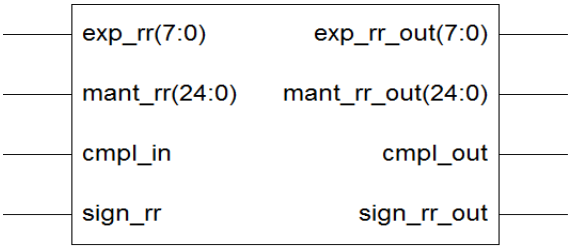


Figure 22. High-level schematic of Residual register used in prenormalization and postnormalization

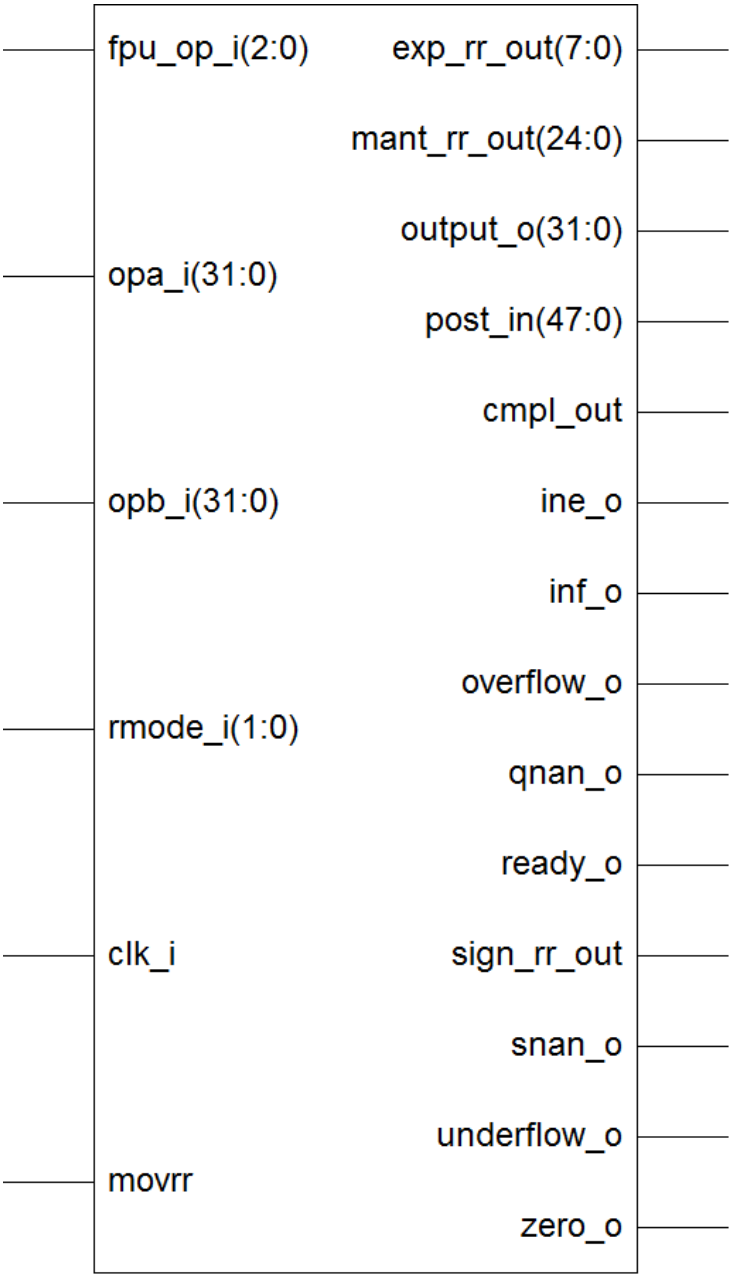


Figure 23. High-level schematic of FPU multiplier

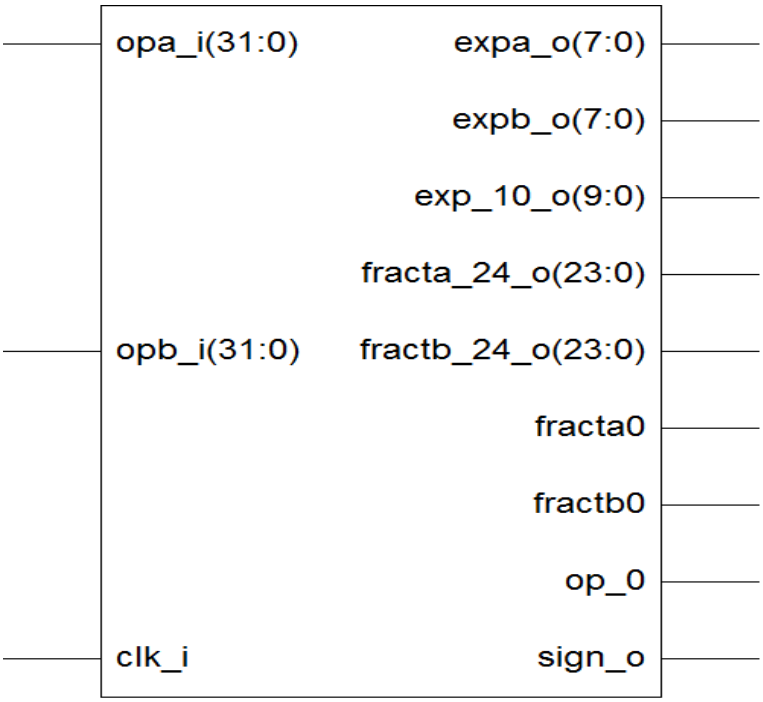


Figure 24. High-level schematic of Prenormalization unit for Multiplier

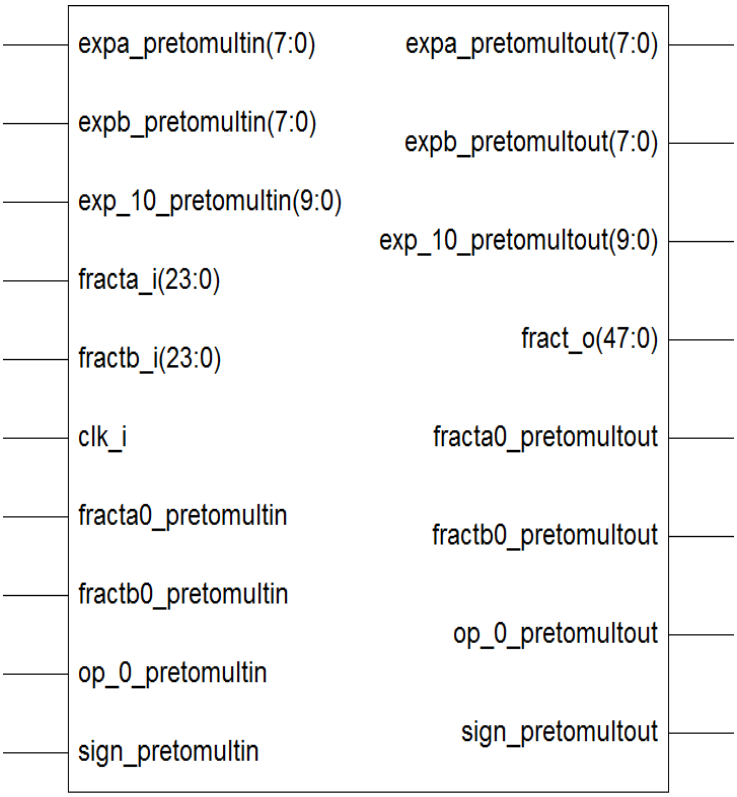


Figure 25. High-level schematic of Multiplier unit

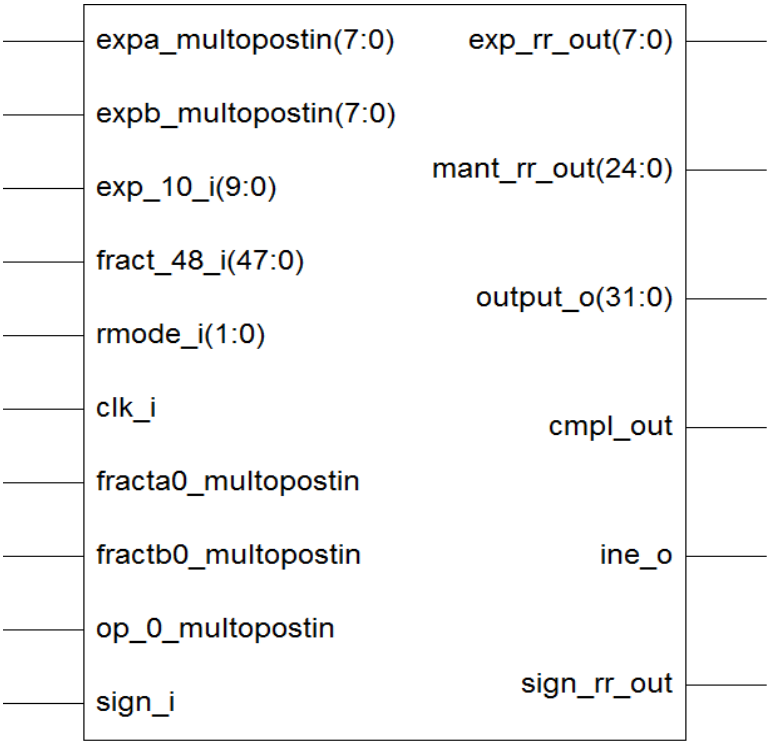


Figure 26. High-level schematic of Postnormalization for Multiplier

VHDL Source Code

FPU Adder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_ARITH.all;

library work;

use work.fpupack.all;


entity fpu_add is
   port (
      clk_i,movrr: in std_logic;
      -- Input Operands A & B
      opa_i              : in std_logic_vector(FP_WIDTH-1 downto 0);  -- Default:
FP_WIDTH=32
      opb_i         : in std_logic_vector(FP_WIDTH-1 downto 0);
      -- fpu operations (fpu_op_i):
                  -- ========================
                  -- 000 = add,
                  -- 001 = substract,
      fpu_op_i          : in std_logic_vector(2 downto 0);
      -- Rounding Mode:

      rmode_i           : in std_logic_vector(1 downto 0);
      -- Output port
      output_o , post_out      : out std_logic_vector(FP_WIDTH-1 downto 0);
      -- Exceptions
      ine_o                      : out std_logic; -- inexact
      overflow_o        : out std_logic; -- overflow
      underflow_o       : out std_logic; -- underflow
      inf_o                      : out std_logic; -- infinity
      zero_o                     : out std_logic; -- zero
      qnan_o                     : out std_logic; -- queit Not-a-Number
      snan_o                     : out std_logic; -- signaling Not-a-Number

                  post_in:out std_logic_vector(27 downto 0);--;
         --residuals
         sign_rr0,sign_rr1,sign_rr2,cmpl_rr0,cmpl_rr1,cmpl_rr2,ready_o:out std_logic;
         exp_rr0,exp_rr1,exp_rr2:out std_logic_vector(EXP_WIDTH-1 downto 0);
```

```
          mant_rr0,mant_rr1,mant_rr2:out std_logic_vector(FRAC_WIDTH + 1 downto
0)

        );
end fpu_add;

architecture rtl of fpu_add is

        -- Input/output registers
        signal s_opa_i, s_opb_i : std_logic_vector(FP_WIDTH-1 downto 0);
        signal s_fpu_op_i               : std_logic_vector(2 downto 0);
        signal s_rmode_i : std_logic_vector(1 downto 0);
        signal s_output_o : std_logic_vector(FP_WIDTH-1 downto 0);
  signal s_ine_o, s_overflow_o, s_underflow_o,  s_inf_o, s_zero_o, s_qnan_o, s_snan_o :
std_logic;

        signal s_output1 : std_logic_vector(FP_WIDTH-1 downto 0);

        --        ***Add/Substract units signals***
        signal s_mant_rr2:std_logic_vector(FRAC_WIDTH + 1 downto 0);
        signal post_norm_fract_in:std_logic_vector(FRAC_WIDTH + 4 downto 0);
        signal post_norm_exp_in:std_logic_vector(EXP_WIDTH-1 downto 0);
  signal post_norm_sign_in:std_logic;
    ------------pipelining signals-----------------------------------------------------

   signal fpu_op_addsub:std_logic;--
   signal rmode_pretoaddsub:std_logic_vector(1 downto 0);--
   signal prenorm_addsub_fracta_28_o:std_logic_vector(FRAC_WIDTH+4 downto 0);
   signal prenorm_addsub_fractb_28_o:std_logic_vector(FRAC_WIDTH+4 downto 0);
   signal prenorm_addsub_exp:std_logic_vector(EXP_WIDTH-1 downto 0);
   signal test_exp_gr8r_24_addin:std_logic;
   signal s_sign_rrpretoadd:std_logic;
   signal s_cmpl_rrpretoadd:std_logic;
   signal s_exp_rrpretoadd:std_logic_vector(EXP_WIDTH-1 downto 0);
   signal s_mant_rrpretoadd:std_logic_vector(FRAC_WIDTH + 1 downto 0);
         signal addsub_fract_o: std_logic_vector(FRAC_WIDTH+4 downto 0);
   signal addsub_sign_o : std_logic;
   signal rmode_addsubpost:std_logic_vector(1 downto 0);--
   signal exp_o_addsubpost:std_logic_vector(EXP_WIDTH -1 downto 0);--
   signal test_exp_gr8r_24_addsubpost:std_logic;--

   signal postnorm_addsub_output_o : std_logic_vector(31 downto 0);
   signal postnorm_addsub_ine_o : std_logic;

   signal fpu_op_addsubpost :std_logic;--
   signal s_sign_rraddtopost,s_cmpl_rraddtopost:std_logic;
```

