FPGA Implementation of IEEE 754 Standard Based Arithmetic Unit for Floating Point Numbers

A thesis submitted in partial fulfilment of the requirement for the award of the degree of

MASTER OF TECHNOLOGY

in

VLSI Design & CAD

Submitted By

MAHENDRA KUMAR SONI

Roll No. 60761008

Under the guidance of

Mr. B. K. HEMANT

Project Faculty, ECED

Thapar University

Department of Electronics and Communication Engineering

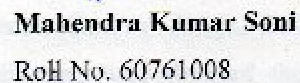
Thapar University, Patiala-147004, India

June, 2009

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"FPGA Implementation of IEEE 754 Standard Based Arithmetic Unit for Floating Point Numbers"** in partial fulfilment of the requirements for the award of the degree of **Master of Technology** in **VLSI Design and CAD** at the **Electronics and Communication Engineering Department** of **Thapar University, Patiala**, is an authentic record of my own work carried out under the supervision of **Mr. B. K. Hemant, Project Faculty**, ECED.

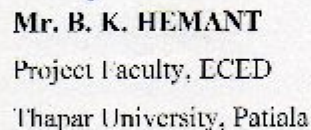
The matter presented in this thesis has not been submitted in any other University/Institute for the award of any degree.



Date: 15-07-09

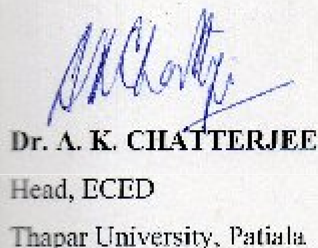
Mahendra Kumar Soni
Roll No. 60761008

It is certified that the above statement made by the student is correct to the best of my knowledge and belief.

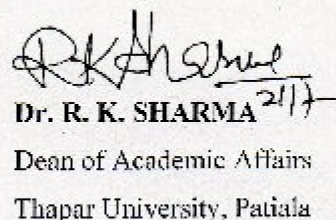


Mr. B. K. HEMANT
Project Faculty, ECED
Thapar University, Patiala

Counter Signed by:



Dr. A. K. CHATTERJEE
Head, ECED
Thapar University, Patiala



Dr. R. K. SHARMA
Dean of Academic Affairs
Thapar University, Patiala

ACKNOWLEDGEMENT

Firstly I would like to express my gratitude to my supervisor, Mr. B. K. Hemant, Project Faculty, ECED, Thapar University, Patiala, for the opportunity to work on my masters thesis under his guidance. He has provided an invaluable help with ideas and discussions throughout my entire time working on this thesis. It was both an honour and a privilege to work with him. He also provided help in technical writing and presentation style and I found this guidance to be extremely valuable.

I would like to thank the entire Electronics and Communication Engineering Department, and specifically, Dr A.K. Chatterjee, Head, and Ms. Alpana Aggarwal, PG Coordinator, for massive support with tools and ideas.

Last but not least I would like to thank my friends, Rashmi Singh, Rohit Sachdeva, Lokesh Kumar Srivastav and Sudhir Kumar Sharma, who devoted their valuable time and helped me in all possible ways towards successful completion of this work. I thank all those who have contributed directly or indirectly to this work.

Mahendra Kumar Soni

ABSTRACT

Arithmetic circuits form an important class of circuits in digital systems. With the remarkable progress in the very large scale integration (VLSI) circuit technology, many complex circuits, unthinkable yesterday have become easily realizable today. Algorithms that seemed impossible to implement now have attractive implementation possibilities for the future. This means that not only the conventional computer arithmetic methods, but also the unconventional ones are worth investigation in new designs.

In this thesis an arithmetic unit based on IEEE standard for floating point numbers has been implemented on Spartan3E FPGA Board. The arithmetic unit implemented has a 32-bit processing unit which allows various arithmetic operations such as, Addition, Subtraction, Multiplication, Division and Square Root, on floating point numbers. Each operation can be selected by a particular operation code. Synthesis of the unit for the FPGA board has been done using XILINX-ISE.

TABLE OF CONTENTS

Certificate i

Acknowledgement ii

Abstract iii

List of figures vi

List of tables vii

1. **Introduction** 1-2
   - 1.1 Overview 1
   - 1.2 Objective 2
   - 1.3 Organization of the thesis 2
2. **Representation Of Floating Point Numbers** 3-15
   - 2.1 Floating point numbers 3
   - 2.2 Floating point format 5
   - 2.3 IEEE 754 standard for binary floating point arithmetic 6
     - 2.3.1 Format 7
       - 2.3.1.1 Single precision 8
       - 2.3.1.2 Double precision 8
     - 2.3.2 Exceptions 10
       - 2.3.2.1 Invalid operation 10
       - 2.3.2.2 Division by zero 11
       - 2.3.2.3 Inexact 11
       - 2.3.2.4 Underflow 11
       - 2.3.2.5 Overflow 12
       - 2.3.2.6 Infinity 12
       - 2.3.2.7 Zero 12
     - 2.3.3 Rounding modes 12
       - 2.3.3.1 Round to nearest even 13
       - 2.3.3.2 Round to zero 13
       - 2.3.3.3 Round up 13

2.3.3.4 Round down 13
2.4 Range of floating point numbers 13
2.4.1 Guard digits 15
3. Design of Arithmetic Unit 16-26
3.1 Introduction 16
3.2 Algorithms for Arithmetic operations 18
3.2.1 Addition and Subtraction 18
3.2.2 Multiplication 21
3.2.3 Division 22
3.2.4 Square root 25
4. Results and Conclusion 27-34
4.1 Simulation results 27
4.1.1 Addition 27
4.1.2 Subtraction 28
4.1.3 Multiplication 29
4.1.4 Division 31
4.1.5 Square root 32
4.2 FPGA implementation 33
4.2.1 Summary of synthesis report 33
4.3 Conclusion 34
4.4 Future scope 34
References 35
Appendix 37

LIST OF FIGURES

Figure 2.1	Single Precision format for Floating Point Numbers	8
Figure 2.2	Bit Double Precision Floating Point Format	9
Figure 2.3	Bit Extended Precision Floating Point Format	10
Figure 3.1	Block Diagram showing the Hardware Implementation of the Arithmetic unit	16
Figure 3.2	Flowchart for floating point addition/subtraction	20
Figure 3.3	Flow chart for floating point multiplication	22
Figure 3.4	Flow chart for floating point division	24
Figure 3.5	Flowchart for floating point square root	25
Figure 3.6	Detailed flowchart for calculating Square root	26
Figure 4.1	Waveforms generated while performing Addition	27
Figure 4.2	Waveforms generated while performing Subtraction	29
Figure 4.3	Waveforms generated while performing Multiplication	30
Figure 4.4	Waveforms generated while performing Division	31
Figure 4.5	Waveforms generated while performing Square Root	32

LIST OF TABLES

Table 2.1 Various basic formats of IEEE 754 standard 7

Table 2.2 Representation of Single Precision floating point numbers 9

Table 2.3 Examples for Round to nearest even 13

Table 4.1 Device utilization summary generated using XILINX ISE 11.1 33

CHAPTER 1
INTRODUCTION

1.1 Overview

Digital arithmetic operations are very important in the design of digital processors and application-specific systems. Arithmetic circuits form an important class of circuits in digital systems. With the remarkable progress in the very large scale integration (VLSI) circuit technology, many complex circuits, unthinkable yesterday have become easily realizable today. Algorithms that seemed impossible to implement now have attractive implementation possibilities for the future. This means that not only the conventional computer arithmetic methods, but also the unconventional ones are worth investigation in new designs.

The notion of real numbers in mathematics is convenient for hand computations and formula manipulations. However, real numbers are not well-suited for general purpose computation, because their numeric representation as a string of digits expressed in, say, base 10 can be very long or even infinitely long. Examples include π, e, and 1/3. In practice, computers store numbers with finite precision. Numbers and arithmetic used in scientific computation should meet a few general criteria:

- Numbers should have modest storage requirements
- Arithmetic operations should be efficient to carry out
- A level of standardization, or portability, is desirable–results obtained on one computer should closely match the results of the same computation on other computers

Internationally-standardized methods for representing numbers on computers have been established by the IEEE 754 standard to satisfy these basic goals [1].

In this thesis an arithmetic unit based on IEEE standard for floating point numbers has been implemented on Spartan3E FPGA Board. The arithmetic unit implemented has a 32-bit processing unit which allows various arithmetic operations such as, Addition, Subtraction, Multiplication, Division and Square Root, on floating point numbers. Each

operation can be selected by a particular operation code. Synthesis of the unit for the FPGA board has been done using XILINX-ISE.

Implemented arithmetic unit is a part of a computer system specially designed to carry out operations on floating point numbers. Some systems (particularly older, microcode-based architectures) can also perform various transcendental functions such as exponential or trigonometric calculations, though in most modern processors these are done with software library routines. In most modern general purpose computer architectures, one or more FPUs are integrated with the CPU; however many embedded processors, especially older designs, do not have hardware support for floating-point operations.

In the past, some systems have implemented floating point via a coprocessor rather than as an integrated unit; in the microcomputer era, this was generally a single microchip, while in older systems it could be an entire circuit board or a cabinet. Not all computer architectures have hardware FPU. In the absence of an FPU, many FPU functions can be emulated.

1.2 Objective

All the modules of the Arithmetic unit has been coded in VHDL with top priority to be able to run at approximately 100-MHz and at the same time as small as possible. Meeting both goals at the same time was very difficult and tradeoffs were made.

1.3 Organization of The Thesis

In **Chapter Two** floating point numbers and their representation in IEEE 754 standard has been described. Various rounding modes and exceptions included in IEEE 754 standard have also been explained.

Chapter Three explains various building blocks of the implemented Arithmetic Unit. It discusses the algorithms used for all arithmetic operations along with the complete working of the unit.

All results and the synthesis report has been included in **Chapter Four**

Finally **Chapter Five** concludes the report and also discusses future aspects.

CHAPTER 2

REPRESENTATION OF FLOATING POINT NUMBERS

2.1 Floating Point Numbers

The term floating point is derived from the fact that there is no fixed number of digits before and after the decimal point, that is, the decimal point can float. There are also representations in which the number of digits before and after the decimal point is set, called fixed-point representations. In general, floating point representations are slower and less accurate than fixed-point representations, but they can handle a larger range of numbers. Floating Point Numbers are numbers that can contain a fractional part. For e.g. following numbers are the floating point numbers: 3.0, -111.5, ½, 3E-5 etc.