```
    signal s_exp_rraddtopost:std_logic_vector(EXP_WIDTH-1 downto 0);
    signal s_mant_rraddtopost:std_logic_vector(FRAC_WIDTH + 1 downto 0);
    signal s_sign_rr2,s_cmpl_rr2:std_logic;
    signal s_exp_rr2:std_logic_vector(EXP_WIDTH-1 downto 0);

  component prenorm_new is
        port(
                        clk_i                   : in std_logic;
                        opa_i                   : in std_logic_vector(FP_WIDTH-1 downto
0);
                        opb_i                   : in std_logic_vector(FP_WIDTH-1 downto
0);
                        fpu_op_pretoaddsub_in:      in std_logic;--
                        rmode_pretoaddsub_in:       in std_logic_vector(1 downto 0);--
                        fpu_op_pretoaddsub_out:     out std_logic;--
                        rmode_pretoaddsub_out:      out std_logic_vector(1 downto 0);--
                        fracta_28_o             : out std_logic_vector(FRAC_WIDTH+4
downto 0);      -- carry(1) & hidden(1) & fraction(23) & guard(1) & round(1) & sticky(1)
                        fractb_28_o             : out std_logic_vector(FRAC_WIDTH+4
downto 0);

                        exp_o_pretoaddsub_out                       : out
std_logic_vector(EXP_WIDTH-1 downto 0);--
                        test_exp_gr8r_24_preout:out std_logic;--
                        sign_o_rr0,cmpl_out0:out std_logic;
                        exp_o_rr0:out std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_o_rr0:out std_logic_vector(FRAC_WIDTH + 1 downto 0);

                        expdiff_out:out std_logic_vector(EXP_WIDTH-1 downto 0);
                        infa,infb,signa,signb,nan_a,nan_b,nan_in,nan_op:out std_logic
                );
  end component;


  component  addsub_28 is
        port(
                        clk_i                   : in std_logic;
                        fracta_i                : in std_logic_vector(FRAC_WIDTH+4
downto 0); -- carry(1) & hidden(1) & fraction(23) & guard(1) & round(1) & sticky(1)
                        fractb_i                : in std_logic_vector(FRAC_WIDTH+4
downto 0);
                        fpu_op_addsub_in    :in std_logic;--
                        rmode_addsub_in     :in std_logic_vector(1 downto 0);--
                        exp_o_addsub_in     : in std_logic_vector(EXP_WIDTH-1
downto 0);--
                        test_exp_gr8r_24_addin:in std_logic;--
```

```
                        exp_i_rr1       :in std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_i_rr1      :in std_logic_vector(FRAC_WIDTH + 1 downto 0);
                        sign_i_rr1,cmpl_in1   : in std_logic;
                        expdiff_in:in std_logic_vector(EXP_WIDTH-1 downto 0);
                        infa,infb,signa,signb,nan_a,nan_b,nan_in,nan_op:in std_logic;
                        fract_o                 : out std_logic_vector(FRAC_WIDTH+4
downto 0);
                        sign_o                  : out std_logic;
                        rmode_addsub_out:out std_logic_vector(1 downto 0);--
                        exp_o_addsub_out:out std_logic_vector(EXP_WIDTH -1 downto
0);--
        test_exp_gr8r_24_addout:out std_logic;--
                        fpu_op_addsub_out    :out std_logic;--
                        sign_o_rr1,cmpl_out1 :out std_logic;
                        exp_o_rr1       :out std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_o_rr1      :out std_logic_vector(FRAC_WIDTH + 1 downto
0);
                        expdiff_out:out std_logic_vector(EXP_WIDTH-1 downto 0);
                        infa_postin,infb_postin,signa_postin,signb_postin:out std_logic;
                        nan_a_postin,nan_b_postin,nan_in_postin,nan_op_postin:out
std_logic);
  end component;

  component postnorm_june20 is
        port(
                        clk_i                   : in std_logic;
                        fract_28_i              : in std_logic_vector(FRAC_WIDTH+4
downto 0);      -- carry(1) & hidden(1) & fraction(23) & guard(1) & round(1) & sticky(1)
                        exp_i                   : in std_logic_vector(EXP_WIDTH-1
downto 0);
                        sign_i                  : in std_logic;
                        postnorm_exprr_set_in:in std_logic;
                        exp_i_rr2:in std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_i_rr2:in std_logic_vector(FRAC_WIDTH +1 downto 0);
                        sign_i_rr2,cmpl_in2: in std_logic;
                        fpu_op_i                : in std_logic;
                        rmode_i                         : in std_logic_vector(1 downto 0);
                        expdiff_postin:in std_logic_vector(EXP_WIDTH-1 downto 0);

                        infa_postin,infb_postin,signa_postin,signb_postin:in std_logic;
                        nan_a_postin,nan_b_postin,nan_in_postin,nan_op_postin:in
std_logic;
                        output_o                : out std_logic_vector(FP_WIDTH-1
downto 0);
                        infa_postout,infb_postout: out std_logic;
                        signa_postout,signb_postout: out std_logic;
```

```
                        exp_o_rr2:out std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_o_rr2:out std_logic_vector(FRAC_WIDTH + 1 downto 0);
      ine_o,sign_o_rr2,cmpl_out2: out std_logic
                );
  end component;

 signal ready: std_logic;
 signal cnt: integer:=0;
 signal expdiffpre_add,expdiffadd_post:std_logic_vector(EXP_WIDTH-1 downto
0):="00000000";
 signal s_infa,s_infb,s_signa,s_signb,s_nan_a,s_nan_b,s_nan_in,s_nan_op: std_logic;
 signal
s_infa_postin,s_infb_postin,s_signa_postin,s_signb_postin,s_nan_a_postin,s_nan_b_post
in,s_nan_in_postin,s_nan_op_postin: std_logic;
 signal s_infa_postout,s_infb_postout,s_signa_postout,s_signb_postout: std_logic;

  begin

    i_prenorm_addsub: prenorm_new
    port map (
    clk_i => clk_i,
    opa_i => s_opa_i,
    opb_i => s_opb_i,
    fpu_op_pretoaddsub_in => s_fpu_op_i(0),
    rmode_pretoaddsub_in => s_rmode_i,
          fpu_op_pretoaddsub_out => fpu_op_addsub,
    rmode_pretoaddsub_out =>rmode_pretoaddsub,
          fracta_28_o => prenorm_addsub_fracta_28_o,
    fractb_28_o => prenorm_addsub_fractb_28_o,
    exp_o_pretoaddsub_out=> prenorm_addsub_exp,
    test_exp_gr8r_24_preout => test_exp_gr8r_24_addin,
    sign_o_rr0=>s_sign_rrpretoadd,
    cmpl_out0=>s_cmpl_rrpretoadd,
    exp_o_rr0=>s_exp_rrpretoadd,
    mant_o_rr0=>s_mant_rrpretoadd,
          expdiff_out=>expdiffpre_add,
          infa=>s_infa,infb=>s_infb,signa=>s_signa,
          signb=>s_signb,nan_a=>s_nan_a,
          nan_b=>s_nan_b,nan_in=>s_nan_in,nan_op=>s_nan_op);


    i_addsub: addsub_28
    port map(
    clk_i => clk_i,
    fracta_i=> prenorm_addsub_fracta_28_o,
    fractb_i=> prenorm_addsub_fractb_28_o,
```

```
fpu_op_addsub_in => fpu_op_addsub,
rmode_addsub_in => rmode_pretoaddsub,
        exp_o_addsub_in => prenorm_addsub_exp,
test_exp_gr8r_24_addin => test_exp_gr8r_24_addin,
exp_i_rr1=>s_exp_rrpretoadd,
        mant_i_rr1=>s_mant_rrpretoadd,
sign_i_rr1=>s_sign_rrpretoadd,
cmpl_in1=>s_cmpl_rrpretoadd,
        expdiff_in=>expdiffpre_add,
        infa=>s_infa,infb=>s_infb,
        signa=>s_signa,signb=>s_signb,
        nan_a=>s_nan_a,nan_b=>s_nan_b,
        nan_in=>s_nan_in,nan_op=>s_nan_op,
fract_o => addsub_fract_o,
sign_o => addsub_sign_o,
rmode_addsub_out => rmode_addsubpost,
exp_o_addsub_out => exp_o_addsubpost,
test_exp_gr8r_24_addout => test_exp_gr8r_24_addsubpost,
fpu_op_addsub_out => fpu_op_addsubpost,
        sign_o_rr1=>s_sign_rraddtopost,
cmpl_out1=>s_cmpl_rraddtopost,
exp_o_rr1=>s_exp_rraddtopost,
mant_o_rr1=>s_mant_rraddtopost,
        expdiff_out=>expdiffadd_post,
        infa_postin=>s_infa_postin,infb_postin=>s_infb_postin,
        signa_postin=>s_signa_postin,signb_postin=>s_signb_postin,
        nan_a_postin=>s_nan_a_postin,nan_b_postin=>s_nan_b_postin,
        nan_in_postin=>s_nan_in_postin,nan_op_postin=>s_nan_op_postin);


i_postnorm_addsub: postnorm_june20
port map(
clk_i => clk_i,
fract_28_i => post_norm_fract_in,
exp_i => post_norm_exp_in,
sign_i => post_norm_sign_in,
postnorm_exprr_set_in=>test_exp_gr8r_24_addsubpost,
exp_i_rr2=>s_exp_rraddtopost,
mant_i_rr2=>s_mant_rraddtopost,
sign_i_rr2=>s_sign_rraddtopost,
cmpl_in2=>s_cmpl_rraddtopost,
fpu_op_i => fpu_op_addsubpost,
rmode_i => rmode_addsubpost,
        expdiff_postin=>expdiffadd_post,
        infa_postin=>s_infa_postin,infb_postin=>s_infb_postin,
        signa_postin=>s_signa_postin,signb_postin=>s_signb_postin,
```

```
        nan_a_postin=>s_nan_a_postin,nan_b_postin=>s_nan_b_postin,
        nan_in_postin=>s_nan_in_postin,nan_op_postin=>s_nan_op_postin,
    output_o => postnorm_addsub_output_o,
        infa_postout=>s_infa_postout,infb_postout=>s_infb_postout,
        signa_postout=>s_signa_postout,signb_postout=>s_signb_postout,
        exp_o_rr2=>s_exp_rr2,
    mant_o_rr2=>s_mant_rr2,
    ine_o => postnorm_addsub_ine_o,
    sign_o_rr2=>s_sign_rr2,
    cmpl_out2=>s_cmpl_rr2);

 --Multplexer for either supplying add/sub output or residual reg value to the post
normalization unit

  post_norm_fract_in<=(s_mant_rr2 &"000")when (movrr='1')else addsub_fract_o;
  post_norm_exp_in<=s_exp_rr2 when (movrr='1')else exp_o_addsubpost;
  post_norm_sign_in<=s_sign_rr2 when (movrr='1')else addsub_sign_o;

post_in<=post_norm_fract_in;
post_out<=postnorm_addsub_output_o;

----------------------------------------------------------------

        -- Input Register

                        s_opa_i <= opa_i ;
                        s_opb_i <= opb_i ;
                        s_fpu_op_i <= fpu_op_i;
                        s_rmode_i <= rmode_i;



        --Output Register
        process(clk_i)
        begin
        if falling_edge(clk_i) then
   if (ready = '1')then
                        output_o <= s_output_o;
                        ine_o <= s_ine_o;
                        overflow_o <= s_overflow_o;
                        underflow_o <= s_underflow_o;

                        inf_o <= s_inf_o;
                        zero_o <= s_zero_o;
                        qnan_o <= s_qnan_o;
                        snan_o <= s_snan_o;
```

```
                end if;

        end if;
        end process;


                sign_rr0<=s_sign_rrpretoadd;
                cmpl_rr0<=s_cmpl_rrpretoadd;
                exp_rr0<=s_exp_rrpretoadd;
                mant_rr0<=s_mant_rrpretoadd;

                sign_rr1<=s_sign_rraddtopost;
                cmpl_rr1<=s_cmpl_rraddtopost;
                exp_rr1<=s_exp_rraddtopost;
                mant_rr1<=s_mant_rraddtopost;

                mant_rr2<=s_mant_rr2;
                exp_rr2<=s_exp_rr2;
                sign_rr2<=s_sign_rr2;
                cmpl_rr2<=s_cmpl_rr2;


        -- Output Multiplexer

        process(clk_i)
        begin
                if rising_edge(clk_i) then
                        if fpu_op_i="000" or fpu_op_i="001" then
                                s_output1               <= postnorm_addsub_output_o;
                                s_ine_o                 <= postnorm_addsub_ine_o;

                        else
                                s_output1       <= (others => '0');
                                s_ine_o                 <= '0';
                        end if;
                end if;
        end process;


        --In round down: the subtraction of two equal numbers other than zero are always
-0!!!
        process(s_output1, s_rmode_i,  s_infa_postout, s_infb_postout, s_qnan_o,
s_snan_o, s_zero_o, s_fpu_op_i, s_signa_postout, s_signb_postout)
        begin
                        if s_rmode_i="00" or ((s_infa or s_infb) or s_qnan_o or
s_snan_o)='1' then --round-to-nearest-even
```

```
                        s_output_o <= s_output1;
                    elsif s_rmode_i="01" and s_output1(30 downto 23)="11111111"
then
                        --In round-to-zero: the sum of two non-infinity operands is
never infinity,even if an overflow occures
                        s_output_o <= s_output1(31) &
"11111110111111111111111111111111";
                    elsif s_rmode_i="10" and s_output1(31 downto 23)="111111111"
then
                        --In round-up: the sum of two non-infinity operands is
never negative infinity,even if an overflow occures
                        s_output_o <= "11111111011111111111111111111111";
                    elsif s_rmode_i="11" then
                        --In round-down: a-a= -0
                    if (s_fpu_op_i="000" or s_fpu_op_i="001") and s_zero_o='1' and
(s_opa_i(31) or (s_fpu_op_i(0) xor s_opb_i(31)))='1' then
                            s_output_o <= "1" & s_output1(30 downto 0);
                        --In round-down: the sum of two non-infinity operands is
never postive infinity,even if an overflow occures
                    elsif s_output1(31 downto 23)="011111111" then
                            s_output_o <=
"01111111011111111111111111111111";
                    else
                            s_output_o <= s_output1;
                    end if;
                    else
                        s_output_o <= s_output1;
                    end if;
        end process;


        -- Generate Exceptions
        s_underflow_o <= '1' when s_output1(30 downto 23)="00000000" and
s_ine_o='1' else '0';
        s_overflow_o <= '1' when s_output1(30 downto 23)="11111111" and s_ine_o='1'
else '0';

        s_inf_o <= '1' when s_output1(30 downto 23)="11111111" and (s_qnan_o or
s_snan_o)='0' else '0';
        s_zero_o <= '1' when or_reduce(s_output1(30 downto 0))='0' else '0';
        s_qnan_o <= '1' when s_output1(30 downto 0)=QNAN else '0';
  s_snan_o <= '1' when s_opa_i(30 downto 0)=SNAN or s_opb_i(30 downto 0)=SNAN
else '0';