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow.[2,3]

A number representation (called a numeral system in mathematics) specifies some way of storing a number that may be encoded as a string of digits. In computing, floating point describes a system for numerical representation in which a string of digits (or bits) represents a rational number. The term floating point refers to the fact that the radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. This position is indicated separately in the internal representation, and floating-point representation can thus be thought of as a computer realization of scientific notation. Over the years, several different floating-point representations have been used in computers; however, for the last ten years the most commonly encountered representation is that defined by the IEEE 754 Standard.

The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values. For example, a fixed-point representation that has seven decimal digits, with the decimal point assumed to be

positioned after the fifth digit, can represent the numbers 12345.67, 8765.43, 123.00, and so on, whereas a floating-point representation (such as the IEEE 754 decimal32 format) with seven decimal digits could in addition represent 1.234567, 123456.7, 0.00001234567, 1234567000000000, and so on. The floating-point format needs slightly more storage (to encode the position of the radix point), so when stored in the same space, floating-point numbers achieve their greater range at the expense of slightly less precision.

There are several mechanisms by which strings of digits can represent numbers. In common mathematical notation, the digit string can be of any length, and the location of the radix point is indicated by placing an explicit "point" character (dot or comma) there. If the radix point is omitted then it is implicitly assumed to lie at the right (least significant) end of the string (that is, the number is an integer). In fixed-point systems, some specific assumption is made about where the radix point is located in the string. For example, the convention could be that the string consists of 8 decimal digits with the decimal point in the middle, so that "00012345" has a value of 1.2345. A floating-point number consists of:[3,4,21]

- A signed digit string of a given length in a given base (or radix). This is known as the significand , or sometimes the mantissa (see below) or coefficient. The radix point is not explicitly included, but is implicitly assumed to always lie in a certain position within the significand—often just after or just before the most significant digit, or to the right of the rightmost digit. This article will generally follow the convention that the radix point is just after the most significant (leftmost) digit. The length of the significand determines the precision to which numbers can be represented.
- A signed integer exponent, also referred to as the characteristic or scale, which modifies the magnitude of the number.

The significand is multiplied by the base raised to the power of the exponent, equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent—to the right if the exponent is positive or to the left if the exponent is negative. Using base-10 (the familiar decimal notation) as an example, the number 152853.5047, which has ten decimal digits of precision, is represented as the significand 1528535047 together with an exponent of 5 (if the implied position of the radix point is

after the first most significant digit, here 1). To recover the actual value, a decimal point is placed after the first digit of the significand and the result is multiplied by 10^5 to give 1.528535047×10^5, or 152853.5047

Symbolically, this final value is

$$\times$$

where *s* is the value of the significand (after taking into account the implied radix point), *b* is the base, and *e* is the exponent.

Equivalently, this is:

$$—— \times$$

where *s* here means the integer value of the entire significand, ignoring any implied decimal point, and *p* is the precision—the number of digits in the significand.

different bases have been used for representing floating-point numbers, with base 2 (binary) being the most common, followed by base 10 (decimal), and other less common varieties

2.2 Floating Point Formats

Several different representations of real numbers have been proposed, but by far the most widely used is the floating-point representation. Floating-point representations have a base b (which is always assumed to be even) and a precision p. If b = 10 and p = 3 then the number 0.1 is represented as 1.00×10^{-1}. If b = 2 and p = 22, then the decimal number 0.1 cannot be represented exactly but is approximately $1.100110011001100110011 \times 2^{-4}$. In general, a floating point number will be represented as $\pm$ d.dd… d $\times b^e$, where d.dd… d is called the Significand and has p digits. More precisely $\pm$ d0 d1 d2 ... $d_{p-1} \times b^e$ represents the number.
The term floating-point number will be used to mean a real number that can be exactly represented in the format under discussion. Two other parameters associated with floating-point representations are the largest and smallest allowable exponents, emax and emin. Since there are b^p possible significands, and emax - emin + 1 possible exponents, a

floating-point number can be encoded in bits, where the final +1 is for the sign bit. The precise encoding is not important for now.

There are two reasons why a real number might not be exactly representable as a floating-point number. The most common situation is illustrated by the decimal number 0.1. Although it has a finite decimal representation, in binary it has an infinite repeating representation. Thus when b = 2, the number 0.1 lies strictly between two floating-point numbers and is exactly representable by neither of them. A less common situation is that a real number is out of range, that is, its absolute value is larger than $b \times b^{emax}$ or smaller than $1.0 \times b^{emin}$.

Floating-point representations are not necessarily unique. For example, both 0.01 × 101 and 1.00 × 10-1 represent 0.1. If the leading digit is nonzero, then the representation is said to be normalized. The floating-point number 1.00×10^{-1} is normalized, while 0.01×10^{1} is not. When b = 2, p = 3, emin = -1 and emax = 2 there are 16 normalized floating-point numbers. The bold hash marks correspond to numbers whose significand is 1.00. Requiring that a floating-point representation be normalized makes the representation unique. Unfortunately, this restriction makes it impossible to represent zero! A natural way to represent 0 is with $1.0 \times bemin^{-1}$, since this preserves the fact that the numerical ordering of nonnegative real numbers corresponds to the lexicographic ordering of their floating-point representations. When the exponent is stored in a k bit field, that means that only 2k - 1 values are available for use as exponents, since one must be reserved to represent 0. Note that the ´ in a floating-point number is part of the notation, and different from a floating-point multiply operation. The meaning of the × symbol should be clear from the context. For example, the expression (2.5 × 10-3) × (4.0 × 102) involves only a single floating-point multiplication.[3-8]

2.3 IEEE 754 Standard for Binary Floating-Point Arithmetic

The IEEE (Institute of Electrical and Electronics Engineers) has produced a Standard to define floating-point representation and arithmetic The standard brought out by the IEEE come to be known as IEEE 754. The IEEE 754 Standard for Floating-Point Arithmetic is the most widely-used standard for floating-point computation, and is followed by many hardware (CPU and FPU) and software implementations. Many computer languages allow or require that some or all arithmetic be carried out using IEEE 754 formats and operations. The current version is IEEE 754-2008, which was published in August 2008;

it includes nearly all of the original IEEE 754-1985 (which was published in 1985) and the IEEE Standard for Radix-Independent Floating-Point Arithmetic (IEEE 854-1987).

The standard specifies :

- Basic and extended floating-point number formats
- Add, subtract, multiply, divide, square root, remainder, and compare operations
- Conversions between integer and floating-point formats
- Conversions between different floating-point formats
- Conversions between basic format floating-point numbers and decimal strings
- Floating-point exceptions and their handling, including non numbers

When it comes to their precision and width in bits, the standard defines two groups: basic and extended format. [3,4,8]

2.3.1 Formats

The standard defines five basic formats, named using their base and the number of bits used to encode them. There are three binary floating-point formats (which can be encoded using 32, 64, or 128 bits) and two decimal floating-point formats (which can be encoded using 64 or 128 bits). The first two binary formats are the 'Single Precision' and 'Double Precision' formats of IEEE 754-1985, and the third is often called 'quad'; the decimal formats are similarly often called 'double' and 'quad'.

Table 2.1: Various basic formats of IEEE 754 standard

parameter → **format name**	***b*** **base**	***p*** **(bits or digits)**	***emax***
binary32	2	23+1 bits	+127
binary64	2	52+1 bits	+1023
binary128	2	112+1 bits	+16383
decimal64	10	16 digits	+384
decimal128	10	34 digits	+6144

2.3.1.1 Single Precision

The most significant bit starts from the left. The three basic components are the sign, exponent, and mantissa. The storage layout for single-precision is show below:

Fig.2.1: Single Precision Format for Floating Point Numbers.

The number represented by the single-precision format is:

Value $=(-1)\ 2\ \times 1.\mathrm{f}$ (normalized) when E > 0 else

$(-1)\ 2\ \times 0.\mathrm{f}$ (denormalized)

Where, f = (+ + +.....+) where =1 or 0

s = sign (0 is positive; 1 is negative)

E =biased exponent; E_{max}=255, E_{min}=0. E=255 and E=0 are used to represent special values.

e =unbiased exponent; e = E – 127(bias)

A bias of 127 is added to the actual exponent to make negative exponents possible without using a sign bit. So for example if the value 100 is stored in the exponent placeholder, the exponent is actually -27 (100 – 127). Not the whole range of E is used to represent numbers. As you may have seen from the above formula, the leading fraction bit before the decimal point is actually implicit (not given) and can be 1 or 0 depending on the exponent and therefore saving one bit. Table 2.2 shows single precision representation with the corresponding values for a given representation to help better understand what was explained above.[3,4,6,8]

2.3.1.2 Double Precision

The double precision format helps overcome the problems of single precision floating point. Using twice the space, the double precision format has an 11-bit excess-1023

exponent and a 53 bit mantissa (with an implied H.O. bit of one) plus a sign bit. This provides a dynamic range of about $10^{\pm 308}$ and 14-1/2 digits of precision, sufficient for most applications. Double precision floating point values take the form shown in Figure2.2.[3]

Table 2.2 Representation of Single Precision floating point numbers

Sign(s)	Exponent(e)	Fraction	Value
0	00000000	00000000000000000000000	+0 (positive zero)
1	00000000	00000000000000000000000	-0 (negative zero)
1	00000000	10000000000000000000000	$-2^{0-127} x0.(2^{-1}) = -2^{0-127} x\ 0.5$
0	00000000	00000000000000000000001	$+2^{0-127} x0.(2^{-23})$ (smallest value)
0	00000001	01000000000000000000000	$+2^{1-127} x1.(2^{-2}) = +2^{1-127} x1.25$
0	10000001	00000000000000000000000	$+2^{129-127} x1.0 = 4$
0	11111111	00000000000000000000000	+ infinity
1	11111111	00000000000000000000000	- infinity
0	11111111	10000000000000000000000	Not a Number(NaN)
1	11111111	10000100010000000001100	Not a Number(NaN)