        ----Ready signal to indicate start of valid outputs --
        process(clk_i)
```

```
        begin
        if(falling_edge(clk_i))then
        if(cnt/=2)then
        cnt <= cnt + 1;
        else
        cnt <= cnt;
        end if;

        if(cnt=2)then
        ready<='1';
        else
        ready<='0';
        end if;
        end if;
        end process;

  ready_o<=ready;

 end rtl;
```

---prenormalization unit---

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_ARITH.all;

library work;
use work.fpupack.all;

entity prenorm_new is
        port(
                        clk_i                   : in std_logic;
                        opa_i                   : in std_logic_vector(FP_WIDTH-1 downto
0);
                        opb_i                   : in std_logic_vector(FP_WIDTH-1 downto
0);
                        fpu_op_pretoaddsub_in:      in std_logic;--
                        rmode_pretoaddsub_in:       in std_logic_vector(1 downto 0);--
                        fpu_op_pretoaddsub_out:     out std_logic;--
                        rmode_pretoaddsub_out:      out std_logic_vector(1 downto 0);--
                        fracta_28_o             : out std_logic_vector(FRAC_WIDTH+4
downto 0);      -- carry(1) & hidden(1) & fraction(23) & guard(1) & round(1) & sticky(1)
                        fractb_28_o             : out std_logic_vector(FRAC_WIDTH+4
downto 0);
```

```
                exp_o_pretoaddsub_out                   : out
std_logic_vector(EXP_WIDTH-1 downto 0);--
                test_exp_gr8r_24_preout:out std_logic;--
            sign_o_rr0,cmpl_out0:out std_logic;
                exp_o_rr0:out std_logic_vector(EXP_WIDTH-1 downto 0);
                mant_o_rr0:out std_logic_vector(FRAC_WIDTH + 1 downto 0);

                expdiff_out:out std_logic_vector(EXP_WIDTH-1 downto 0);
                infa,infb,signa,signb,nan_a,nan_b,nan_in,nan_op:out std_logic
        );
end prenorm_new;


architecture rtl of prenorm_new is


        signal s_exp_o : std_logic_vector(EXP_WIDTH-1 downto 0);
        signal s_fracta_28_o, s_fractb_28_o : std_logic_vector(FRAC_WIDTH+4
downto 0);
        signal s_expa, s_expb : std_logic_vector(EXP_WIDTH-1 downto 0);
        signal s_fracta, s_fractb : std_logic_vector(FRAC_WIDTH-1 downto 0);

        signal s_fracta_28, s_fractb_28, s_fract_sm_28:
std_logic_vector(FRAC_WIDTH+4 downto 0);

        signal s_exp_diff,s_exp_sm : std_logic_vector(EXP_WIDTH-1 downto 0);
        signal s_rzeros : std_logic_vector(5 downto 0);

  signal s_expa_lt_expb : std_logic;
        signal s_expa_eq_expb : std_logic;
        signal s_fracta_1 : std_logic;
        signal s_fractb_1 : std_logic;
 signal s_op_dn : std_logic;
        signal s_opa_dn, s_opb_dn : std_logic;
        signal s_mux_diff : std_logic_vector(1 downto 0);
        signal s_mux_exp,exp_gr8r_24 : std_logic;
        signal s_sticky : std_logic;
        signal s_rr_mant:std_logic_vector(FRAC_WIDTH + 1 downto 0);
        signal s_expdiff_int:integer:=0;

  signal s_fract_shr_28:std_logic_vector(FRAC_WIDTH+4 downto 0);
        signal andsig:std_logic_vector(FRAC_WIDTH+1 downto 0);
  signal rr_rev:std_logic_vector(FRAC_WIDTH+1 downto 0);

  signal s_sign_o_rr0,s_cmpl_out0:std_logic:='Z';
        signal s_exp_o_rr0:std_logic_vector(EXP_WIDTH-1 downto 0);
```

```
        signal s_mant_o_rr0:std_logic_vector(FRAC_WIDTH+1 downto 0) ;

        signal s_infa,s_infb,s_nan_a,s_nan_b,s_nan_in,s_nan_op:std_logic;

component residualreg is port(sign_rr:in std_logic;exp_rr:in
std_logic_vector(EXP_WIDTH - 1 downto 0);cmpl_in:in std_logic;
mant_rr:in std_logic_vector(FRAC_WIDTH+1 downto 0);sign_rr_out:out
std_logic;exp_rr_out:out std_logic_vector(EXP_WIDTH - 1 downto 0);
cmpl_out:out std_logic;mant_rr_out:out std_logic_vector(FRAC_WIDTH+1 downto 0));
end component residualreg;


begin

        -- Input Register

                        s_expa <= opa_i(30 downto 23);
                        s_expb <= opb_i(30 downto 23);
                        s_fracta <= opa_i(22 downto 0);
                        s_fractb <= opb_i(22 downto 0);



        -- Output Register
        process(clk_i)
        begin
        if falling_edge(clk_i) then
                exp_o_pretoaddsub_out <= s_exp_o;
                fracta_28_o <= s_fracta_28_o;
                fractb_28_o <= s_fractb_28_o;
    fpu_op_pretoaddsub_out<=fpu_op_pretoaddsub_in;
                rmode_pretoaddsub_out<=rmode_pretoaddsub_in;
                test_exp_gr8r_24_preout<= exp_gr8r_24;
                expdiff_out<=s_exp_diff;
                sign_o_rr0<=s_sign_o_rr0;
                exp_o_rr0<=s_exp_o_rr0;
                cmpl_out0<=s_cmpl_out0;

    signa<=opa_i(31);
    signb<=opb_i(31);

          infa <= s_infa;
          infb <= s_infb;

                nan_a<=s_nan_a;
                nan_b<=s_nan_b;
```

```
                nan_in<=s_nan_in;
                nan_op<=s_nan_op;

  end if;
        end process;

        mant_o_rr0<=s_mant_o_rr0;

--      s_expa_eq_expb <= '1' when s_expa = s_expb else '0';
        s_expa_lt_expb <= '1' when s_expa > s_expb else '0';


        -- '1' if fraction is not zero
--      s_fracta_1 <= or_reduce(s_fracta);
--      s_fractb_1 <= or_reduce(s_fractb);
--
        -- opa or Opb is denormalized
--      s_op_dn <= s_opa_dn or s_opb_dn;
        s_opa_dn <= not or_reduce(s_expa);
        s_opb_dn <= not or_reduce(s_expb);

--       output the larger exponent

        s_mux_exp <= s_expa_lt_expb;
        process(clk_i)
        begin
                if rising_edge(clk_i) then
                        case s_mux_exp is
                                when '0' => s_exp_o <= s_expb;
                                when '1' => s_exp_o <= s_expa;
                                when others => s_exp_o <= "11111111";
                        end case;
                end if;
        end process;

--       convert to an easy to handle floating-point format
        s_fracta_28 <= "01" & s_fracta & "000" when s_opa_dn='0' else "00" & s_fracta
& "000";
        s_fractb_28 <= "01" & s_fractb & "000" when s_opb_dn='0' else "00" & s_fractb
& "000";

        s_mux_diff <= s_expa_lt_expb & (s_opa_dn xor s_opb_dn);   ---a>b concat
expa/expb..one only = 0 .

        s_exp_diff <= s_expb - s_expa when(s_mux_diff="00")else
                  s_expb - (s_expa+"00000001")when(s_mux_diff="01")else
```

```
                                        s_expa - s_expb when(s_mux_diff="10")else
                                        s_expa -
(s_expb+"00000001")when(s_mux_diff="11")else
                                        "ZZZZZZZZ";


        s_expdiff_int<=conv_integer(s_exp_diff);

        process(clk_i)
        begin
   if rising_edge(clk_i) then
         andsig<="00000000000000000000000000";
   if(s_expdiff_int<25)then
         andsig(s_expdiff_int-1 downto 0)<=(others=>'1');
         else
         andsig<=(others=>'1');
         end if;
   end if;
        end process;

        process(clk_i)
        begin
        if(falling_edge(clk_i))then
        s_rr_mant<=rr_rev;
        end if;
        end process;


        s_fract_sm_28 <= s_fracta_28 when s_expa_lt_expb='0' else s_fractb_28;
        s_exp_sm<=s_expb when s_expa_lt_expb='1' else s_expa;

  s_fract_shr_28 <= shr(s_fract_sm_28,s_exp_diff);

        rr_rev<=s_fract_sm_28(FRAC_WIDTH+4 downto 3) and andsig;
        -- count the zeros from right to check if result is inexact
        s_rzeros <= count_r_zeros(s_fract_sm_28);
        s_sticky <= '1' when s_exp_diff > s_rzeros and or_reduce(s_fract_sm_28)='1' else
'0';

  s_fracta_28_o<=s_fracta_28 when s_expa_lt_expb='1' else s_fract_shr_28(27 downto
1)&(s_sticky or s_fract_shr_28(0));
  s_fractb_28_o<=s_fractb_28 when s_expa_lt_expb='0' else s_fract_shr_28(27 downto
1)&(s_sticky or s_fract_shr_28(0));
```

```
  rr0:residualreg port
map('0',s_exp_sm,'0',s_rr_mant,s_sign_o_rr0,s_exp_o_rr0,s_cmpl_out0,s_mant_o_rr0);


  exp_gr8r_24<= '1' when (s_expdiff_int > 23) else '0';


        s_infa <= '1' when opa_i(30 downto 23)="11111111"  else '0';
        s_infb <= '1' when opb_i(30 downto 23)="11111111"  else '0';
        s_nan_a <= '1' when (s_infa='1' and or_reduce (opa_i(22 downto 0))='1') else '0';
        s_nan_b <= '1' when (s_infb='1' and or_reduce (opb_i(22 downto 0))='1') else '0';
        s_nan_in <= '1' when s_nan_a='1' or  s_nan_b='1' else '0';
        s_nan_op <= '1' when (s_infa and s_infb)='1' and (opa_i(31) xor
(fpu_op_pretoaddsub_in xor opb_i(31)) )='1' else '0'; -- inf-inf=Nan




end rtl;
--------------------------------------------------------------------------------
```

Adder/subtractor

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use IEEE.std_logic_arith.all;

library work;
use work.fpupack.all;

entity addsub_28 is
        port(
                        clk_i                   : in std_logic;
                        fracta_i                : in std_logic_vector(FRAC_WIDTH+4
downto 0); -- carry(1) & hidden(1) & fraction(23) & guard(1) & round(1) & sticky(1)
                        fractb_i                : in std_logic_vector(FRAC_WIDTH+4
downto 0);
                        fpu_op_addsub_in    :in std_logic;--
                        rmode_addsub_in     :in std_logic_vector(1 downto 0);--
                        exp_o_addsub_in     : in std_logic_vector(EXP_WIDTH-1
downto 0);--
                        test_exp_gr8r_24_addin:in std_logic;
                        exp_i_rr1       :in std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_i_rr1      :in std_logic_vector(FRAC_WIDTH + 1 downto 0);
```

```
                        sign_i_rr1,cmpl_in1    : in std_logic;
                        expdiff_in:in std_logic_vector(EXP_WIDTH-1 downto 0);
                        infa,infb,signa,signb,nan_a,nan_b,nan_in,nan_op:in std_logic;
                        fract_o                : out std_logic_vector(FRAC_WIDTH+4
downto 0);
                        sign_o                 : out std_logic;
                        rmode_addsub_out:out std_logic_vector(1 downto 0);--
                        exp_o_addsub_out:out std_logic_vector(EXP_WIDTH -1 downto
0);
                        test_exp_gr8r_24_addout:out std_logic;--
                        fpu_op_addsub_out    :out std_logic;--
                        sign_o_rr1,cmpl_out1 :out std_logic;
                        exp_o_rr1       :out std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_o_rr1      :out std_logic_vector(FRAC_WIDTH + 1 downto
0);
                        expdiff_out:out std_logic_vector(EXP_WIDTH-1 downto 0);
                        infa_postin,infb_postin,signa_postin,signb_postin:out std_logic;
                        nan_a_postin,nan_b_postin,nan_in_postin,nan_op_postin:out
std_logic);
end addsub_28;


architecture rtl of addsub_28 is

signal s_fracta_i, s_fractb_i : std_logic_vector(FRAC_WIDTH+4 downto 0);
signal s_fract_o : std_logic_vector(FRAC_WIDTH+4 downto 0);
signal s_signa_i, s_signb_i, s_sign_o : std_logic;
signal s_fpu_op_i : std_logic;
signal fracta_lt_fractb : std_logic;
signal s_addop: std_logic;

begin

-- Input Register

                s_fracta_i <= fracta_i;
                s_fractb_i <= fractb_i;
                s_signa_i<= signa;
                s_signb_i<= signb;
                s_fpu_op_i <= fpu_op_addsub_in;
--
-- Output Register
process(clk_i)
begin
if falling_edge(clk_i) then
                fract_o <= s_fract_o;
```

```
                sign_o <= s_sign_o;
                rmode_addsub_out<=rmode_addsub_in;
                exp_o_addsub_out<=exp_o_addsub_in;
                test_exp_gr8r_24_addout<=test_exp_gr8r_24_addin;
    infa_postin<=infa;
                infb_postin<=infb;
                signa_postin<=signa;
                signb_postin<=signb;
                nan_a_postin<=nan_a;
                nan_b_postin<=nan_b;
                nan_in_postin<=nan_in;
                nan_op_postin<=nan_op;
    fpu_op_addsub_out<=fpu_op_addsub_in;
                sign_o_rr1<=sign_i_rr1;
    exp_o_rr1<=exp_i_rr1;
    cmpl_out1<=cmpl_in1;
                mant_o_rr1<=mant_i_rr1;
                expdiff_out<=expdiff_in;

end if;
end process;

fracta_lt_fractb <= '1' when s_fracta_i > s_fractb_i else '0';