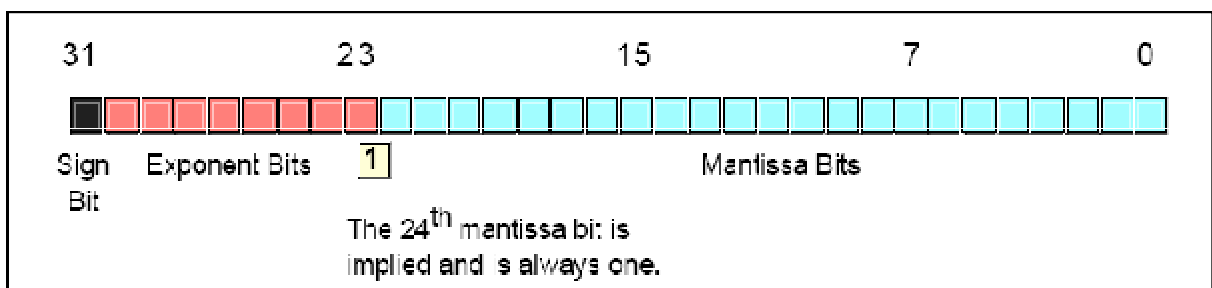


Fig.2.2: Bit Double Precision Floating Point Format

2.3.1.3 Extended Format

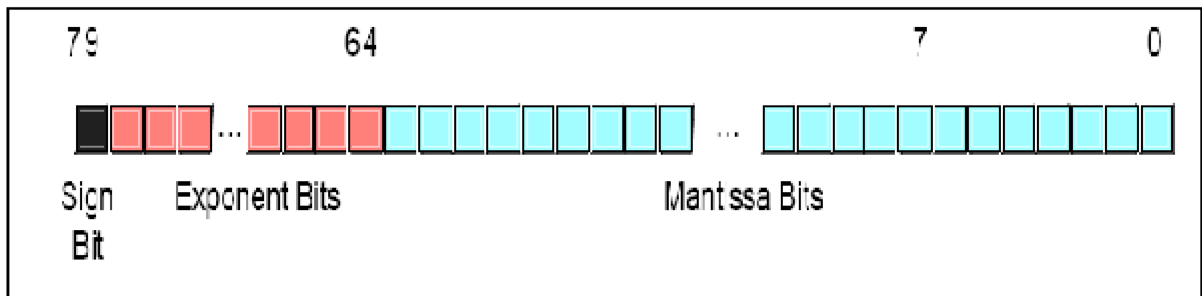


Fig. 2.3: Bit Extended Precision Floating Point Format[3]

In order to help ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa; four of the additional bits are appended to the end of the exponent. Unlike the single and double precision values, the extended precision format does not have an implied H.O. bit which is always one. Therefore, the extended precision format provides a 64 bit mantissa, a 15 bit excess-16383 exponent, and a one bit sign. The format for the extended precision floating point value is shown in Figure 2.3.[3,4,8]

2.3.2 Exceptions

The IEEE standard defines five types of exceptions that should be signalled through a one bit status flag when encountered.

2.3.2.1 Invalid Operation

Some arithmetic operations are invalid, such as a division by zero or square root of a negative number. The result of an invalid operation shall be a NaN (Not a number). There are two types of NaN, quiet NaN (QNaN) and signaling NaN (SNaN). They have the following format, where s is the sign bit:

QNaN = s 11111111 10000000000000000000000

SNaN = s 11111111 00000000000000000000001

The result of every invalid operation shall be a NaN string with a QNaN or SNaN exception. The SNaN string can never be the result of any operation, only the SNaN

exception can be signalled and this happens whenever one of the input operand is a SNaN string otherwise the QNaN exception will be signalled. The SNaN exception can for example be used to signal operations with uninitialized operands, if we set the uninitialized operands to SNaN.

The following are some arithmetic operations which are invalid operations and that give as a result a QNaN string and that signal a QNaN exception:

- Any operation on a NaN
- Addition or subtraction: $\infty + (-\infty)$
- Multiplication: $\pm 0 \times \pm \infty$
- Division: $\pm 0 / \pm 0$ or $\pm \infty / \pm \infty$
- Square root: if the operand is less than zero

2.3.2.2 Division by Zero

In mathematics, a division is called a division by zero if the divisor is zero. Such a division can be formally expressed as a/0 where a is the dividend. Whether this expression can be assigned a well-defined value depends upon the mathematical setting.

In ordinary (real number) arithmetic, the expression has no meaning. In computer programming, integer division by zero may cause a program to terminate or, as in the case of floating point numbers, may result in a special not-a-number value.

The division of any number by zero other than zero itself gives infinity as a result. The addition or multiplication of two numbers may also give infinity as a result. So to differentiate between the two cases, a divide-by-zero exception was implemented.[8,17,21,]

2.3.2.3 Inexact

This exception should be signalled whenever the result of an arithmetic operation is not exact due to the restricted exponent and/or precision range.

2.3.2.4 Underflow

Two events cause the underflow exception to be signalled, tininess and loss of accuracy. Tininess is detected after or before rounding when a result lies between ±2Emin. Loss of accuracy is detected when the result is simply inexact or only when a renormalizations

loss occurs. The implementer has the choice to choose how these events are detected. They should be the same for all operations. The implemented FPU core signals an underflow exception whenever tininess is detected after rounding and at the same time the result is inexact. [5]

2.3.2.5 Overflow

The overflow exception is signalled whenever the result exceeds the maximum value that can be represented due to the restricted exponent range. It is not signalled when one of the operands is infinity, because infinity arithmetic is always exact. Division by zero also doesn't trigger this exception.

2.3.2.6 Infinity

This exception is signalled whenever the result is infinity without regard to how that occurred. This exception is not defined in the standard and was added to detect faster infinity results.

2.3.2.7 Zero

This exception is signalled whenever the result is zero without regard to how that occurred. This exception is not defined in the standard and was added to detect faster zero results.[3,4,8,,21]

2.3.3 Rounding Modes

Since the result precision is not infinite, sometimes rounding is necessary. To increase the precision of the result and to enable round-to-nearest-even rounding mode, three bits were 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-bits 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 no 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, therefore giving a more accurate result.[17,21]

The standard specifies four rounding modes:

2.3.3.1 Round to nearest even

This is the standard default rounding. The value is rounded up or down to the nearest infinitely precise result. If the value is exactly halfway between two infinitely precise results, then it should be rounded up to the nearest infinitely precise even.

Table 2.3: Examples for Round to nearest even

Unrounded	Rounded
3.4	3
5.6	6
3.5	4
2.5	2

2.3.3.2 Round-to-Zero

Basically in this mode the number will not be rounded. The excess bits will simply get truncated, e.g. 3.47 will be truncated to 3.4.

2.3.3.3 Round-Up

In this mode the number will be rounded up towards +∞, e.g. 3.2 will be rounded to 4, while -3.2 to -3.

2.3.3.4 Round-Down

The opposite of round-up, the number will be rounded up towards -∞, e.g. 3,2 will be rounded to 3, while -3,2 to -4.

2.4 Range of Floating Point Numbers

By allowing the radix point to be adjustable, floating-point notation allows calculations over a wide range of magnitudes, using a fixed number of digits, while maintaining good precision. For example, in a decimal floating-point system with three digits, the multiplication that human would write as

$0.12 \times 0.12 = 0.0144$

Would be expressed as

$(1.2 \times 10^{-1}) \times (1.2 \times 10^{-1}) = (1.44 \times 10^{-2})$

In a fixed-point system with the decimal point at the left, it would be

$0.120 \times 0.120 = 0.014$

A digit of the result was lost because of the inability of the digits and decimal point to 'float' relative to each other within the digit string.

The range of floating-point numbers depends on the number of bits or digits used for representation of the significand (the significant digits of the number) and for the exponent. On a typical computer system, a 'double precision' (64-bit) binary floating-point number has a coefficient of 53 bits (one of which is implied), an exponent of 11 bits, and one sign bit. Positive floating-point numbers in this format have an approximate range of 10^{-308} to 10^{308} (because 308 is approximately $1023 \times \log_{10}(2)$, since the range of the exponent is [−1022,1023]). The complete range of the format is from about -10^{308} through $+10^{308}$.

The number of normalized floating point numbers in a system F(B, P, L, U) (where B is the base of the system, P is the precision of the system to P numbers, L is the smallest exponent representable in the system, and U is the largest exponent used in the system) is: 2 * (B - 1) * B^(P-1) * (U - L + 1) + 1. The one is added because the number could be zero. There is a smallest positive normalized floating-point number, Underflow level = UFL = B^L which has a 1 as the leading digit and 0 for the remaining digits of the mantissa, and the smallest possible value for the exponent.

There is a largest floating point number, Overflow level = OFL = B^ (U + 1) * (1 - B^ (-P)) which has B - 1 as the value for each digit of the mantissa and the largest possible value for the exponent. Any number larger than OFL cannot be represented in the given floating-point system, and no number smaller than the UFL can be represented in the floating point system.

The IEEE standard goes further than just requiring the use of a guard digit. It gives an algorithm for addition, subtraction, multiplication, division and square root, and requires

that implementations produce the same result as that algorithm. Thus when a program is moved from one machine to another, the results of the basic operations will be the same in every bit if both machines support the IEEE standard. [3,4,17,21]

2.4.1 Guard Digits

One method of computing the difference between two floating-point numbers is to compute the difference exactly and then round it to the nearest floating point number. This is very expensive if the operands differ greatly in size.

Assuming $p = 3$, $2.15 \times 10^{12} - 1.25 \times 10^{-5}$ would be calculated as

$x = 2.15 \times 10^{12}$

$y = .0000000000000000125 \times 10^{12}$

$x - y = 2.1499999999999999875 \times 10^{12}$

which rounds to 2.15×10^{12}.

Rather than using all these digits, floating-point hardware normally operates on a fixed number of digits. Suppose that the number of digits kept is p, and that when the smaller operand is shifted right, digits are simply discarded (as opposed to rounding). Then

$2.15 \times 10^{12} - 1.25 \times 10^{-5}$ becomes

$x = 2.15 \times 10^{12}$

$y = 0.00 \times 10^{12}$

$x - y = 2.15 \times 10^{12}$

The answer is exactly the same as if the difference had been computed exactly and then rounded.

CHAPTER 3

DESIGN OF ARITHMETIC UNIT

3.1 INTRODUCTION

The block diagram of the designed arithmetic unit showing its hardware implementation has been given in figure 3.1. The arithmetic unit was designed to be as modular as possible. The unit supports five arithmetic operations: Add, Subtract, Multiply, Divide and Square Root.

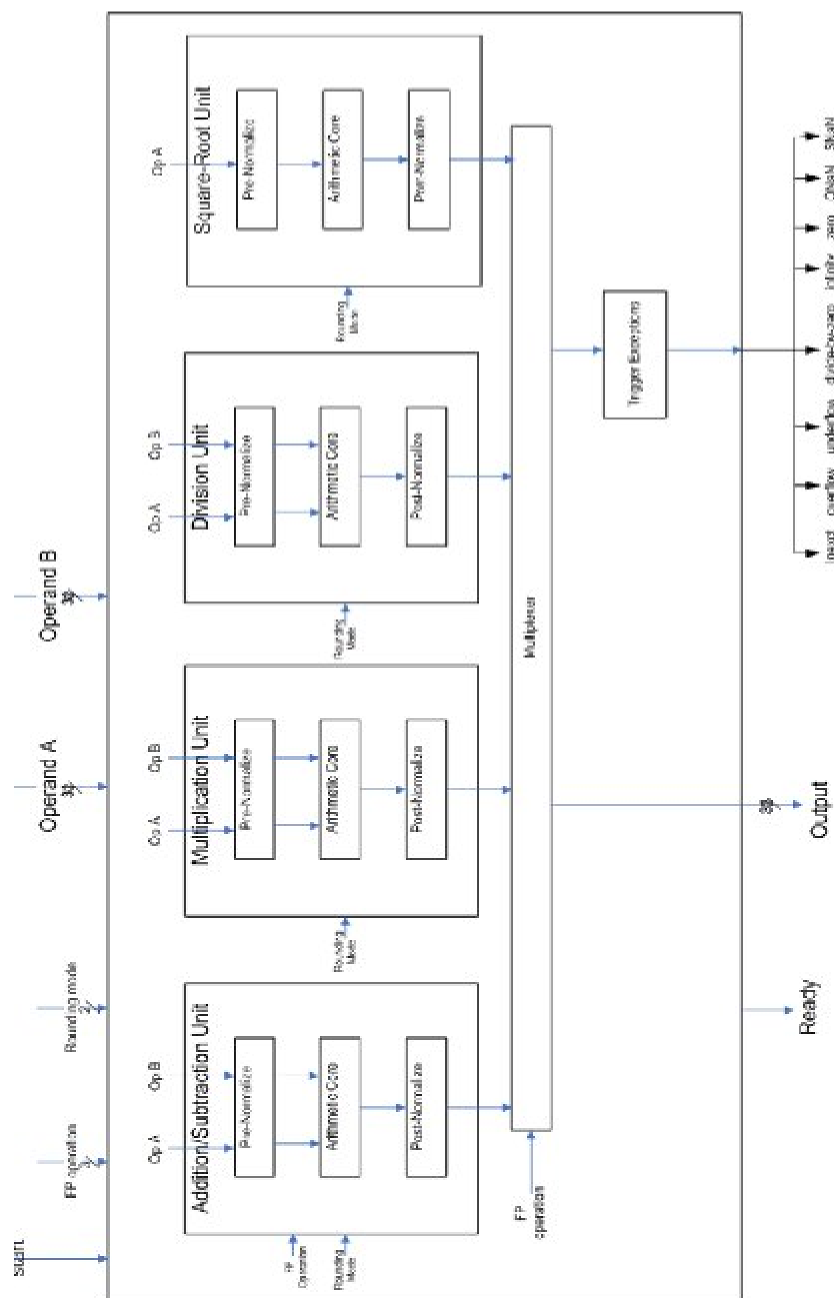


Figure 3.1: Block Diagram showing the Hardware Implementation of the Arithmetic unit.

The unit has following inputs:

- two 32-bit operands in IEEE format
- one 3-bit operation code (op-code)
- one 2-bit code for rounding operation
- and a start bit which also works as reset

and the following outputs:

- 32-bit output in IEEE format
- one ready bit
- and eight Exceptions
    - Inexact
    - Overflow
    - Underflow
    - Divide-by-zero
    - Infinity
    - Zero
    - QNaN
    - SNaN

All arithmetic operations have been carried out in four separate modules one for addition and subtraction and one each for multiplication, division and square root as shown in figure 3.1. In this unit one can select operation to be performed on the 32-bit operands by a 3-bit op-code and the same op-code selects the output from that particular module and connects it to the final output of the unit. Ready bit will be high when result will be available to be taken from the output. Particular exception signal will be high whenever that type of exception will occur as explained in section 2.3.2.

To save logic elements on the chip, one can disable the arithmetic units (modules) that are not needed by modifying the output multiplexer code, since all units are totally independent from each other. It is made modular so that future arithmetic units can be added very easily just by instantiating the unit and connecting its output to the output

multiplexer. Post normalization unit can be shared among modules but it has been made separate for each module to make each module independent and to increase modularity.

All arithmetic operations have these three stages:

- **Pre-normalize:** the operands are transformed into formats that makes them easy and efficient to handle internally.
- **Arithmetic core:** the basic arithmetic operations are done here.
- **Post-normalize:** the result will be normalized if possible (leading bit before decimal point will be 1, if normalized) and then transformed into the format specified by the IEEE standard.

The algorithm used by the arithmetic core to carry out particular operation has been explained in next section.

3.2 Algorithms for Arithmetic Operations

3.2.1 Addition and Subtraction

The conventional floating-point addition algorithm consists of five stages - exponent difference, pre-alignment, addition, normalization and rounding. Given floating-point numbers $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, the stages for computing $X_1 + X_2$ are described as follows:

- Find exponent difference $d = e_1 - e_2$. If e1 < e2, swap position of mantissas. Set larger exponent as tentative exponent of result.
- Pre-align mantissas by shifting smaller mantissa right by d bits.
- Add or subtract mantissas to get tentative result for mantissa.
- Normalization. If there are leading-zeros in the tentative result, shift result left and decrement exponent by the number of leading zeros. If tentative result overflows, shift right and increment exponent by 1-bit.
- Round mantissa result. If it overflows due to rounding, shift right and increment exponent by 1-bit.

The algorithm used for adding or subtracting floating point numbers is shown in the figure 3.2. The pre-alignment and normalization stages require large shifters. The pre-

alignment stage requires a right shifter that is twice the number of mantissa bits (i.e., 48-bits for single-precision, 106-bits for double-precision) because the bits shifted out have to be maintained to generate the guard, round and sticky bits needed for rounding. The shifter only needs to shift right by up to 24 places for single-precision or 53 places for double-precision. The normalization stage requires a left shifter equal to the number of mantissa bits plus 1 (to shift in the guard bit), i.e., 25-bits for single-precision and 54-bits for double precision.

An example is given further to demonstrate the basic steps for adding/subtracting two FP numbers.

Take two 5-digits binary FP numbers:

$2^4 \times 1.1001 + 2^2 \times 1.0010$

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

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

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

   1.1001 000

   \+ 0.0100 100

3) Add both fractions

   1.1001 000

   \+ 0.0100 100

   1.1101 100

4) Round-to-nearest-even

   1.1110

5) Result

   $2^4 \times 1.1110$

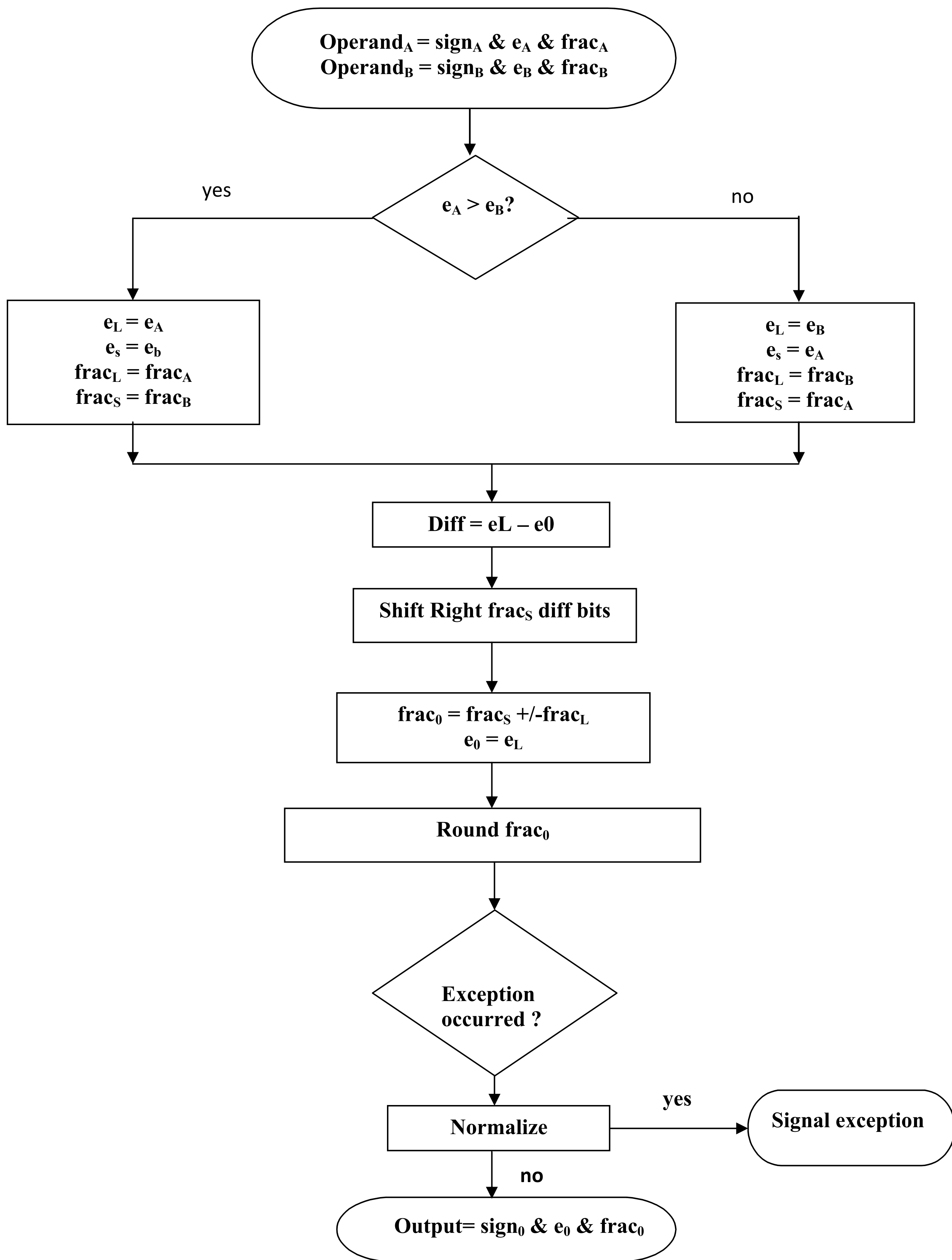


Figure 3.2: Flowchart for floating point addition/subtraction.[11]

3.2.2 Multiplication

In discussing floating-point multiplication, by complies with the IEEE 754 Standard, the two mantissas are to be multiplied, and the two exponents are to be added. In order to perform floating-point multiplication, a simple algorithm is realized:

- Add the exponents and subtract 127 (bias)
- Multiply the mantissas and determine the sign of the result
- Normalize the resulting value, if necessary

Figure 3.3 depicts a generic flow chart for a floating-point multiplier. The multiplication was done parallel to save clock cycles, at the cost of hardware. If done serial it would have taken 32 clock cycles (without pre-, post-normalization) instead of the actual 5 clock cycles needed. Disadvantage, the hardware needed for the parallel 32-bit multiplier is approximately 3 times that of serial.