-- check if its a subtraction or an addition operation
s_addop <= ((s_signa_i xor s_signb_i)and not (s_fpu_op_i)) or ((s_signa_i xnor
s_signb_i)and (s_fpu_op_i));

-- sign of result
s_sign_o <= '0' when s_fract_o = conv_std_logic_vector(0,28) and (s_signa_i and
s_signb_i)='0' else
                                                                ((not
s_signa_i) and ((not fracta_lt_fractb) and (s_fpu_op_i xor s_signb_i))) or
                                                                ((s_signa_i)
and (fracta_lt_fractb or (s_fpu_op_i xor s_signb_i)));

-- add/substract
process(s_fracta_i, s_fractb_i, s_addop, fracta_lt_fractb)
begin
        if s_addop='0' then
                s_fract_o <= s_fracta_i + s_fractb_i;
        else
                if fracta_lt_fractb = '1' then
                        s_fract_o <= s_fracta_i - s_fractb_i;
                else
                        s_fract_o <= s_fractb_i - s_fracta_i;
```

```
                end if;
        end if;
end process;



end rtl;
```

Postnormalization

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_arith.all;

library work;
use work.fpupack.all;

entity postnorm_june20 is
        port(
                        clk_i                   : in std_logic;
                        fract_28_i              : in std_logic_vector(FRAC_WIDTH+4
downto 0);      -- carry(1) & hidden(1) & fraction(23) & guard(1) & round(1) & sticky(1)
                        exp_i                   : in std_logic_vector(EXP_WIDTH-1
downto 0);
                        sign_i                  : in std_logic;
                        postnorm_exprr_set_in:in std_logic;
                        exp_i_rr2:in std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_i_rr2:in std_logic_vector(FRAC_WIDTH +1 downto 0);
                        sign_i_rr2,cmpl_in2: in std_logic;
                        fpu_op_i                : in std_logic;
                        rmode_i                         : in std_logic_vector(1 downto 0);
                        expdiff_postin:in std_logic_vector(EXP_WIDTH-1 downto 0);

                        infa_postin,infb_postin,signa_postin,signb_postin:in std_logic;
                        nan_a_postin,nan_b_postin,nan_in_postin,nan_op_postin:in
std_logic;

                        output_o                : out std_logic_vector(FP_WIDTH-1
downto 0);
                        infa_postout,infb_postout: out std_logic;
                        signa_postout,signb_postout: out std_logic;
                        exp_o_rr2:out std_logic_vector(EXP_WIDTH-1 downto 0);
                        mant_o_rr2:out std_logic_vector(FRAC_WIDTH + 1 downto 0);
      ine_o,sign_o_rr2,cmpl_out2: out std_logic
```

```
                    );
end postnorm_june20;


architecture rtl of postnorm_june20 is

signal s_fract_28_i                     : std_logic_vector(FRAC_WIDTH+4 downto 0);
signal s_exp_i                          : std_logic_vector(EXP_WIDTH-1 downto 0);
signal s_sign_i,signa,signb             : std_logic;
signal s_fpu_op_i                       : std_logic;
signal s_rmode_i                        : std_logic_vector(1 downto 0);
signal s_output_o: std_logic_vector(FP_WIDTH-1 downto 0);
signal s_ine_o                  : std_logic;
signal s_overflow                       : std_logic;

signal s_shr1, s_shr2, s_shl, s_shr1e : std_logic;

signal s_expr1_9, s_expr2_9  : std_logic_vector(EXP_WIDTH downto 0);
signal s_exp_shr1, s_exp_shr2, s_exp_shl : std_logic_vector(EXP_WIDTH-1 downto 0);
signal s_fract_shr1, s_fract_shr2: std_logic_vector(FRAC_WIDTH+4 downto 0);
signal s_fract_shl      : std_logic_vector(FRAC_WIDTH + 4 downto 0);
signal s_zeros  : std_logic_vector(5 downto 0);
signal shl_pos: std_logic_vector(5 downto 0);

signal s_fract_1, s_fract_2     : std_logic_vector(FRAC_WIDTH+4 downto 0);
signal s_exp_1, s_exp_2 : std_logic_vector(EXP_WIDTH-1 downto 0);

signal s_fract_rnd : std_logic_vector(FRAC_WIDTH+4 downto 0);
signal s_roundup : std_logic;
signal s_sticky : std_logic;

signal s_zero_fract : std_logic;
signal s_lost : std_logic;
signal s_infa, s_infb : std_logic;
signal s_nan_in, s_nan_op, s_nan_a, s_nan_b, s_nan_sign : std_logic;
signal cmpl_2_br,cmpl_2_ar,rr_sign2_br,rr_sign2_ar:std_logic;
signal s_exp_rr2_br,s_exp_rr2_ar,s_exptemprr_ar:std_logic_vector(EXP_WIDTH-1
downto 0);
signal s_mant_rr2_final:std_logic_vector(FRAC_WIDTH+2 downto 0);
signal mantrr2_cmpl:std_logic_vector(FRAC_WIDTH+1 downto 0):=(others=>'0');
signal s_mant_rr2_br:std_logic_vector(FRAC_WIDTH+2 downto 0);
signal s_mant_rr2_ar:std_logic_vector(FRAC_WIDTH+3 downto 0);
signal s_mant_rr2_ar_trunc:std_logic_vector(FRAC_WIDTH+1 downto
0):=(others=>'0');

signal s_exp_o_rr2:std_logic_vector(EXP_WIDTH-1 downto 0);
```

```
signal s_mant_o_rr2:std_logic_vector(FRAC_WIDTH + 1 downto 0);
signal s_sign_o_rr2,s_cmpl_out2:std_logic;

signal d1:std_logic_vector(FRAC_WIDTH + 2 downto 0):=(others=>'0');
signal d2:std_logic_vector(FRAC_WIDTH + 3 downto 0):=(others=>'0');

component residualreg is port(sign_rr:in std_logic;exp_rr:in
std_logic_vector(EXP_WIDTH - 1 downto 0);cmpl_in:in std_logic;
mant_rr:in std_logic_vector(FRAC_WIDTH+1 downto 0);sign_rr_out:out
std_logic;exp_rr_out:out std_logic_vector(EXP_WIDTH - 1 downto 0);
cmpl_out:out std_logic;mant_rr_out:out std_logic_vector(FRAC_WIDTH+1 downto 0));
end component residualreg;

component dec_br is port(sel:in std_logic_vector(7 downto 0);en:in std_logic;d:out
std_logic_vector(25 downto 0));
end component dec_br;

component dec_ar is port(sel:in std_logic_vector(7 downto 0);en1,en2:in std_logic;d:out
std_logic_vector(26 downto 0));
end component dec_ar;

signal a:std_logic;  -- to see if expdiff>24

signal sum:std_logic_vector(8 downto 0);
signal a1:std_logic;
signal mask:std_logic_vector(24 downto 0):=(others=>'0');

begin

        -- Input Register

                        s_fract_28_i <= fract_28_i;
                        s_exp_i <= exp_i;
                        s_sign_i <= sign_i;
                        s_fpu_op_i <= fpu_op_i;
                        s_rmode_i <= rmode_i;
                        cmpl_2_br<=cmpl_in2;
                        rr_sign2_br<=sign_i_rr2;
                        s_infa<=infa_postin;
                        s_infb<=infb_postin;

                        s_nan_a<=nan_a_postin;
                        s_nan_b<=nan_b_postin;
                        s_nan_in<=nan_in_postin;
                        s_nan_op<=nan_op_postin;
                        signa<=signa_postin;
```

```
                signb<=signb_postin;

        a<=expdiff_postin(7)or expdiff_postin(6)or
expdiff_postin(5)or(expdiff_postin(4)and expdiff_postin(3) and(expdiff_postin(2) or
expdiff_postin(1) or expdiff_postin(0)));

        --Output Register
        process(clk_i)
        begin
        if falling_edge(clk_i) then
                output_o <= s_output_o;
                infa_postout<=infa_postin;
                infb_postout<=infb_postin;
                signa_postout<=signa_postin;
                signb_postout<=signb_postin;
                ine_o <= s_ine_o;
                exp_o_rr2<=s_exp_o_rr2;
                mant_o_rr2<=s_mant_o_rr2;
                sign_o_rr2<=s_sign_o_rr2;
                cmpl_out2<=s_cmpl_out2;
        end if;
        end process;

        -- check if shifting is needed
        -- stage 1a: right-shift (when necessary)
        s_shr1 <= s_fract_28_i(27);
        s_shr1e <= '1' when s_fract_28_i(26)='1' and or_reduce(s_exp_i)='0' else '0'; --if
exp is zero, and hidden bit=1, then exp=exp+1 ( no need to check s_fract_28_i(27)! )
        s_expr1_9 <= "0"&s_exp_i + "000000001";
  s_fract_shr1 <= shr(s_fract_28_i, "1");
        s_exp_shr1 <= s_expr1_9(7 downto 0);

        -- stage 1b: left-shift (when necessary)
        s_shl <= '1' when s_fract_28_i(27 downto 26)="00" and s_exp_i /= "00000000"
else '0';
        -- count the leading zero's of fraction, needed for left-shift
        s_zeros <= count_l_zeros(s_fract_28_i(26 downto 0));
        --s_expl_9 <= ("0"&s_exp_i) - ("000"&s_zeros);
        shl_pos <= "000000" when s_exp_i="00000001" else s_zeros;

        s_fract_shl <= shl(s_fract_28_i, shl_pos);
        s_exp_shl <= "00000000" when s_exp_i="00000001" else s_exp_i -
("00"&shl_pos);

        s_fract_1<=s_fract_shr1 when(s_shr1='1')else
                s_fract_shl when(s_shl='1')else
```

```
s_fract_28_i;


        s_exp_1<=s_exp_shr1 when(s_shr1='1')else
        s_exp_shl when(s_shl='1')else
        s_exp_i;


        dec1:dec_br port map(expdiff_postin,s_shr1,d1);
        s_mant_rr2_br<=('0' & mant_i_rr2)or d1;



        s_exp_rr2_br<=exp_i_rr2;

        -- round

        s_sticky <='1' when s_fract_1(0)='1' or (s_fract_28_i(0) and s_fract_28_i(27))='1'
else '0'; --check last bit, before and after right-shift

        s_roundup <= s_fract_1(2) and ((s_fract_1(1) or s_sticky)or s_fract_1(3)) when
s_rmode_i="00" else -- round to nearset even
                                                (s_fract_1(2) or s_fract_1(1) or
s_sticky) and (not s_sign_i) when s_rmode_i="10" else -- round up
                                                (s_fract_1(2) or s_fract_1(1) or
s_sticky) and (s_sign_i) when s_rmode_i="11" else -- round down
                                                '0'; -- round to zero(truncate = no
rounding)

        s_fract_rnd <= s_fract_1 + "00000000000000000000000001000" when
s_roundup='1' else s_fract_1;

        -- stage 2: right-shift after rounding (when necessary)
        s_shr2 <= s_fract_rnd(27);
        s_expr2_9 <= ("0"&s_exp_1) + "000000001";
        s_fract_shr2 <= shr(s_fract_rnd , "1");
        s_exp_shr2 <= s_expr2_9(7 downto 0);

        s_fract_2 <= s_fract_shr2 when s_shr2='1' else s_fract_rnd;
        s_exp_2 <= s_exp_shr2 when s_shr2='1' else s_exp_1;

        dec2:dec_ar port map(expdiff_postin,s_shr1,s_shr2,d2);
        s_mant_rr2_ar<=('0' & s_mant_rr2_br)or d2;
--
  s_exptemprr_ar<=conv_std_logic_vector(conv_integer(s_exp_i) -
2*(FRAC_WIDTH+1),8);
```

```
        s_exp_rr2_ar<=s_exptemprr_ar
when((postnorm_exprr_set_in='1')and(cmpl_2_ar='1'))else s_exp_rr2_br;


        s_mant_rr2_ar_trunc<=s_mant_rr2_ar(FRAC_WIDTH+1 downto 0)
when((s_shr1 or s_shr2)='0')else
                        s_mant_rr2_ar(FRAC_WIDTH+1 downto 0) when(((s_shr1 xor
s_shr2)='1')and(a='0'))else

        s_mant_rr2_ar(FRAC_WIDTH+2 downto 1) when(((s_shr1 xor
s_shr2)='1')and(a='1'))else

        s_mant_rr2_ar(FRAC_WIDTH+1 downto 0) when(((s_shr1 and
s_shr2)='1')and(a='0'))else

        s_mant_rr2_ar(FRAC_WIDTH+3 downto 2) when(((s_shr1 and
s_shr2)='1')and(a='1'))else

        s_mant_rr2_ar(FRAC_WIDTH+1 downto 0);
        --------------added

        cmpl_2_ar<=signa xor signb xor s_roundup;
  rr_sign2_ar<=s_sign_i xor s_roundup;
  mantrr2_cmpl<=(not s_mant_rr2_ar_trunc);

----    process(clk_i)
----    begin
----            if rising_edge(clk_i) then
----    if(cmpl_2_ar='0')then
----            s_mant_rr2_final<=s_mant_rr2_ar_trunc;
----
        elsif((s_expdiff_int<25)and(s_shr1='0')and(s_shr2='0')and(cmpl_2_ar='1'))then
----            s_mant_rr2_final<=s_mant_rr2_ar_trunc(FRAC_WIDTH + 1 downto
s_expdiff_int)& mantrr2_cmpl(s_expdiff_int-1 downto 0);
----            elsif((s_expdiff_int<25)and((s_shr1 xor
s_shr2)='1')and(cmpl_2_ar='1'))then
----            s_mant_rr2_final<=s_mant_rr2_ar_trunc(FRAC_WIDTH + 1 downto
s_expdiff_int+1)& mantrr2_cmpl(s_expdiff_int downto 0);
----            elsif((s_expdiff_int<25)and((s_shr1 and
s_shr2)='1')and(cmpl_2_ar='1'))then
----            s_mant_rr2_final<=s_mant_rr2_ar_trunc(FRAC_WIDTH + 1 downto
s_expdiff_int+2)& mantrr2_cmpl(s_expdiff_int+1 downto 0);
----            elsif((s_expdiff_int>25)and(cmpl_2_ar='1'))then
----            s_mant_rr2_final<=mantrr2_cmpl;
----            end if;
```

```
----                end if;
----   end process;

sum<=('0' & expdiff_postin)+("00000000"&s_shr1)+("00000000"&s_shr2);
a1<=(sum(4)and sum(3)and(sum(0) or sum(1)or sum(2)))or sum(8)or sum(7)or sum(6)or
sum(5);

 mask <= "00000000000000000000000000" when ((sum = "000000000")and(a1='0')) else
        "00000000000000000000000001" when ((sum = "000000001")and(a1='0')) else
        "00000000000000000000000011" when ((sum = "000000010")and(a1='0')) else
        "00000000000000000000000111" when ((sum = "000000011")and(a1='0')) else
        "00000000000000000000001111" when ((sum = "000000100")and(a1='0')) else
        "00000000000000000000011111" when ((sum = "000000101")and(a1='0')) else
        "00000000000000000000111111" when ((sum = "000000110")and(a1='0')) else
        "00000000000000000001111111" when ((sum = "000000111")and(a1='0')) else
        "00000000000000000011111111" when ((sum = "000001000")and(a1='0')) else
        "00000000000000000111111111" when ((sum = "000001001")and(a1='0')) else
        "00000000000000001111111111" when ((sum = "000001010")and(a1='0')) else
        "00000000000000011111111111" when ((sum = "000001011")and(a1='0')) else
        "00000000000000111111111111" when ((sum = "000001100")and(a1='0')) else
        "00000000000001111111111111" when ((sum = "000001101")and(a1='0')) else
        "00000000000011111111111111" when ((sum = "000001110")and(a1='0')) else
        "00000000000111111111111111" when ((sum = "000001111")and(a1='0')) else
        "00000000001111111111111111" when ((sum = "000010000")and(a1='0')) else
        "00000000011111111111111111" when ((sum = "000010001")and(a1='0')) else
        "00000001111111111111111111" when ((sum = "000010010")and(a1='0')) else
        "00000011111111111111111111" when ((sum = "000010011")and(a1='0')) else
        "00000111111111111111111111" when ((sum = "000010100")and(a1='0')) else
        "00001111111111111111111111" when ((sum = "000010101")and(a1='0')) else
        "00011111111111111111111111" when ((sum = "000010110")and(a1='0')) else
        "00111111111111111111111111" when ((sum = "000010111")and(a1='0')) else
        "01111111111111111111111111" when ((sum = "000011000")and(a1='0')) else
        "11111111111111111111111111" when (a1='1') else
"11111111111111111111111111";

s_mant_rr2_final<=('0'&(mantrr2_cmpl and
mask)+"00000000000000000000000001")when(cmpl_2_ar='1')else ('0' &
s_mant_rr2_ar_trunc);



          rr2:residualreg port
map(rr_sign2_ar,s_exp_rr2_ar,cmpl_2_ar,s_mant_rr2_final(24 downto
0),s_sign_o_rr2,s_exp_o_rr2,s_cmpl_out2,s_mant_o_rr2);

--      signa<=s_opa_i(31);
```

```
--      signb<=s_opb_i(31);
--
--        s_infa <= '1' when s_opa_i(30 downto 23)="11111111"  else '0';
--        s_infb <= '1' when s_opb_i(30 downto 23)="11111111"  else '0';

--        s_nan_a <= '1' when (s_infa='1' and or_reduce (s_opa_i(22 downto 0))='1') else
'0';
--        s_nan_b <= '1' when (s_infb='1' and or_reduce (s_opb_i(22 downto 0))='1') else
'0';
--        s_nan_in <= '1' when s_nan_a='1' or  s_nan_b='1' else '0';
--        s_nan_op <= '1' when (s_infa and s_infb)='1' and (s_opa_i(31) xor (s_fpu_op_i
xor s_opb_i(31)) )='1' else '0'; -- inf-inf=Nan
--
        s_nan_sign <= s_sign_i when (s_nan_a and s_nan_b)='1' else
                                                        signa when s_nan_a='1' else
                                                        signb;


        -- check if result is inexact;
        s_lost <= or_reduce(s_fract_28_i(2 downto 0)) or  or_reduce(s_fract_1(2 downto
0)) or or_reduce(s_fract_2(2 downto 0));
        s_ine_o <= '1' when (s_lost or s_overflow)='1' and (s_infa or s_infb)='0' else '0';

        s_overflow <='1' when (s_expr1_9(8) or s_expr2_9(8))='1' and (s_infa or
s_infb)='0' else '0';
        s_zero_fract <= '1' when s_zeros=27 and s_fract_28_i(27)='0' else '0'; -- '1' if
fraction result is zero

        process(s_sign_i, s_exp_2, s_fract_2, s_nan_in, s_nan_op, s_nan_sign, s_infa,
s_infb, s_overflow, s_zero_fract)
        begin
                if (s_nan_in or s_nan_op)='1' then
                        s_output_o <= s_nan_sign & QNAN;

                elsif (s_infa or s_infb)='1' or s_overflow='1' then
                                s_output_o <= s_sign_i & INF;
                elsif s_zero_fract='1' then
                                s_output_o <= s_sign_i & ZERO_VECTOR;
                else
                                s_output_o <= s_sign_i & s_exp_2 & s_fract_2(25 downto
3);
                end if;
        end process;

        end rtl;
```