The multiplication algorithm takes 3 steps. To demonstrate the basic steps, take two 5-digits FP numbers to multiply:

$2^{100} \times 1.1001$

$\times\ 2^{110} \times 1.0010$

---

1) Multiply fractions and calculate the result exponent.

1.1001

$\times\ 1.0010$

1.11000010

So $frac_O = 1.11000010$ and $e_O = 2^{100+110-bias} = 2^{83}$

2) Round the fraction to nearest-even

$frac_O = 1.1100$

3) Result $= 2^{83} \times 1.1100$

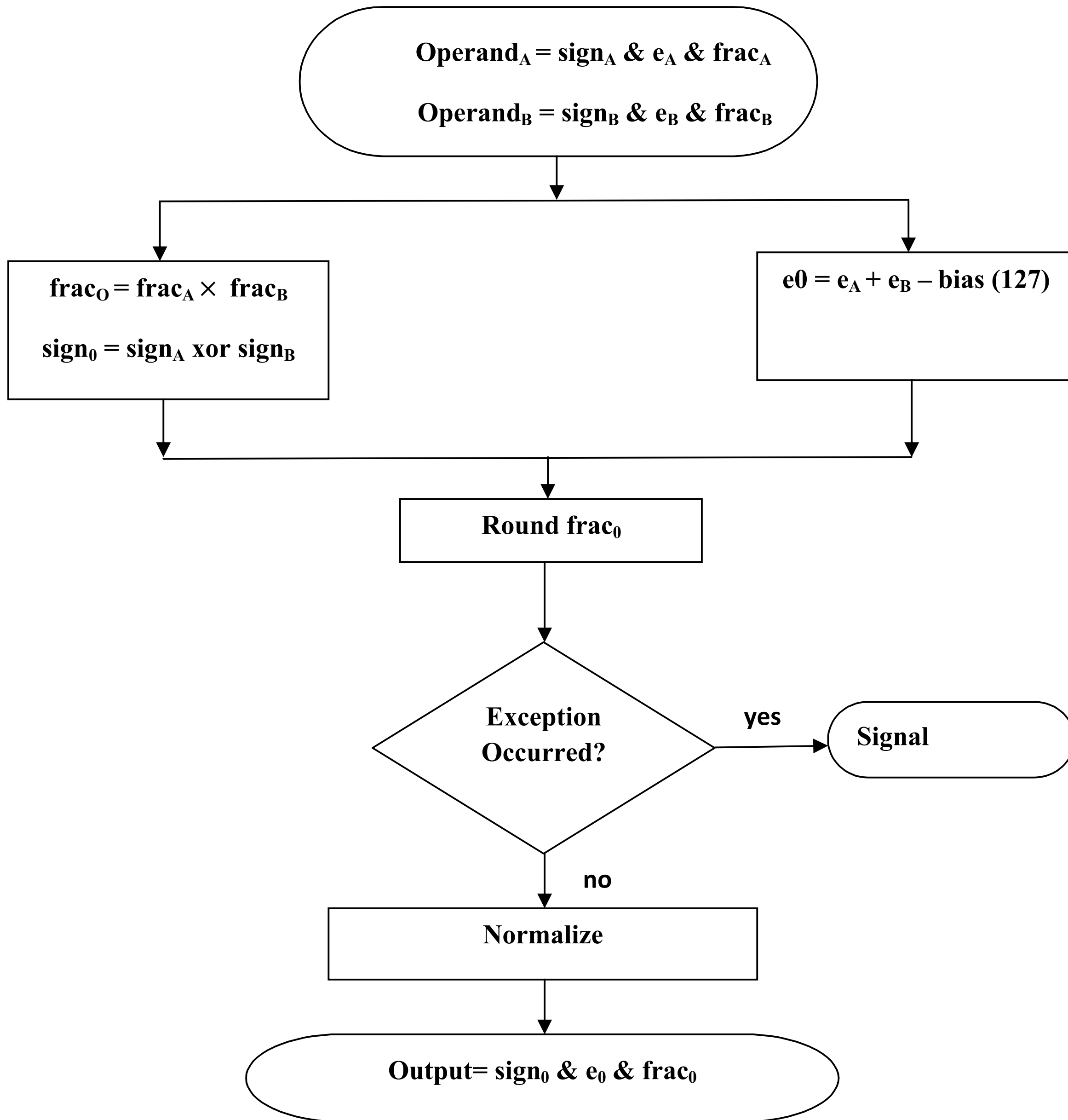


Figure 3.3: Flowchart for floating point multiplication.[11]

3.2.3 Division

The division was done serially using the basic algorithm taught in most schools, which is division through multiple subtractions. Since divisions are not needed as often as multiplications. Divisions can be done also through multiplication. It was implemented as serial and in the process saving some hardware area.

The conventional floating-point division algorithm consists of five stages – counting leading zeroes in both numbers, shifting left, division, rounding and. Normalization.

Figure 3.4 shows the flowchart for floating point multiplication. Given floating-point numbers A = (s_1, e_1, f_1) and B = (s_2, e_2, f_2), the stages for computing A/B are described as follows:

- Count leading zeroes in both floating point numbers.
- Shift left fractional bits of both floating point numbers according to number of zeroes.
- Divide the fractional bits. Sign of result is calculated from exoring sign of two operand.
- Exponent of result is calculated by equation:

$$\mathbf{e_0 = e_A - e_B + bias\ (127) - z_A + z_B}$$

- Round the fraction and normalize it if required.

To demonstrate the basic steps of division, let's say we want to divide two 5-digits FP numbers:

$2^{110} \times 1.0000$

$\div\, 2^{100} \times 0.0011$

---

1) Count leading zeros in both fractions.

   $z_A = 0,\ z_B = 3$

2) Shift-left the fractions according to z_A, z_B. Calculate the result exponent

   $frac_A = 10000\ 00000$

   $frac_B = 00000\ 11000$

   $e_O = 2^{110-100+bias-0+3} = 2^{140}$

3) Divide both fractions

   100000.0000

   ÷ 000001.1000

   1.0101

4) Result = 1.0101×2^{140}

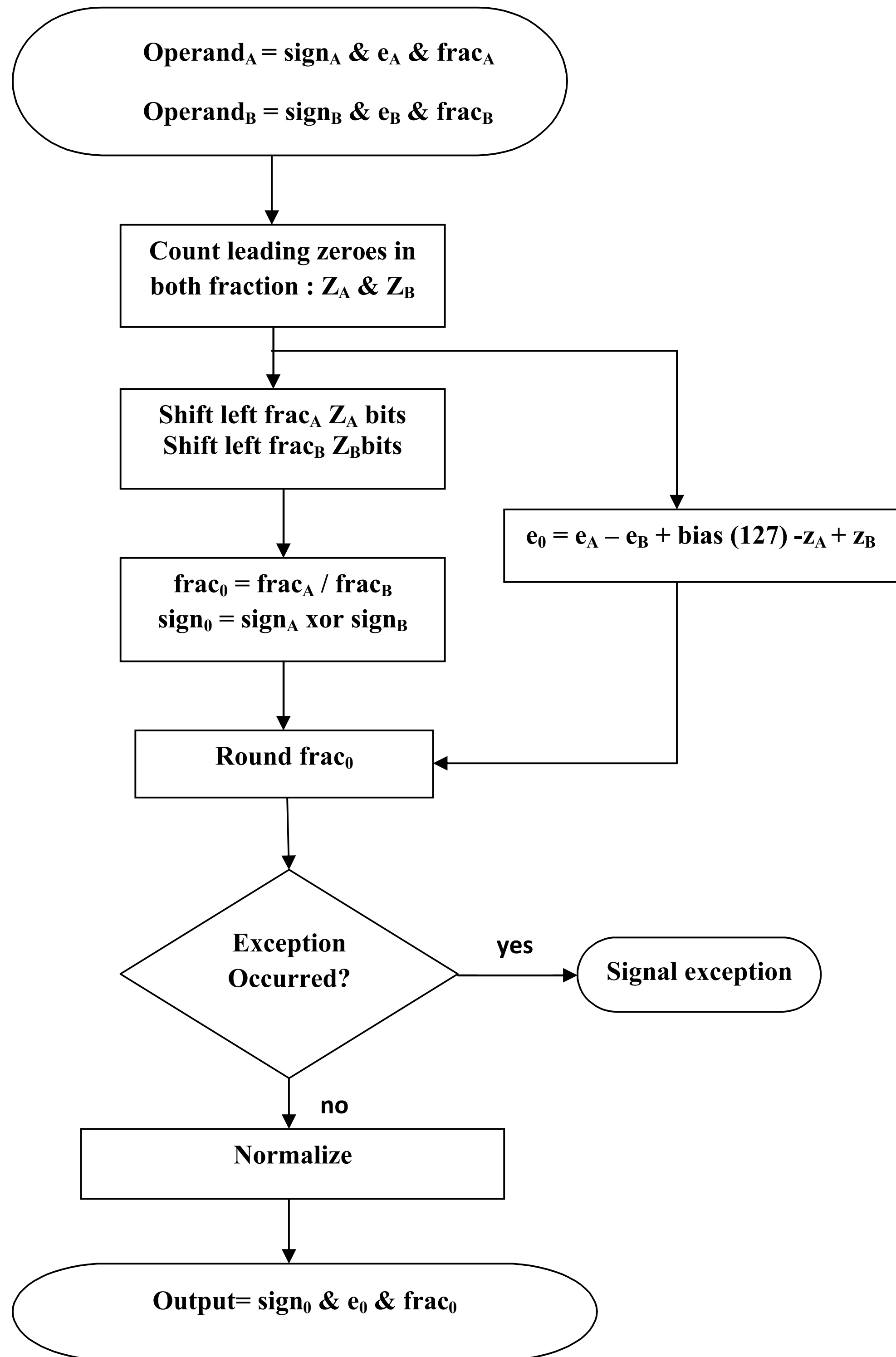


Figure 3.4: Flowchart for floating point Division.[20]

3.2.4 Square root

The function square root is the most important elementary function. Since square rooting is widely used in many applications, and hardware realization of square-rooting has quite in common with division. The IEEE floating point standard specifies square-rooting as a basic arithmetic operation.