Package – FPU pack

```
library  ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

package fpupack is


        -- Data width of floating-point number. Deafult: 32
        constant FP_WIDTH : integer := 32;

        -- Data width of fraction. Deafult: 23
        constant FRAC_WIDTH : integer := 23;

        -- Data width of exponent. Deafult: 8
        constant EXP_WIDTH : integer := 8;

        --Zero vector
        constant ZERO_VECTOR: std_logic_vector(30 downto 0) :=
"0000000000000000000000000000000";

        -- Infinty FP format
constant INF  : std_logic_vector(30 downto 0) :=
"1111111100000000000000000000000";

        -- QNaN (Quit Not a Number) FP format (without sign bit)
  constant QNAN : std_logic_vector(30 downto 0) :=
"1111111110000000000000000000000";

   -- SNaN (Signaling Not a Number) FP format (without sign bit)
  constant SNAN : std_logic_vector(30 downto 0) :=
"1111111100000000000000000000001";

   -- count the  zeros starting from left
   function count_l_zeros (signal s_vector: std_logic_vector) return std_logic_vector;

   -- count the zeros starting from right
        function count_r_zeros (signal s_vector: std_logic_vector) return
std_logic_vector;

end fpupack;

package body fpupack is

   -- count the  zeros starting from left
function count_l_zeros (signal s_vector: std_logic_vector) return std_logic_vector is
```

```
		variable v_count : std_logic_vector(5 downto 0);
	begin
		v_count := "000000";
		for i in s_vector'range loop
			case s_vector(i) is
				when '0' => v_count := v_count + "000001";
				when others => exit;
			end case;
		end loop;
		return v_count;
	end count_l_zeros;


	-- count the zeros starting from right
function count_r_zeros (signal s_vector: std_logic_vector) return std_logic_vector is
		variable v_count : std_logic_vector(5 downto 0);
	begin
		v_count := "000000";
		for i in 0 to s_vector'length-1 loop
			case s_vector(i) is
				when '0' => v_count := v_count + "000001";
				when others => exit;
			end case;
		end loop;
		return v_count;
	end count_r_zeros;

end fpupack;
```

Testbench for Adder with residual register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_ARITH.all;
use ieee.std_logic_textio.all;
use std.textio.all;

ENTITY fpu_add_test_vhd IS
--port(clk_out: out std_logic);
--Type Text is file of String;
--Type Line is access String;

END fpu_add_test_vhd;
```

```
ARCHITECTURE behavior OF fpu_add_test_vhd IS

	-- Component Declaration for the Unit Under Test (UUT)

	COMPONENT fpu_add
	PORT( clk_i : IN std_logic;
	movrr : IN std_logic;
	opa_i : IN std_logic_vector(31 downto 0);
	opb_i : IN std_logic_vector(31 downto 0);
	fpu_op_i : IN std_logic_vector(2 downto 0);
	rmode_i : IN std_logic_vector(1 downto 0);
	output_o : OUT std_logic_vector(31 downto 0);
	ine_o : OUT std_logic;
	overflow_o : OUT std_logic;
	underflow_o : OUT std_logic;
	inf_o : OUT std_logic;
	zero_o : OUT std_logic;
	qnan_o : OUT std_logic;
	snan_o : OUT std_logic;
	post_in:out std_logic_vector(27 downto 0);
	sign_rr0 : OUT std_logic;sign_rr1 : OUT std_logic; sign_rr2: OUT std_logic;
	cmpl_rr0 :OUT std_logic;cmpl_rr1: OUT std_logic;cmpl_rr2 : OUT
std_logic;ready_o:OUT std_logic;
	exp_rr0:OUT std_logic_vector(7 downto 0);exp_rr1:OUT std_logic_vector(7
downto 0);exp_rr2 : OUT std_logic_vector(7 downto 0);
	mant_rr0 : OUT std_logic_vector(24 downto 0);mant_rr1 : OUT
std_logic_vector(24 downto 0);mant_rr2 : OUT std_logic_vector(24 downto 0));

END COMPONENT;

--Inputs
SIGNAL clk_i :  std_logic := '0';
SIGNAL movrr :  std_logic := '0';
SIGNAL opa_i :  std_logic_vector(31 downto 0) := (others=>'0');
SIGNAL opb_i :  std_logic_vector(31 downto 0) := (others=>'0');
SIGNAL fpu_op_i :  std_logic_vector(2 downto 0) := (others=>'0');
SIGNAL rmode_i :  std_logic_vector(1 downto 0) := (others=>'0');

--Outputs
SIGNAL output_o :  std_logic_vector(31 downto 0);
SIGNAL ine_o :  std_logic;
SIGNAL overflow_o :  std_logic;
SIGNAL underflow_o :  std_logic;
SIGNAL div_zero_o :  std_logic;
SIGNAL inf_o :  std_logic;
```

```
SIGNAL zero_o :  std_logic;
SIGNAL qnan_o :  std_logic;
SIGNAL snan_o :  std_logic;
SIGNAL sign_rr0 :  std_logic;
SIGNAL sign_rr1 :  std_logic;
SIGNAL sign_rr2 :  std_logic;
SIGNAL cmpl_rr0 :  std_logic;
SIGNAL cmpl_rr1 :  std_logic;
SIGNAL cmpl_rr2 :  std_logic;
SIGNAL exp_rr0 :  std_logic_vector(7 downto 0);
SIGNAL exp_rr1 :  std_logic_vector(7 downto 0);
SIGNAL exp_rr2 :  std_logic_vector(7 downto 0);
SIGNAL mant_rr0:std_logic_vector(24 downto 0);
SIGNAL mant_rr1:std_logic_vector(24 downto 0);
SIGNAL mant_rr2:std_logic_vector(24 downto 0);
signal sig,temp_mrr: std_logic := '0';
signal cnt : integer:=0;
signal ready_o: std_logic;
signal post_in: std_logic_vector(27 downto 0);
signal result_in : std_logic_vector(31 downto 0);
signal rr_in : std_logic_vector(31 downto 0);
signal rr_out : std_logic_vector(31 downto 0);
signal err_op,err_rr,err: std_logic:='0';

BEGIN

        -- Instantiate the Unit Under Test (UUT)

uut: fpu_add PORT MAP(
                clk_i => clk_i,
                movrr => movrr,
                opa_i => opa_i,
                opb_i => opb_i,
                fpu_op_i => fpu_op_i,
                rmode_i => rmode_i,
                output_o => output_o,
                ine_o => ine_o,
                overflow_o => overflow_o,
                underflow_o => underflow_o,
                inf_o => inf_o,
                zero_o => zero_o,
                qnan_o => qnan_o,
                snan_o => snan_o,
                post_in => post_in,
                sign_rr0 => sign_rr0,
                sign_rr1 => sign_rr1,
```

```
                sign_rr2 => sign_rr2,
                cmpl_rr0 => cmpl_rr0,
                cmpl_rr1 => cmpl_rr1,
                cmpl_rr2 => cmpl_rr2,
                ready_o => ready_o,
                exp_rr0 => exp_rr0,
                exp_rr1 => exp_rr1,
                exp_rr2 => exp_rr2,
                mant_rr0 => mant_rr0,
                mant_rr1 => mant_rr1,
                mant_rr2 => mant_rr2);

fpu_op_i<= "000";
rmode_i <= "00";

clk_i <= not (clk_i) after 50 ns;

movrr_gen:process(clk_i)
begin
if(falling_edge(clk_i))then
if(cnt/= 5)then
cnt<=cnt+1;
else
cnt<=0;
end if;
end if;
if(rising_edge(clk_i))then
if(cnt=4)then
movrr<='1';
else
movrr<='0';
end if;
if(cnt=5)then
temp_mrr<='1';
else
temp_mrr<='0';
end if;
end if;
end process movrr_gen;


read_proc_ab : process is
file infile : TEXT open read_mode is "testdata.txt";
variable opa_in : std_logic_vector(31 downto 0);
variable opb_in : std_logic_vector(31 downto 0);
variable val:std_logic_vector(127 downto 0);
```

```
variable buf_temp : line;
BEGIN
while not endfile(infile) loop
READLINE(infile,buf_temp);
hread(buf_temp,val);
opa_i<= val(127 downto 96);
opb_i<= val(95 downto 64);
result_in<= val(63 downto 32);
wait for 600 ns;
end loop ;
wait for 600 ns;
end process read_proc_ab;

read_proc_oprr : process is
file infile : TEXT open read_mode is "testdata.txt";
variable val:std_logic_vector(127 downto 0);
variable buf_temp : line;
BEGIN
while not endfile(infile) loop
READLINE(infile,buf_temp);
hread(buf_temp,val);
rr_in <= val(31 downto 0);
wait for 700 ns;
end loop ;
wait for 700 ns;
end process read_proc_oprr;

write_proc : process(temp_mrr)  is
file outfile : text open write_mode is "sum_out.txt";
variable buf_temp : line;
begin
if(rising_edge(temp_mrr))then
hwrite(buf_temp,output_o);
writeline(outfile,buf_temp);

if(output_o/=result_in)then
err_op<='1';
else
err_op<='0';
end if;
end if;
END PROCESS  write_proc;

write_rr :process(temp_mrr) is
file outfile : text open write_mode is "rr_out.txt";
variable buf1_temp,buf2_temp : line;
```

```
begin
if(falling_edge(temp_mrr))then
hwrite(buf1_temp,output_o);
writeline(outfile,buf1_temp);

if(rr_in/=output_o)then
err_rr<='1';
else
err_rr<='0';
end if;
end if;
END PROCESS  write_rr;

err<= err_op or err_rr;

END;
```

FPU – Multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_misc.all;

library work;

use work.fpupack.all;


entity fpu_mult is
   port (
      clk_i,movrr                        : in std_logic;

      -- Input Operands A & B
      opa_i                : in std_logic_vector(FP_WIDTH-1 downto 0);  -- Default:
FP_WIDTH=32
      opb_i          : in std_logic_vector(FP_WIDTH-1 downto 0);

      -- fpu operations (fpu_op_i):
                  -- ========================
                  -- 000 = add,
```

```
                -- 001 = substract,
                -- 010 = multiply,
        fpu_op_i            : in std_logic_vector(2 downto 0);

        -- Rounding Mode:
        -- ==============
        -- 00 = round to nearest even(default),
        -- 01 = round to zero,
        -- 10 = round up,
        -- 11 = round down
        rmode_i             : in std_logic_vector(1 downto 0);

        -- Output port
        output_o        : out std_logic_vector(FP_WIDTH-1 downto 0);
                    sign_rr_out:out std_logic;
                    exp_rr_out:out std_logic_vector(EXP_WIDTH - 1 downto 0);
        cmpl_out:out std_logic;
                    mant_rr_out:out std_logic_vector(FRAC_WIDTH+1 downto 0);

         ready_o            : out std_logic;
                            post_in: out std_logic_vector(47 downto 0);

        -- Exceptions
        ine_o                       : out std_logic; -- inexact
        overflow_o          : out std_logic; -- overflow
        underflow_o         : out std_logic; -- underflow

        inf_o                       : out std_logic; -- infinity
        zero_o                      : out std_logic; -- zero
        qnan_o                      : out std_logic; -- queit Not-a-Number
        snan_o                      : out std_logic -- signaling Not-a-Number

            );
end fpu_mult;

architecture rtl of fpu_mult is


            -- Input/output registers
            signal s_opa_i, s_opb_i : std_logic_vector(FP_WIDTH-1 downto 0);
            signal s_fpu_op_i               : std_logic_vector(2 downto 0);
            signal s_rmode_i : std_logic_vector(1 downto 0);
            signal s_output_o : std_logic_vector(FP_WIDTH-1 downto 0);
      signal s_ine_o, s_overflow_o, s_underflow_o, s_inf_o, s_zero_o, s_qnan_o, s_snan_o :
std_logic;
```

```
        signal cnt : integer;
        signal ready : std_logic;
        signal s_output1 : std_logic_vector(FP_WIDTH-1 downto 0);
        signal s_infa, s_infb : std_logic;

        --      ***Multiply units signals***

        signal pre_norm_mul_exp_10 : std_logic_vector(9 downto 0);
        signal pre_norm_mul_fracta_24       : std_logic_vector(23 downto 0);
        signal pre_norm_mul_fractb_24       : std_logic_vector(23 downto 0);

        signal mul_fract_48,s_postnorm_fract_in: std_logic_vector(47 downto 0);
        signal mul_sign,s_postnorm_sign_in: std_logic;

        signal post_norm_mul_output         : std_logic_vector(31 downto 0);
        signal post_norm_mul_ine    : std_logic;

        signal
s_expa_pretomultin,s_expb_pretomultin,s_expa_pretomultout,s_expb_pretomultout:std_l
ogic_vector(EXP_WIDTH-1 downto 0);
        signal s_exp_10_pretomultout,s_postnorm_exp_in:
std_logic_vector(EXP_WIDTH+1 downto 0);

        signal
s_sign_pretomultin,s_op_0_pretomultin,s_fracta0_pretomultin,s_fractb0_pretomultin:
std_logic;
        signal s_op_0_multopostin,s_fracta0_multopostin,s_fractb0_multopostin:
std_logic;
 ----- components -----

 signal s_sign_rr_out,s_cmpl_out:std_logic;
 signal s_exp_rr_out:std_logic_vector(EXP_WIDTH - 1 downto 0);
 signal s_mant_rr_out:std_logic_vector(FRAC_WIDTH+1 downto 0);


        component pre_norm_mul is
        port(
                        clk_i           : in std_logic;
                        opa_i                   : in std_logic_vector(FP_WIDTH-1 downto
0);
                        opb_i                   : in std_logic_vector(FP_WIDTH-1 downto
0);
                        exp_10_o                        : out
std_logic_vector(EXP_WIDTH+1 downto 0);
                        fracta_24_o             : out std_logic_vector(FRAC_WIDTH
downto 0);      -- hidden(1) & fraction(23)
```

```
                fractb_24_o         : out std_logic_vector(FRAC_WIDTH
downto 0);
                expa_o,expb_o: out std_logic_vector(EXP_WIDTH-1 downto 0);
                sign_o,op_0,fracta0,fractb0:out std_logic
        );
    end component;

    component mul_24 is
    port(
                clk_i                   : in std_logic;
                fracta_i                : in std_logic_vector(FRAC_WIDTH
downto 0); -- hidden(1) & fraction(23)
                fractb_i                : in std_logic_vector(FRAC_WIDTH
downto 0);
                expa_pretomultin                    : in
std_logic_vector(EXP_WIDTH-1 downto 0);
                expb_pretomultin                    : in
std_logic_vector(EXP_WIDTH-1 downto 0);
                exp_10_pretomultin          : in
std_logic_vector(EXP_WIDTH+1 downto 0);

sign_pretomultin,op_0_pretomultin,fracta0_pretomultin,fractb0_pretomultin:in std_logic;
                fract_o                 : out
std_logic_vector(2*FRAC_WIDTH+1 downto 0);
                sign_pretomultout           : out std_logic;
                expa_pretomultout                   : out
std_logic_vector(EXP_WIDTH-1 downto 0);
                expb_pretomultout                   : out
std_logic_vector(EXP_WIDTH-1 downto 0);
                op_0_pretomultout,fracta0_pretomultout,fractb0_pretomultout:out
std_logic;
                exp_10_pretomultout         : out
std_logic_vector(EXP_WIDTH+1 downto 0)

                );
    end component;

    component post_norm_mul is
    port(
                clk_i                   : in std_logic;
                expa_multopostin    : in std_logic_vector(EXP_WIDTH-1
downto 0);
                expb_multopostin    : in std_logic_vector(EXP_WIDTH-1
downto 0);
                exp_10_i                : in
std_logic_vector(EXP_WIDTH+1 downto 0);
```

```
                        fract_48_i                      : in
std_logic_vector(2*FRAC_WIDTH+1 downto 0);   -- hidden(1) & fraction(23)
                        sign_i                          : in std_logic;
                        rmode_i                           : in std_logic_vector(1 downto 0);
                        op_0_multopostin,fracta0_multopostin,fractb0_multopostin: in
std_logic;
                        output_o        : out std_logic_vector(FP_WIDTH-1 downto 0);
                        ine_o: out std_logic ;
----------------------------------------------------------------------
        sign_rr_out,cmpl_out:out std_logic;
        exp_rr_out:out std_logic_vector(EXP_WIDTH - 1 downto 0);
        mant_rr_out:out std_logic_vector(FRAC_WIDTH+1 downto 0)
                );
        end component;


begin

        --***Multiply units***

        i_pre_norm_mul: pre_norm_mul
        port map(
                clk_i => clk_i,
                opa_i => s_opa_i,
                opb_i => s_opb_i,
                exp_10_o => pre_norm_mul_exp_10,
                fracta_24_o     => pre_norm_mul_fracta_24,
                fractb_24_o     => pre_norm_mul_fractb_24,
                expa_o => s_expa_pretomultin,
                expb_o => s_expb_pretomultin,
                sign_o => s_sign_pretomultin,
                op_0 => s_op_0_pretomultin,
                fracta0 => s_fracta0_pretomultin,
                fractb0 => s_fractb0_pretomultin
                );

        i_mul_24 : mul_24
        port map(
                        clk_i => clk_i,
                        fracta_i => pre_norm_mul_fracta_24,
                        fractb_i => pre_norm_mul_fractb_24,
                        expa_pretomultin => s_expa_pretomultin,
                        expb_pretomultin => s_expb_pretomultin,
                        exp_10_pretomultin =>pre_norm_mul_exp_10,
                        sign_pretomultin => s_sign_pretomultin,
                        op_0_pretomultin => s_op_0_pretomultin,
```

```
                        fracta0_pretomultin => s_fracta0_pretomultin,
                        fractb0_pretomultin => s_fractb0_pretomultin,
                        fract_o => mul_fract_48,
                        sign_pretomultout =>mul_sign,
                        expa_pretomultout => s_expa_pretomultout,
                        expb_pretomultout => s_expb_pretomultout,
                        op_0_pretomultout => s_op_0_multopostin,
                        fracta0_pretomultout => s_fracta0_multopostin,
                        fractb0_pretomultout => s_fractb0_multopostin,
                        exp_10_pretomultout => s_exp_10_pretomultout
                        );

        i_post_norm_mul : post_norm_mul
        port map(
                        clk_i => clk_i,
                        expa_multopostin => s_expa_pretomultout,
                        expb_multopostin => s_expb_pretomultout,
                        --exp_10_i => s_exp_10_pretomultout,
                        exp_10_i => s_postnorm_exp_in,
                        --fract_48_i    => mul_fract_48,
                        fract_48_i      =>s_postnorm_fract_in,
                        --sign_i        => mul_sign,
                        sign_i => s_postnorm_sign_in,
                        rmode_i => s_rmode_i,
                        op_0_multopostin => s_op_0_multopostin,
                        fracta0_multopostin => s_fracta0_multopostin,
                        fractb0_multopostin => s_fractb0_multopostin,
                        output_o => post_norm_mul_output,
                        ine_o => post_norm_mul_ine ,
                        sign_rr_out => s_sign_rr_out,
                        exp_rr_out => s_exp_rr_out,
        cmpl_out => s_cmpl_out,
                        mant_rr_out => s_mant_rr_out
                       );
          s_postnorm_fract_in<=(s_mant_rr_out & "00000000000000000000000")when
(movrr='1')else mul_fract_48;
  s_postnorm_exp_in<=("00" & s_exp_rr_out) when (movrr='1')else
s_exp_10_pretomultout;
  s_postnorm_sign_in<=s_sign_rr_out when (movrr='1')else mul_sign;

--  s_postnorm_fract_in<= mul_fract_48;
--   s_postnorm_exp_in<=s_exp_10_pretomultout;
--   s_postnorm_sign_in<=mul_sign;

        ------------------------------------------------------------
```

```
-- Input Register

                s_opa_i <= opa_i;
                s_opb_i <= opb_i;
                s_fpu_op_i <= fpu_op_i;
                s_rmode_i <= rmode_i;

-- Output Register
process(clk_i)
begin
        if rising_edge(clk_i) then
        if(ready = '1')then
                output_o <= s_output_o;
                ine_o <= s_ine_o;
                overflow_o <= s_overflow_o;
                underflow_o <= s_underflow_o;

                inf_o <= s_inf_o;
                zero_o <= s_zero_o;
                qnan_o <= s_qnan_o;
                snan_o <= s_snan_o;

                sign_rr_out<=s_sign_rr_out;
                cmpl_out<=s_cmpl_out;
                exp_rr_out<=s_exp_rr_out;
                mant_rr_out<=s_mant_rr_out;

                post_in<=s_postnorm_fract_in;
        end if;
        end if;
end process;


-- Output Multiplexer
process(clk_i)
begin
        if rising_edge(clk_i) then
         if fpu_op_i="010" then
                        s_output1       <= post_norm_mul_output;
                        s_ine_o                 <= post_norm_mul_ine;

                else
                        s_output1       <= (others => '0');
                        s_ine_o                 <= '0';
                end if;
        end if;
```

```
	end process;



	s_infa <= '1' when s_opa_i(30 downto 23)="11111111"  else '0';
	s_infb <= '1' when s_opb_i(30 downto 23)="11111111"  else '0';


	--In round down: the subtraction of two equal numbers other than zero are always
-0!!!
	process(s_output1, s_rmode_i, s_infa, s_infb, s_qnan_o, s_snan_o, s_zero_o,
s_fpu_op_i, s_opa_i, s_opb_i )
	begin

			if s_rmode_i="00" or ((s_infa or s_infb) or s_qnan_o or
s_snan_o)='1' then --round-to-nearest-even
				s_output_o <= s_output1;
			elsif s_rmode_i="01" and s_output1(30 downto 23)="11111111"
then
				--In round-to-zero: the sum of two non-infinity operands is
never infinity,even if an overflow occures
				s_output_o <= s_output1(31) &
"11111110111111111111111111111111";
			elsif s_rmode_i="10" and s_output1(31 downto 23)="111111111"
then
				--In round-up: the sum of two non-infinity operands is
never negative infinity,even if an overflow occures
				s_output_o <= "11111111011111111111111111111111";
			elsif s_rmode_i="11" then
				--In round-down: a-a= -0
				if (s_fpu_op_i="000" or s_fpu_op_i="001") and
s_zero_o='1' and (s_opa_i(31) or (s_fpu_op_i(0) xor s_opb_i(31)))='1' then
					s_output_o <= "1" & s_output1(30 downto 0);
				--In round-down: the sum of two non-infinity operands is
never postive infinity,even if an overflow occures
				elsif s_output1(31 downto 23)="011111111" then
					s_output_o <=
"01111111011111111111111111111111";
				else
					s_output_o <= s_output1;
				end if;
			else
				s_output_o <= s_output1;
			end if;

	end process;
```

```
        -- Generate Exceptions
        s_underflow_o <= '1' when s_output1(30 downto 23)="00000000" and
s_ine_o='1' else '0';
        s_overflow_o <= '1' when s_output1(30 downto 23)="11111111" and s_ine_o='1'
else '0';

        s_inf_o <= '1' when s_output1(30 downto 23)="11111111" and (s_qnan_o or
s_snan_o)='0' else '0';
        s_zero_o <= '1' when or_reduce(s_output1(30 downto 0))='0' else '0';
        s_qnan_o <= '1' when s_output1(30 downto 0)=QNAN else '0';
   s_snan_o <= '1' when s_opa_i(30 downto 0)=SNAN or s_opb_i(30 downto 0)=SNAN
else '0';

         ----Ready signal to indicate start of valid outputs --
        process(clk_i)
        begin
        if(rising_edge(clk_i))then
        if(cnt/=4)then
        cnt <= cnt + 1;
        else
        cnt <= cnt;
        end if;

        if(cnt=4)then
        ready<='1';
        else
        ready<='0';
        end if;
        end if;
        end process;

  ready_o<=ready;



end rtl;
```