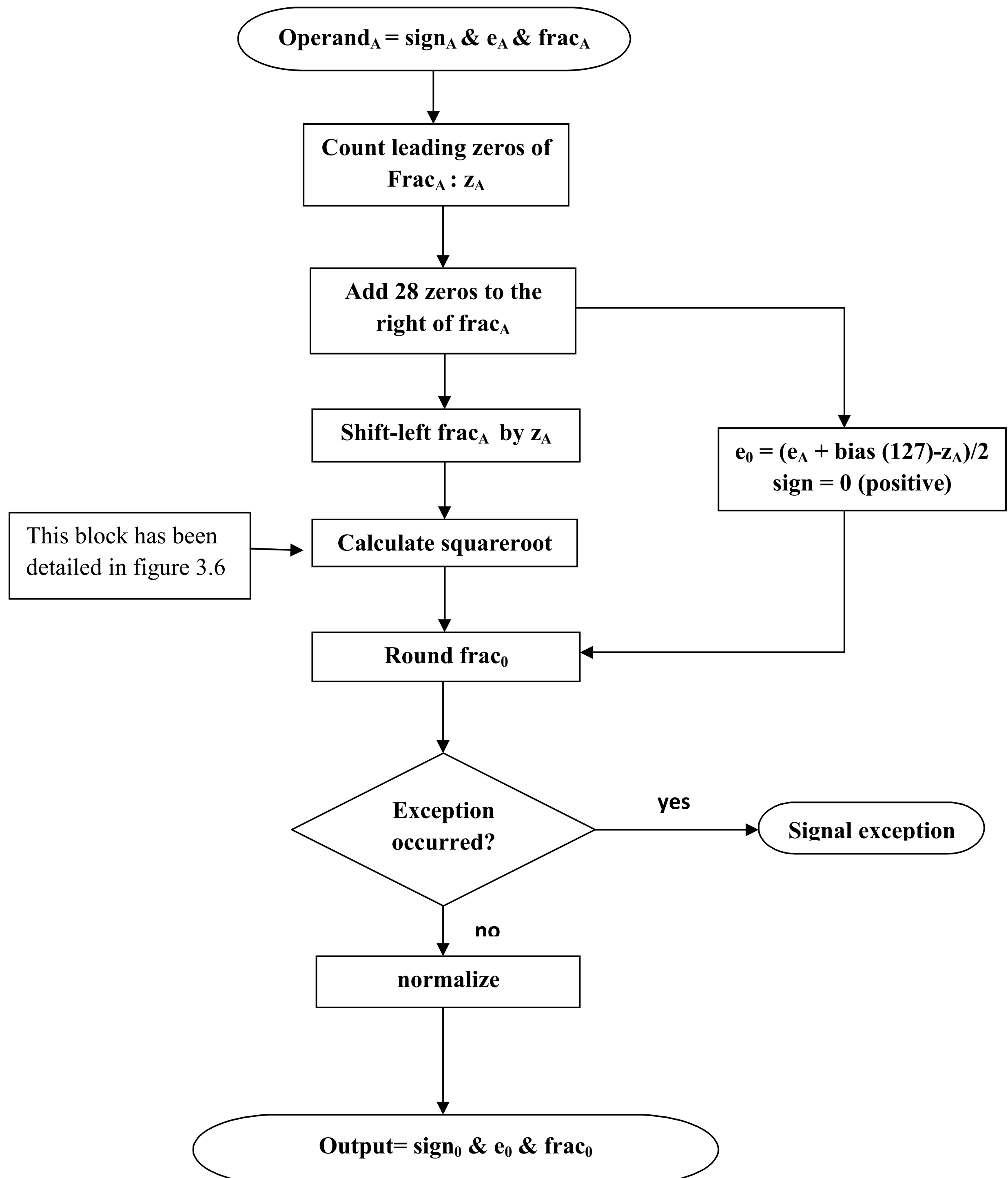


Figure 3.5: Flowchart for floating point Square root.

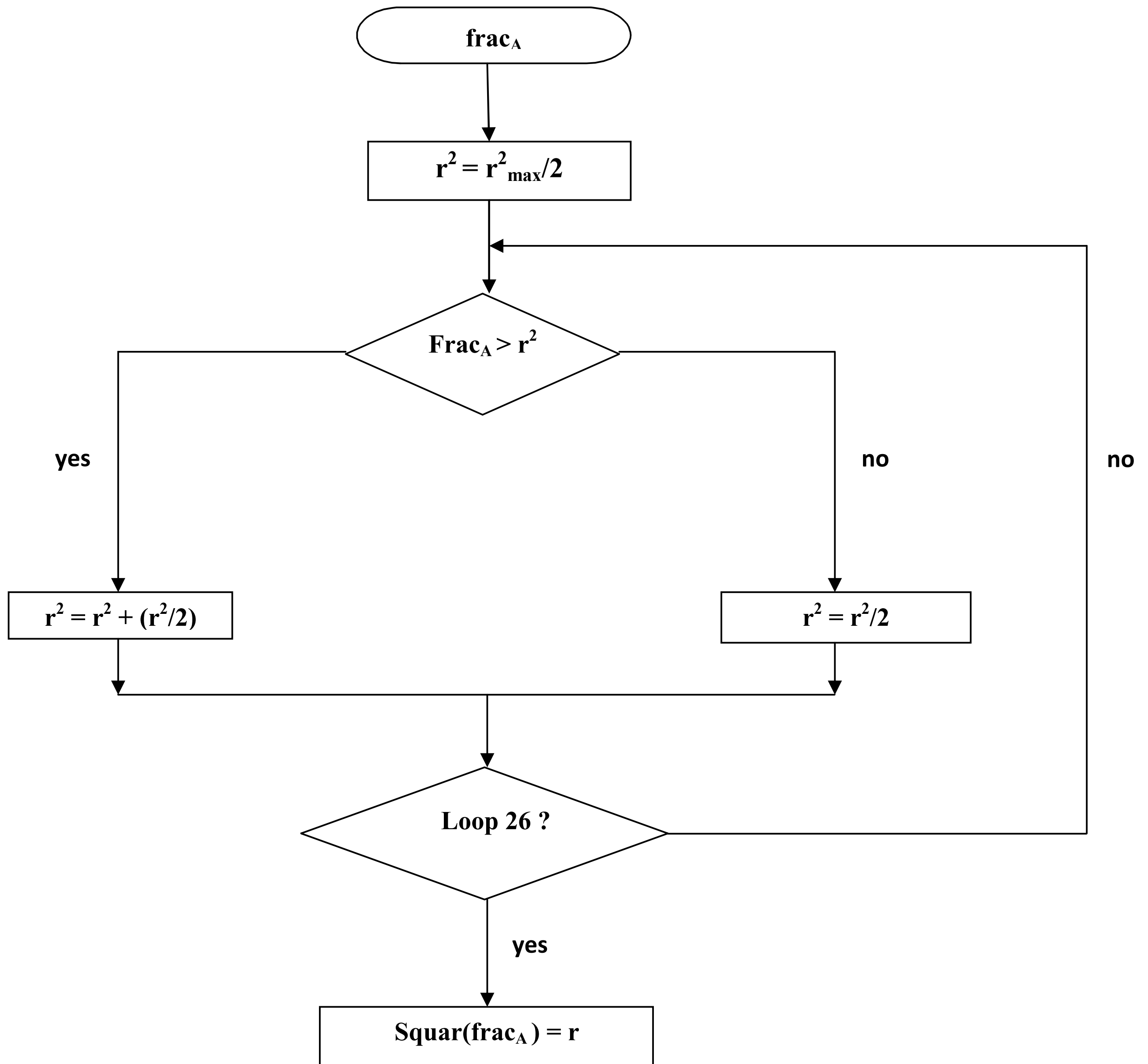


Figure 3.6: Detailed flowchart for calculating Square root.

The square root has been calculated using an iterative algorithm, which needs the same number of loops as the precision of the result. The square-root algorithm used here doesn't need any multipliers or divisors, because all multiplications were replaced with left-shifts and all divisions with right-shifts. This makes the algorithm very efficient and fast for hardware implementations. Figure 3.5 shows the flowchart for the floating point square root. The detailed flowchart for calculating square root (figure 3.5) is given in figure 3.6.[12,19]

CHAPTER 4

RESULTS AND CONCLUSION

4.1 SIMULATION RESULTS

The arithmetic unit explained in last chapter has been coded in VHDL and simulated using modelsim simulater. The simulation results for all arithmetic operations performed by the unit are shown below.

4.1.1 Addition

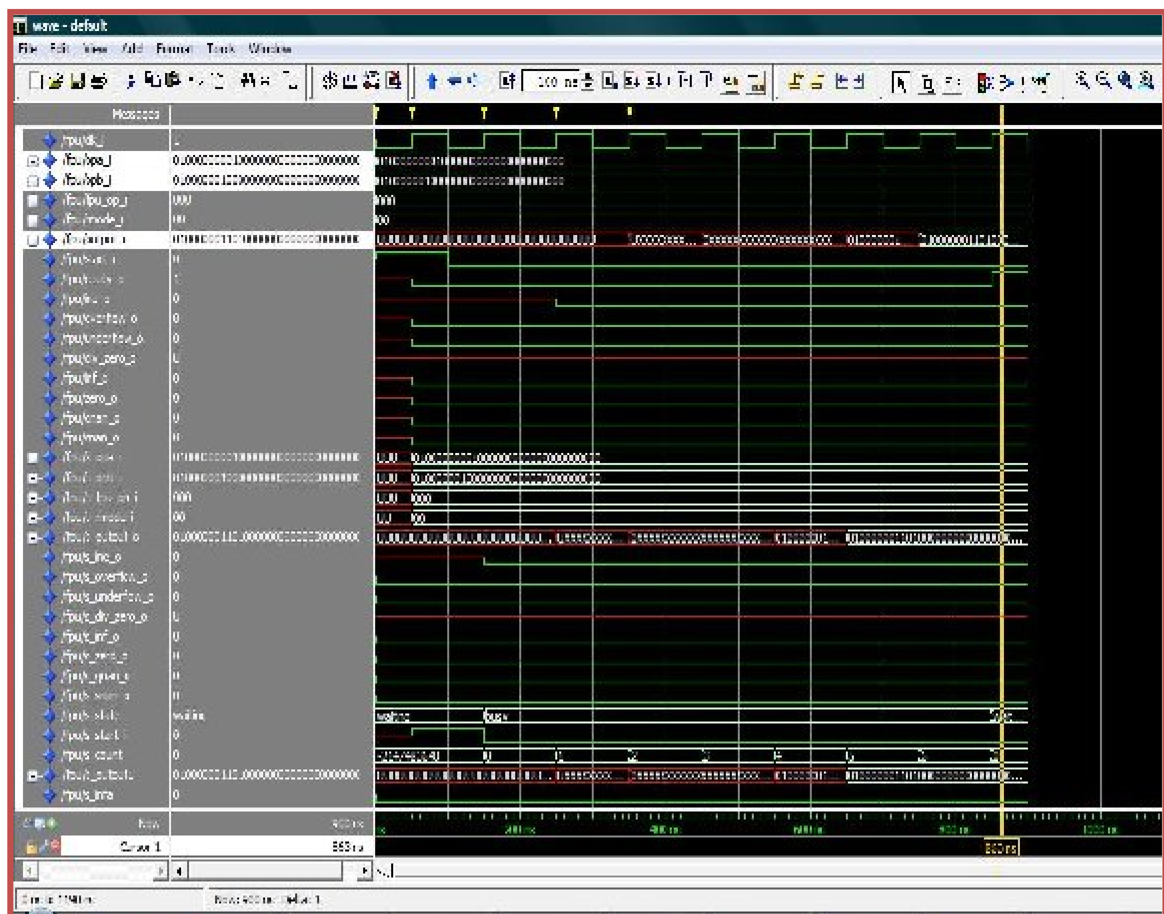


Figure 4.1: Waveforms generated while performing Addition.