Prenormalization

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
```

```
library work;
use work.fpupack.all;

entity pre_norm_mul is
	port(
			clk_i		: in std_logic;
			opa_i		: in std_logic_vector(FP_WIDTH-1 downto
0);
			opb_i		: in std_logic_vector(FP_WIDTH-1 downto
0);
			exp_10_o		: out
std_logic_vector(EXP_WIDTH+1 downto 0);
			fracta_24_o	: out std_logic_vector(FRAC_WIDTH
downto 0);	-- hidden(1) & fraction(23)
			fractb_24_o	: out std_logic_vector(FRAC_WIDTH
downto 0);
			expa_o,expb_o: out std_logic_vector(EXP_WIDTH-1 downto 0);
			sign_o,op_0,fracta0,fractb0:out std_logic
		);
end pre_norm_mul;

architecture rtl of pre_norm_mul is

signal s_expa, s_expb : std_logic_vector(EXP_WIDTH-1 downto 0);
signal s_fracta, s_fractb : std_logic_vector(FRAC_WIDTH-1 downto 0);
signal s_exp_10_o, s_expa_in, s_expb_in : std_logic_vector(EXP_WIDTH+1 downto 0);

signal s_opa_dn, s_opb_dn : std_logic;

begin


		s_expa <= opa_i(30 downto 23);
		s_expb <= opb_i(30 downto 23);
		s_fracta <= opa_i(22 downto 0);
		s_fractb <= opb_i(22 downto 0);

	-- Output Register
	process(clk_i)
	begin
		if rising_edge(clk_i) then
			exp_10_o <= s_exp_10_o;
--			opa_o<=opa_i;
--			opb_o<=opb_i;
```

```
                        sign_o <= opa_i(31) xor opb_i(31);

                        expa_o<=opa_i(30 downto 23);
                        expb_o<=opb_i(30 downto 23);
                        fracta_24_o <= not(s_opa_dn) & s_fracta;
            fractb_24_o <= not(s_opb_dn) & s_fractb;
                        ------------------------------signals reqd in postnormalization-----------
-----
                    op_0 <= not(or_reduce(opa_i(30 downto 0)) and or_reduce(opb_i(30
downto 0)));
                    fracta0<= or_reduce (opa_i(22 downto 0));
                    fractb0<= or_reduce (opb_i(22 downto 0));
                  end if;
        end process;

        -- opa or opb is denormalized
        s_opa_dn <= not or_reduce(s_expa);
        s_opb_dn <= not or_reduce(s_expb);


        s_expa_in <= ("00"&s_expa) + ("000000000"&s_opa_dn);
        s_expb_in <= ("00"&s_expb) + ("000000000"&s_opb_dn);

        s_exp_10_o <= s_expa_in + s_expb_in - "0001111111";

 end rtl;
```

Multiplier

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library work;
use work.fpupack.all;

entity mul_24 is
        port(
                        clk_i                   : in std_logic;
                        fracta_i                : in std_logic_vector(FRAC_WIDTH
downto 0); -- hidden(1) & fraction(23)
                        fractb_i                : in std_logic_vector(FRAC_WIDTH
downto 0);
```

```
                        expa_pretomultin                    : in
std_logic_vector(EXP_WIDTH-1 downto 0);
                        expb_pretomultin                    : in
std_logic_vector(EXP_WIDTH-1 downto 0);
                        exp_10_pretomultin          : in
std_logic_vector(EXP_WIDTH+1 downto 0);

sign_pretomultin,op_0_pretomultin,fracta0_pretomultin,fractb0_pretomultin:in std_logic;
                        fract_o                 : out
std_logic_vector(2*FRAC_WIDTH+1 downto 0);
                        sign_pretomultout           : out std_logic;
                        expa_pretomultout                   : out
std_logic_vector(EXP_WIDTH-1 downto 0);
                        expb_pretomultout                   : out
std_logic_vector(EXP_WIDTH-1 downto 0);
                        op_0_pretomultout,fracta0_pretomultout,fractb0_pretomultout:out
std_logic;
                        exp_10_pretomultout         : out
std_logic_vector(EXP_WIDTH+1 downto 0)

                        );
end mul_24;

architecture rtl of mul_24 is



signal s_fracta_i, s_fractb_i : std_logic_vector(FRAC_WIDTH downto 0);
signal s_fract_o: std_logic_vector(2*FRAC_WIDTH+1 downto 0);

signal a_h, a_l, b_h, b_l : std_logic_vector(11 downto 0);
signal a_h_h, a_h_l, b_h_h, b_h_l, a_l_h, a_l_l, b_l_h, b_l_l : std_logic_vector(5 downto
0);

type op_6 is array (7 downto 0) of std_logic_vector(5 downto 0);
type prod_6 is array (3 downto 0) of op_6;

type prod_48 is array (4 downto 0) of std_logic_vector(47 downto 0);
type sum_24 is array (3 downto 0) of std_logic_vector(23 downto 0);

type a is array (3 downto 0) of std_logic_vector(23 downto 0);
type prod_24 is array (3 downto 0) of a;

signal prod : prod_6;
signal sum : sum_24;
signal prod_a_b : prod_48;
```

```
signal prod2 : prod_24;
begin


-- Input Register
                s_fracta_i <= fracta_i;
                s_fractb_i <= fractb_i;

-- Output Register
process(clk_i)
begin
        if rising_edge(clk_i) then
                fract_o <= s_fract_o;
                sign_pretomultout <= sign_pretomultin;
                expa_pretomultout <= expa_pretomultin;
                expb_pretomultout <= expb_pretomultin;
                op_0_pretomultout<=op_0_pretomultin;
                fracta0_pretomultout<=fracta0_pretomultin;
                fractb0_pretomultout<=fractb0_pretomultin;
                exp_10_pretomultout<=exp_10_pretomultin;
        end if;
end process;

--------------------------------------------------------------------------------
--"000000000000"
-- A = A_h x 2^N + A_l , B = B_h x 2^N + B_l
-- A x B = A_hxB_hx2^2N + (A_h xB_l + A_lxB_h)2^N + A_lxB_l
a_h <= s_fracta_i(23 downto 12);
a_l <= s_fracta_i(11 downto 0);
b_h <= s_fractb_i(23 downto 12);
b_l <= s_fractb_i(11 downto 0);



a_h_h <= a_h(11 downto 6);
a_h_l <= a_h(5 downto 0);
b_h_h <= b_h(11 downto 6);
b_h_l <= b_h(5 downto 0);

a_l_h <= a_l(11 downto 6);
a_l_l <= a_l(5 downto 0);
b_l_h <= b_l(11 downto 6);
b_l_l <= b_l(5 downto 0);

--------------------------------------------------------------------------------
prod(0)(0) <= a_h_h; prod(0)(1) <= b_h_h;
```

```
prod(0)(2) <= a_h_h; prod(0)(3) <= b_h_l;
prod(0)(4) <= a_h_l; prod(0)(5) <= b_h_h;
prod(0)(6) <= a_h_l; prod(0)(7) <= b_h_l;


prod(1)(0) <= a_h_h; prod(1)(1) <= b_l_h;
prod(1)(2) <= a_h_h; prod(1)(3) <= b_l_l;
prod(1)(4) <= a_h_l; prod(1)(5) <= b_l_h;
prod(1)(6) <= a_h_l; prod(1)(7) <= b_l_l;

prod(2)(0) <= a_l_h; prod(2)(1) <= b_h_h;
prod(2)(2) <= a_l_h; prod(2)(3) <= b_h_l;
prod(2)(4) <= a_l_l; prod(2)(5) <= b_h_h;
prod(2)(6) <= a_l_l; prod(2)(7) <= b_h_l;

prod(3)(0) <= a_l_h; prod(3)(1) <= b_l_h;
prod(3)(2) <= a_l_h; prod(3)(3) <= b_l_l;
prod(3)(4) <= a_l_l; prod(3)(5) <= b_l_h;
prod(3)(6) <= a_l_l; prod(3)(7) <= b_l_l;

-------------------------------------------------------------------------------
    prod2(0)(0) <= (prod(0)(0)*prod(0)(1))&"000000000000";
                   prod2(0)(1) <= "000000"&(prod(0)(2)*prod(0)(3))&"000000";
                   prod2(0)(2) <= "000000"&(prod(0)(4)*prod(0)(5))&"000000";
                   prod2(0)(3) <= "000000000000"&(prod(0)(6)*prod(0)(7));

                   prod2(1)(0) <= (prod(1)(0)*prod(1)(1))&"000000000000";
                   prod2(1)(1) <= "000000"&(prod(1)(2)*prod(1)(3))&"000000";
                   prod2(1)(2) <= "000000"&(prod(1)(4)*prod(1)(5))&"000000";
                   prod2(1)(3) <= "000000000000"&(prod(1)(6)*prod(1)(7));

                   prod2(2)(0) <= (prod(2)(0)*prod(2)(1))&"000000000000";
                   prod2(2)(1) <= "000000"&(prod(2)(2)*prod(2)(3))&"000000";
                   prod2(2)(2) <= "000000"&(prod(2)(4)*prod(2)(5))&"000000";
                   prod2(2)(3) <= "000000000000"&(prod(2)(6)*prod(2)(7));

                   prod2(3)(0) <= (prod(3)(0)*prod(3)(1))&"000000000000";
                   prod2(3)(1) <= "000000"&(prod(3)(2)*prod(3)(3))&"000000";
                   prod2(3)(2) <= "000000"&(prod(3)(4)*prod(3)(5))&"000000";
                   prod2(3)(3) <= "000000000000"&(prod(3)(6)*prod(3)(7));

-------------------------------------------------------------------------------
 sum(0) <= prod2(0)(0) + prod2(0)(1) + prod2(0)(2) + prod2(0)(3);
 sum(1) <= prod2(1)(0) + prod2(1)(1) + prod2(1)(2) + prod2(1)(3);
 sum(2) <= prod2(2)(0) + prod2(2)(1) + prod2(2)(2) + prod2(2)(3);
 sum(3) <= prod2(3)(0) + prod2(3)(1) + prod2(3)(2) + prod2(3)(3);
```

```
-------------------------------------------------------------------------

-- Last stage


        prod_a_b(0) <= sum(0)&"00000000000000000000000";
        prod_a_b(1) <= "000000000000"&sum(1)&"000000000000";
        prod_a_b(2) <= "000000000000"&sum(2)&"000000000000";
        prod_a_b(3) <= "000000000000000000000000"&sum(3);

        prod_a_b(4) <= prod_a_b(0) + prod_a_b(1) + prod_a_b(2) + prod_a_b(3);
-------------------------------------------------------------------------
        s_fract_o <= prod_a_b(4);

end rtl;
```

Postnormalization

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_arith.all;

library work;
use work.fpupack.all;

entity post_norm_mul is
        port(
                        clk_i                           : in std_logic;
                        expa_multopostin      : in std_logic_vector(EXP_WIDTH-1
downto 0);
                        expb_multopostin      : in std_logic_vector(EXP_WIDTH-1
downto 0);
                        exp_10_i                        : in
std_logic_vector(EXP_WIDTH+1 downto 0);
                        fract_48_i                      : in
std_logic_vector(2*FRAC_WIDTH+1 downto 0);   -- hidden(1) & fraction(23)
                        sign_i                          : in std_logic;
                        rmode_i                           : in std_logic_vector(1 downto 0);
                        op_0_multopostin,fracta0_multopostin,fractb0_multopostin: in
std_logic;
                        output_o        : out std_logic_vector(FP_WIDTH-1 downto 0);
                        ine_o: out std_logic;
-----------------------------------------------------------------
        sign_rr_out,cmpl_out:out std_logic;
```

```
        exp_rr_out:out std_logic_vector(EXP_WIDTH - 1 downto 0);
        mant_rr_out:out std_logic_vector(FRAC_WIDTH+1 downto 0)
                );
end post_norm_mul;

architecture rtl of post_norm_mul is

signal s_expa1, s_expb1: std_logic_vector(EXP_WIDTH-1 downto 0);
signal s_exp_10_i : std_logic_vector(EXP_WIDTH+1 downto 0);

signal s_sign_1: std_logic;
signal s_output_o : std_logic_vector(FP_WIDTH-1 downto 0):=X"00000000";
signal s_ine_o, s_overflow : std_logic;
signal s_rmode_1: std_logic_vector(1 downto 0);

signal s_zeros  : std_logic_vector(5 downto 0);
signal s_carry  : std_logic;
signal s_shr2, s_shl2 : std_logic_vector(5 downto 0):="000000";
signal s_expo1 : std_logic_vector(8 downto 0);
signal s_exp_10a, s_exp_10b : std_logic_vector(9 downto 0);
signal s_frac2a: std_logic_vector(47 downto 0);

signal s_sticky, s_guard, s_round : std_logic;
signal s_roundup : std_logic;
signal s_frac_rnd, s_frac3 : std_logic_vector(24 downto 0);
signal s_shr3 : std_logic;
signal s_r_zeros1: std_logic_vector(5 downto 0);
signal s_lost : std_logic;
signal s_op_0 : std_logic;
signal s_expo3, s_expo2b : std_logic_vector(8 downto 0);

signal s_infa, s_infb : std_logic;
signal s_nan_in, s_nan_op, s_nan_a, s_nan_b : std_logic;
------------------------------------------------------------
----pipeline signals-----
signal s_fract_48_1:std_logic_vector(2*FRAC_WIDTH+1 downto 0);
signal s_or_a,s_or_b:std_logic;
------------------------------------------------------------
----residual register component-----
component residualreg is port(sign_rr:in std_logic;exp_rr:in
std_logic_vector(EXP_WIDTH - 1 downto 0);cmpl_in:in std_logic;
mant_rr:in std_logic_vector(FRAC_WIDTH+1 downto 0);sign_rr_out:out
std_logic;exp_rr_out:out std_logic_vector(EXP_WIDTH - 1 downto 0);
cmpl_out:out std_logic;mant_rr_out:out std_logic_vector(FRAC_WIDTH+1 downto 0));
end component;
------------------------------------------------------------
```

```
----residual register signals----
signal s_sign_rr,s_sign_rr_out,s_cmpl_in,s_cmpl_out:std_logic;
signal s_exp_rr,s_exp_rr_out:std_logic_vector(EXP_WIDTH - 1 downto 0);
signal s_mant_rr_br:std_logic_vector(FRAC_WIDTH-1 downto 0);
signal s_mant_rr_ar:std_logic_vector(FRAC_WIDTH downto 0);
signal s_mant_rr_final,s_mant_rr_out:std_logic_vector(FRAC_WIDTH+1 downto 0);
------------------------------------------------------------

signal s_r_zeros2: std_logic_vector(5 downto 0);
signal s_sign_2,s_op_0_2,s_or_a2,s_or_b2:std_logic;

begin

        -- Input Register

                        s_expa1 <= expa_multopostin;
                        s_expb1 <= expb_multopostin;
                        s_exp_10_i <= exp_10_i;
                        s_fract_48_1 <= fract_48_i;
                        s_sign_1 <= sign_i;
                        s_rmode_1 <= rmode_i;
                        s_op_0 <= op_0_multopostin;
                        s_or_a <= fracta0_multopostin;
                        s_or_b <= fractb0_multopostin;

--              -- Output Register
        process(clk_i)
        begin
                if rising_edge(clk_i) then
                        output_o <= s_output_o; --
                        ine_o   <= s_ine_o;
                        sign_rr_out<=s_sign_rr_out;
                        cmpl_out<=s_cmpl_out;
                        exp_rr_out<=s_exp_rr_out;
                        mant_rr_out<=s_mant_rr_out;
                end if;
        end process;

    s_zeros <= count_l_zeros(s_fract_48_1(46 downto 1)) when
(s_fract_48_1(47)='0')else "000000";
          s_r_zeros1 <= count_r_zeros(s_fract_48_1);

          s_exp_10a <= s_exp_10_i + ("000000000"& s_fract_48_1(47));
          s_exp_10b <= s_exp_10_i + ("000000000"& s_fract_48_1(47)) -
("0000"&s_zeros);
```

```
  s_carry <= s_fract_48_1(47);

  process(clk_i)
        variable v_shr1, v_shl1 : std_logic_vector(9 downto 0);
        begin
        if rising_edge(clk_i) then
                if s_exp_10a(9)='1' or s_exp_10a="0000000000" then
                        v_shr1 := "0000000001" - s_exp_10a + ("000000000"&s_carry);
                        v_shl1 := (others =>'0');
                        s_expo1 <= "000000001";
                else
                        if s_exp_10b(9)='1' or s_exp_10b="0000000000" then
                                v_shr1 := (others =>'0');
                                v_shl1 := ("0000"&s_zeros) - s_exp_10a;
                                s_expo1 <= "000000001";
                        elsif s_exp_10b(8)='1' then
                                v_shr1 := (others =>'0');
                                v_shl1 := (others =>'0');
                                s_expo1 <= "011111111";
                        else
                                v_shr1 := ("000000000"&s_carry);
                                v_shl1 := ("0000"&s_zeros);
                                s_expo1 <= s_exp_10b(8 downto 0);
                        end if;
                end if;
                if  v_shr1(6)='1' then --"110000" = 48; maximal shift-right postions
                 s_shr2 <= "111111";
          else
                 s_shr2 <= v_shr1(5 downto 0);
                end if;

                s_shl2 <= v_shl1(5 downto 0);

                end if;
        end process;
-- *** Stage 2 ***

        process(clk_i)
        begin
        if(rising_edge(clk_i))then
        if(s_shr2 /= "000000")then
        s_frac2a <= shr(s_fract_48_1, s_shr2);
        elsif(s_shl2 /= "000000")then
  s_frac2a <= shl(s_fract_48_1, s_shl2);
        else
        s_frac2a <= s_fract_48_1;
```

```
        end if;
        end if;
        end process;




        -- signals if precision was last during the right-shift above
        s_lost <= '1' when (s_shr2+("00000"&s_shr3)) > s_r_zeros1 else '0';
        -- ***Stage 3***
        -- Rounding
        --                                                          23
        --                                                             |
        --                          xx00000000000000000000000grsxxxxxxxxxxxxxxxxxxxxx
        -- guard bit: s_frac2a(23) (LSB of output)
  -- round bit: s_frac2a(22)
        s_guard <= s_frac2a(22);
        s_round <= s_frac2a(21);
        s_sticky <= or_reduce(s_frac2a(20 downto 0)) or s_lost;

        s_roundup <= s_guard and ((s_round or s_sticky)or s_frac2a(23)) when
s_rmode_1="00" else -- round to nearset even
                                (s_guard or s_round or s_sticky) and (not s_sign_1) when
s_rmode_1="10" else -- round up
                                (s_guard or s_round or s_sticky) and (s_sign_1) when
s_rmode_1="11" else -- round down
                                '0'; -- round to zero(truncate = no rounding)


        s_mant_rr_br<= s_frac2a(22 downto 0);    -- before rounding

        process(clk_i)
        begin
        if(rising_edge(clk_i))then
  s_r_zeros2<=s_r_zeros1;
        s_sign_2<=s_sign_1;
        s_op_0_2<=s_op_0;
        s_or_a2<=s_or_a;
        s_or_b2<=s_or_b;
        end if;
        end process;

  s_frac_rnd <= (s_frac2a(47 downto 23)) + "1" when(s_roundup='1')else s_frac2a(47
downto 23);
        s_expo2b <= s_expo1 - "000000001" when s_frac2a(46)='0' else s_expo1;
```

```
  s_shr3 <= s_frac_rnd(24);


        s_frac3 <= ("0"&s_frac_rnd(24 downto 1))when(s_shr3='1' and s_expo2b /=
"011111111")else s_frac_rnd;
        s_expo3 <= s_expo2b + '1' when(s_shr3='1' and s_expo2b /= "011111111")else
s_expo2b;

        s_mant_rr_ar<= (s_frac_rnd(0) & s_mant_rr_br) when(s_shr3='1' and s_expo2b
/= "011111111")else ('0' & s_mant_rr_br);


        ---***Stage 4****
        -- Output



        s_infa <= '1' when s_expa1="11111111"  else '0';
        s_infb <= '1' when s_expb1="11111111"  else '0';

        s_nan_a  <= '1' when (s_infa='1' and s_or_a2='1') else '0';
        s_nan_b  <= '1' when (s_infb='1' and s_or_b2='1') else '0';
        s_nan_in <= '1' when s_nan_a='1' or  s_nan_b='1' else '0';
        s_nan_op <= '1' when (s_infa or s_infb)='1' and s_op_0_2='1' else '0';-- 0 * inf =
nan

        s_overflow <= '1' when s_expo3 = "011111111" and (s_infa or s_infb)='0' else '0';
        s_ine_o    <= '1' when s_op_0_2='0' and (s_lost or s_or_a2 or s_overflow)='1' else
'0';

        process(s_sign_2, s_expo3, s_frac3, s_nan_in, s_nan_op, s_infa, s_infb,
s_overflow, s_r_zeros2)
        begin
                if (s_nan_in or s_nan_op)='1' then
                        s_output_o <= s_sign_2 & QNAN;
                elsif (s_infa or s_infb)='1' or s_overflow='1' then
                                s_output_o <= s_sign_2 & INF;
                elsif s_r_zeros1=48 then
                                s_output_o <= s_sign_2 & ZERO_VECTOR;

                else
                                s_output_o <= s_sign_2 & s_expo3(7 downto 0) &
s_frac3(22 downto 0);
                end if;
        end process;
```

```
        s_sign_rr<= s_sign_1 xor s_roundup;
        s_cmpl_in<= s_roundup;

        s_exp_rr<= conv_std_logic_vector(conv_integer(s_expo3(7 downto 0) -
(FRAC_WIDTH+1)),8);

--residual register added----

s_mant_rr_final<=('0' & not(s_mant_rr_ar))when((s_shr3='1')and(s_cmpl_in='1'))else
            ("00" & not(s_mant_rr_ar(22 downto
0)))when((s_shr3='0')and(s_cmpl_in='1'))else
            ('0' & s_mant_rr_ar);

rreg:residualreg port
map(s_sign_rr,s_exp_rr,s_cmpl_in,s_mant_rr_final,s_sign_rr_out,s_exp_rr_out,s_cmpl_o
ut,s_mant_rr_out);

end rtl;
```

References

1. H. G. Dietz, W. R. Dieter, R. Fisher, and K. Chang, " Floating-point computation with just enough accuracy," *Lecture Notes in Computer Science*, vol. 3991, pp.226 – 233, April 2006.

2. W. R. Dieter, H. G. Dietz, Low-Cost Microarchitectural Support for Improved Floating-point Accuracy, *UK ECE Technical Report #ECE–2006-10-14*, October 2006.

3. T. J. Dekker, A Floating-point Technique for extending the available precision, *Numerische Mathematik,* vol. 18, no. 3, June 1971.

4. D. H. Bailey, High-precision Software Directory. http://crd.lbl.gov/~dhbailey/mpdist/

5. "Basic requirements for a future floating-point arithmetic standard"., *GAMM Fachausschuss on Computer Arithmetic and Scientific Computing.* http://www.math.uni-wuppertal.de/~xsc/gamm-fa/BasicRequ.pdf

6. Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, Katherine Yelick, "The Potential of the Cell processor for scientific computing," *Proceedings of the 3rd conference on Computing frontiers*, pp.9-20,2006

7. Guillaume Da Gracca, David Defour, "Implementation of float-float operators on graphics hardware," http://hal.archives-ouvertes.fr/ccsd-00021443/

8. D. Goldberg. "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, vol. 23, no. 1, Mar 1991.

9. D. H. Bailey, H. Simon, J. Barton, and M. Fouts. "Floating-point arithmetic in future supercomputers". *Int. J. Supercomput. Appl. High Perform. Eng*. vol. 3, no. 3, pp. 86-90, 1989.

10. D. H. Bailey, "High-precision floating-point arithmetic in scientific computation".

    http://crd.lbl.gov/~dhbailey/dhbpapers/high-prec-arith.pdf

11. Bruce Greer, John Harrison, Greg Henry and Wei Li Peter Tang. Scientific computing on the Itanium processor.

12. Julie Langou, Piotr Luszczek, Alfredo Buttari, Julien Langou, Jakub Kurzak and Jack Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy". http://icl.cs.utk.edu/projectsfiles/iter-ref/files/iter-ref.pdf

13. W. Kahan, "On the cost of floating-point computation without extra-precise arithmetic," http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf, Nov 2004.

14. Tech report, W. R. Dieter and H. G. Dietz, "Horseshoes & Hand Grenades". http://aggregate.org/WHITE/sc06accprec.pdf, November 2006.

15. Poster Presentation, Andrew Thall, "Extended-precision floating-point numbers for GPU computation," *ACM SIGGRAPH*, 2006. http://delivery.acm.org/10.1145/1180000/1179682/p52-thall.pdf?key1=1179682&key2=1986636021&coll=GUIDE&dl=GUIDE&CFID=60732742&CFTOKEN=20419339

16. W. Kahan, "Why do we need a floating-point arithmetic standard?" February, 1981. http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf

17. Nicholas J. Higham, "Accuracy and stability of numerical algorithms", Society for Industrial and Applied Mathematics, 2002.

18. *IEEE 754 Standard for floating-point arithmetic*, ANSI/IEEE Std 754-1985 Vol , Issue , 12 Aug 1985.

19. Jidan Al-Eryani, Floating point unit. http://www.opencores.org.uk/projects.cgi/web/fpu100/, August 2006.

20. IBM 700/7000 series. Wikipedia Document. http://en.wikipedia.org/wiki/IBM_705#Data_formats

21. Konrad Zuse, Z3 Computer, http://en.wikipedia.org/wiki/Z3

22. Konrad Zuse, Z4 Computer, http://irb.cs.tu-berlin.de/~zuse/Konrad_Zuse/en/Rechner_Z4.html.

23. William R. Dieter, Akil Kaveti, Henry G. Dietz, Low-Cost Microarchitectural Support for Improved Floating-Point Accuracy, *IEEE Computer Architecture Letters*, Vol. 6, No. 1, 2007.

24. Floating-point formats, http://www.quadibloc.com/comp/cp0201.htm.

25. North Star Computers Inc., *NorthStar Hardware Floating point board FPB-A manual*, 25015B, 1977.

26. Yozo Hida, Xiaoye S. Li and David H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic", *ARITH-15*, Oct. 2000.

27. D. H. Bailey. "A Fortran-90 based multiprecision system," *ACM Transactions on Mathematical Software*, vol. 21, no. 4, pp. 379-387, 1995.

28. K. Briggs. The doubledouble library, 1998. http://keithbriggs.info/software.html

29. Jonathan R. Shewchuk. "Adaptive precision floating-point arithmetic and fast robust geometric predicates". *In Discrete and Computational Geometry*, vol. 18, pp. 305-363, 1997.

Vita

- Date of Birth: 6th Feb, 1984.
- Place of Birth: Hyderabad, Andhra Pradesh, India.
- Bachelor of Engineering, M.V.S.R Engineering College (Affiliated to Osmania University), Nadergul, R.R. District, A.P, India.
- Publications:
    - William R. Dieter, Akil Kaveti, Henry G. Dietz, Low–Cost Microarchitectural Support for Improved Floating-Point Accuracy, IEEE Computer Architecture Letters, Vol. 6, No. 1,2007.
    - Akil Kaveti, N.Lakshmi, G.K. Sandeep, Implementation of Rijndael-AES Crypto-processor in FPGA, IETE Journal, 2005.
- Name: Akil Kaveti.