Figure 4.1 shows the waveforms generated using modelsim while performing addition. The detailed description of the given inputs and the output generated is given further.

Input operends

Operand A (in base 2) = 01000000001000000000000000000000

So, S_A = 0, E_A = 128 – 127 = 1, f_A =25

Operand A (in base 10) = + $2^1 \times 1.25 = 2.5$

Operand B (in base 2) = 01000000010000000000000000000000

So, S_B = 0, E_B = 129 – 127 = 2, f_B =0

Operand B (in base 10) = + $2^2 \times 1.0 = 4$

Operation Code

000, for addition.

Output

Output (in base 2) = 01000000110100000000000000000000

S_O = 0, e_O = 129 – 127 = 2, f_O = 625

So, Output (in base 10) = + $2^2 \times 1.625 = 6.5$

4.1.2 Subtraction

Figure 4.2 shows the waveforms generated using modelsim while performing subtraction. The detailed description of the given inputs and the output generated is given further.

Input operends

Operand A (in base 2) = 01000000001000000000000000000000

So, S_A = 0, E_A = 128 – 127 = 1, f_A =25

Operand A (in base 10) = + $2^1 \times 1.25 = 2.5$

Operand B (in base 2) = 01000000010000000000000000000000

So, S_B = 0, E_B = 129 – 127 = 2, f_B =0

Operand B (in base 10) = + $2^2 \times 1.0 = 4$

Operation Code

001, for subtraction.

Output

Output (in base 2) = 10111111110000000000000000000000

$S_O = 1$, $e_O = 127 - 127 = 0$, $f_O = 5$

So, Output (in base 10) = $- 2^0 \times 1.5 = -1.5$

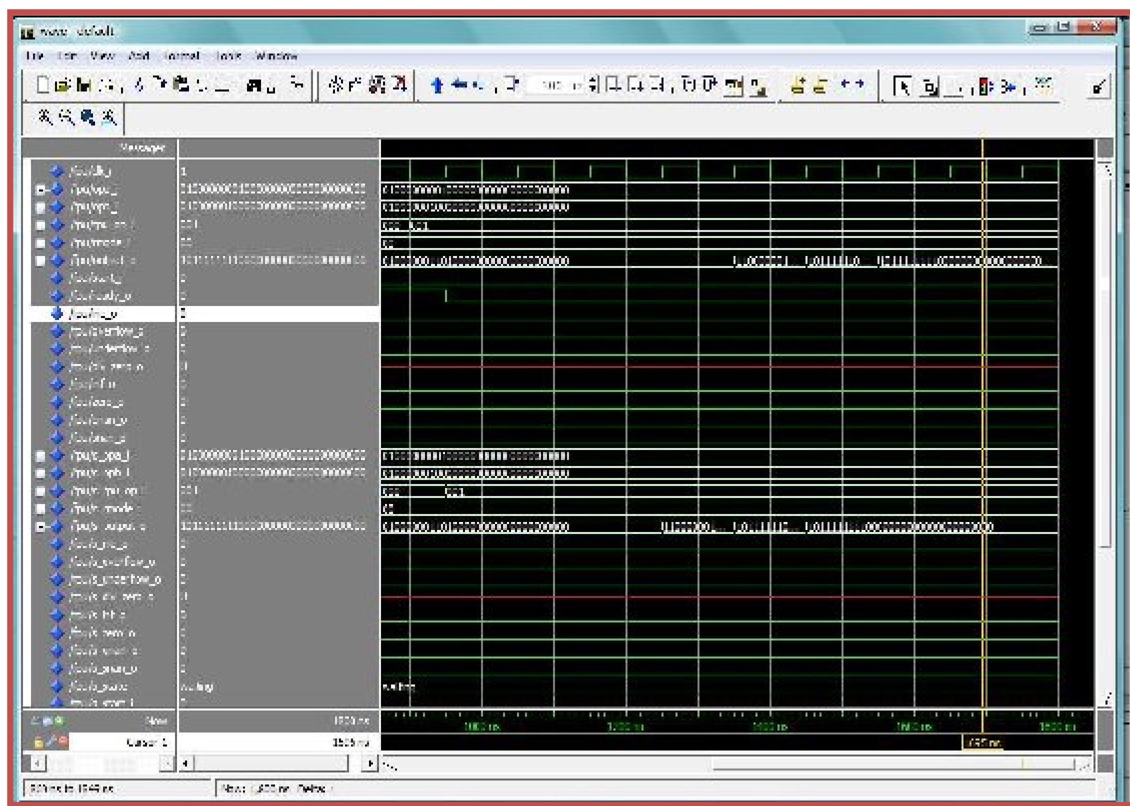


Figure 4.2: Waveforms generated while performing Subtraction.

4.1.3 Multiplication

Figure 4.3 shows the waveforms generated using modelsim while performing multiplication. The detailed description of the given inputs and the output generated is given further.

Input operends

Operand A (in base 2) = 01000000100000000000000000000000

So, $S_A = 0$, $E_A = 129 – 127 = 2$, $f_A = 0$

Operand A (in base 10) = $+ 2^2 \times 1.0 = 4.0$

Operand B (in base 2) = 01000000100000000000000000000000

So, $S_B = 0$, $E_B = 129 – 127 = 2$, $f_B = 0$

Operand B (in base 10) = $+ 2^2 \times 1.0 = 4$

Operation Code

010, for multiplication.

Output

Output (in base 2) = 01000001100000000000000000000000

$S_O = 0$, $e_O = 131 – 127 = 4$, $f_O = 0$

So, Output (in base 10) = $+ 2^4 \times 1.0 = 16.0$

Figure 4.3: Waveforms generated while performing Multiplication.

4.1.4 Division

Figure 4.4 shows the waveforms generated using modelsim while performing division. The detailed description of the given inputs and the output generated is given further.

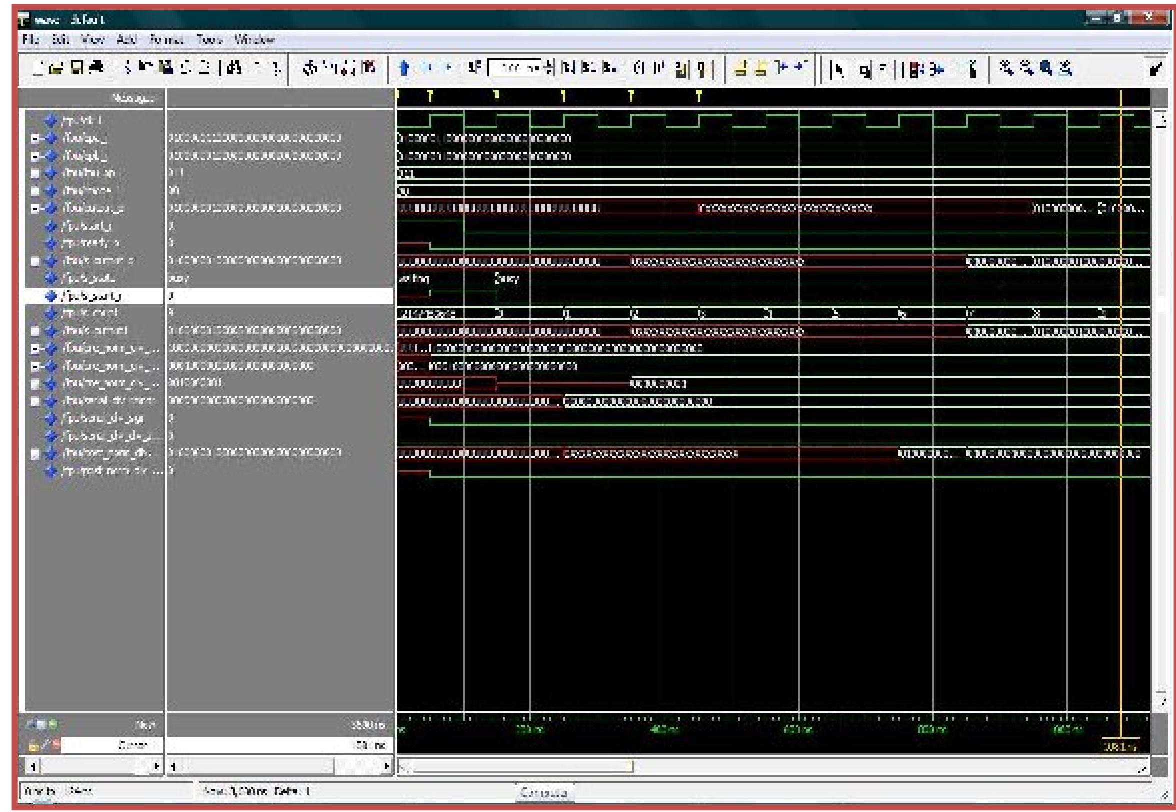


Figure 4.4: Waveforms generated while performing Division.

Input operends

Operand A (in base 2) = 01000001100000000000000000000000

So, $S_A = 0$, $E_A = 131 – 127 = 4$, $f_A = 0$

Operand A (in base 10) = $+ 2^4 \times 1.0 = 16.0$

Operand B (in base 2) = 01000000010000000000000000000000

So, $S_B = 0$, $E_B = 129 – 127 = 2$, $f_B = 0$

Operand B (in base 10) = $+ 2^2 \times 1.0 = 4$

Operation Code

011, for division.

Output

Output (in base 2) = 01000000100000000000000000000000

$S_O = 0$, $e_O = 129 - 127 = 2$, $f_O = 0$

So, Output (in base 10) = $+ 2^2 \times 1.0 = 4.0$

4.1.5 Square Root

Figure 4.5 shows the waveforms generated using modelsim while performing square root. The detailed description of the given inputs and the output generated is given further.

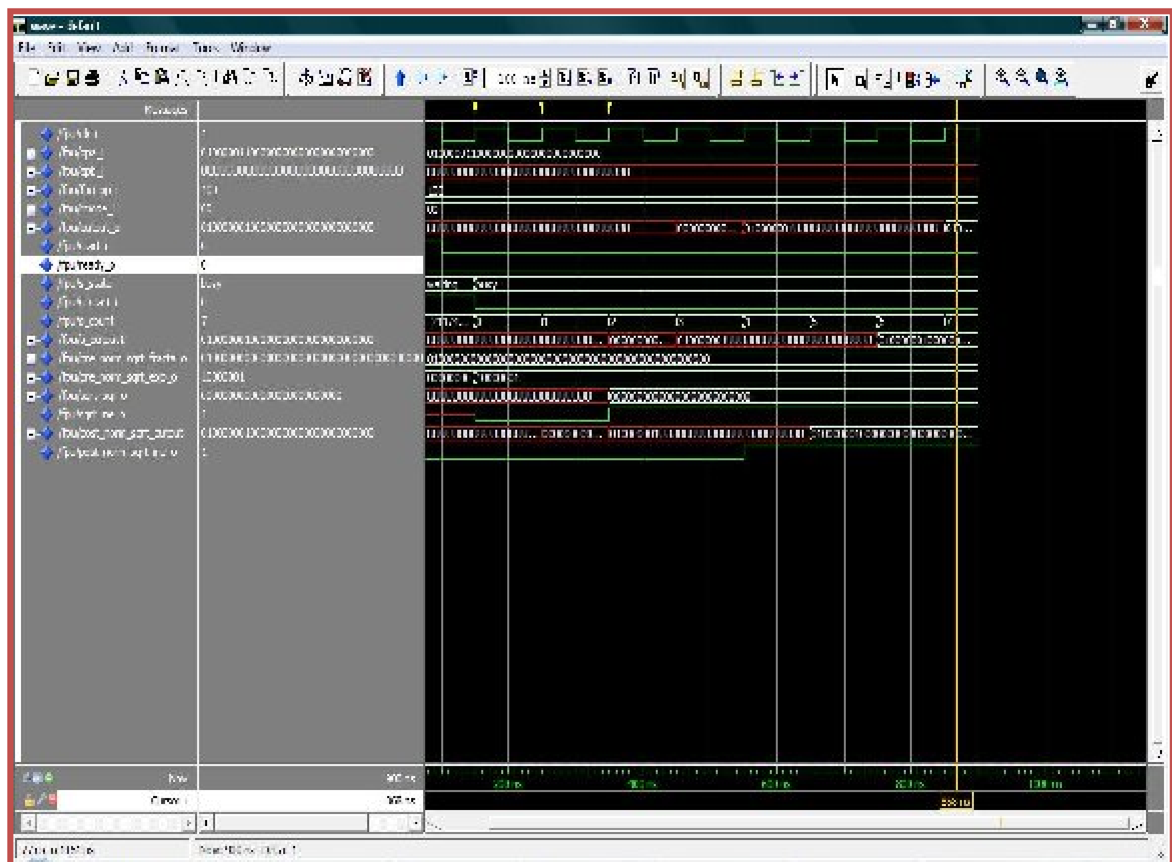


Figure 4.5: Waveforms generated while performing Square Root.

Input operends

Operand A (in base 2) = 01000001100000000000000000000000

So, $S_A = 0$, $E_A = 131 - 127 = 4$, $f_A = 0$

Operand A (in base 10) = $+ 2^4 \times 1.0 = 16.0$

Operation Code

100, for square root.

Output

Output (in base 2) = 01000000010000000000000000000000

$S_O = 0$, $e_O = 129 – 127 = 2$, $f_O = 0$

So, Output (in base 10) = $+ 2^2 \times 1.0 = 4.0$

4.2 FPGA IMPLEMENTAION

The simulated code has been synthesized for XILINX Spartan3E FPGA board by using XILINX ISE 11.1. Then code is successfully burnt to the FPGA board and the results have been verified. The summary of synthesis report is given below and the detailed synthesis report has been shown in appendix A.

4.2.1 Summary of Synthesis Report

Table 4.1 shows the summary of device utilization summary generated using XILINX ISE 11.1.

Table 4.1: Device utilization summary generated using XILINX ISE 11.1

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	3414	960	355%
Number of Slice Flip Flops	924	1920	48%
Number of 4 input LUTs	6556	1920	341%
Number of bonded IOBs	112	66	169%
Number of MULT18X18SIOs	4	4	100%
Number of GCLKs	1	24	4%

Targeted Device

Vendor : Xilinx,

Device Family: Spartan 3E

Device : XC3S5000

Package : FG320

Speed : -5

4.3 CONCLUSION

Arithmetic unit has been designed to perform five arithmetic operations, addition, subtraction, multiplication, division and square root, on floating point numbers. IEEE 754 standard based floating point representation has been used. The unit has been coded in VHDL. Code has been synthesised for the SPARTAN3E FPGA board using XILINX ISE and has been implemented and verified on the board successfully.

4.4 FUTURE SCOPE

The designed arithmetic unit operates on 32-bit operends. It can be designed for 64-bit operands to enhance precision. It can be extended to have more mathematical operations like trigonometric, logarithmic and exponential functions.

REFERENES

[1]. D. Goldberg, "What every computer scientist should know about floating-point arithmetic" pp. 5-48 in ACM Computing Surveys vol. 23-1 (1991).

[2]. Charles Farnum, "Compiler Support for Floating-Point Computation" Software Practices and Experience, pp. 701-9 vol. 18, July 1988.

[3]. IEEE computer society: IEEE Standard 754 for Binary Floating-Point Arithmetic, 1985.

[4]. W. Kahan "IEEE Standard 754 for Binary Floating-Point Arithmetic," 1996.

[5]. J.W. Demmel "The Effects of Underflow on Numerical Computation" SIAM Journal of Scientific & Statistical Computing, pp. 887- 919, vol. 5-4 (Dec. 1984).

[6]. C. Moler "IEEE Standard unifies arithmetic model Floating points".

[7]. A. A. Gaffar and Wayne "Customising Floating-Point Designs", department of Computing Imperial College 180 Queen's Gate London SW7 2BZ, England.

[8]. The New IEEE-754 Standard for Floating Point Arithmetic Peter Markstein Woodside, CA 94062.

[9]. IEC 60559: 1989, Binary floating-point arithmetic for microprocessor systems (previously designated IEC 559:1989, and equivalent to IEEE-754(1985))

[10]. Alex N. D Zamfirescus, "Floating Point Types for Synthesis", Alternative system concepts, Inc. 644 Emerson stuite 10, Palo Alto, CA USA

[11]. Michael L. Overton, "Numerical Computing with IEEE Floating Point Arithmetic," Published by Society for Industrial and Applied Mathematics, 2001.

[12]. Behrooz Parhami "Computer Arithmetic Algorithms and Hardware Designs," Published by Oxford University press, 2000.

[13]. Milios D. Ercegovac, Tomas Lang, "Digital Arithmetic," Published by Morgan Kaufmann, 2004

[14]. Taek-Jun Kwon, Jeff Sondeen, Jeff Draper USC Information Sciences Institute Design Trade-Offs institute "Floating-Point Unit Implementation for Embedded and Processing-In-Memory Systems" 4676 Admiralty Way Marina del Rey, CA 90292 U.S.A.

[15]. Liddicoat and M.J. Flynn, "High-Performance Floating-Point Divide", Euromicro Symposium on Digital System Design, Sep. 2001

[16]. W Kahan, “Why do we need a floating-point arithmetic standard?” University of California at Berkeley.

[17]. Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Paul Zimmermann “MPFR: A Multiple-Precision Binary Floating-Point Library With Correct Rounding”.

[18]. M. Leeser, X. Wang, “ Variable Precision Floating Point Division and Square Root”, Department of Electrical and Computer Engineering Northeastern University.

[19]. C. Jeannerod, H. Knochel, C. Monat, Member, IEEE, and Guillaume Revy Laboratoire, “Faster floating-point square root for integer processors”.

[20]. A. Tlawar & William Roberts, “IEEE 754 Floating-Point Division Unit,” December 9, 2004.

[21]. J. R. Hauser ACM Trans, "Handling Floating-point Exceptions in Numeric Programs", on Prog. Lang. and Syst. vol. 8-2 (Mar. 1996).

APPENDIX A

DETAILED SYNTHESIS REPORT

CONTENTS OF REPORT

1. Synthesis Options Summary
2. HDL Compilation
3. Design Hierarchy Analysis
4. HDL Analysis
5. HDL Synthesis
    a. HDL Synthesis Report
6. Advanced HDL Synthesis
    a. Advanced HDL Synthesis Report
7. Low Level Synthesis
8. Partition Report
9. Final Report
    a. Device Utilization Summary
    b. Partition Resource Summary
    c. Timing Report

===

* Synthesis Options Summary *

===

---- Source Parameters

Input File Name : "fpu.prj"

Input Format : mixed

Ignore Synthesis Constraint File : NO

---- Target Parameters

Output File Name : "fpu"

Output Format : NGC

Target Device : xc3s5000e-5-vq100

---- Source Options

Top Module Name : fpu

Automatic FSM Extraction : YES

FSM Encoding Algorithm : Auto

Safe Implementation : No

FSM Style : lut

RAM Extraction : Yes

RAM Style : Auto

ROM Extraction : Yes

Mux Style : Auto

Decoder Extraction : YES

Priority Encoder Extraction : YES

Shift Register Extraction : YES

Logical Shifter Extraction : YES

XOR Collapsing : YES

ROM Style : Auto

Mux Extraction : YES

Resource Sharing : YES

Asynchronous To Synchronous : NO

Automatic Register Balancing : No

---- Target Options

Add IO Buffers : YES

Add Generic Clock Buffer(BUFG) : 24

Register Duplication : YES

Slice Packing : YES

Optimize Instantiated Primitives : NO

Use Clock Enable : Yes

Use Synchronous Set : Yes

Use Synchronous Reset : Yes

Pack IO Registers into IOBs : auto

Equivalent register Removal : YES

---- General Options

Optimization Goal : Speed

Optimization Effort : 1

Library Search Order : fpu.lso

Keep Hierarchy : NO

Netlist Hierarchy : as_optimized

RTL Output : Yes

Global Optimization : AllClockNets

Read Cores : YES

Write Timing Constraints : NO

Cross Clock Analysis : NO

Hierarchy Separator : /

Bus Delimiter : <>

Case Specifier : maintain

Slice Utilization Ratio : 100

BRAM Utilization Ratio : 100

Verilog 2001 : YES

Auto BRAM Packing : NO

Slice Utilization Ratio Delta : 5

===

* HDL Compilation *

===

Compiling vhdl file "D:/soni_ise/Soni_fpu/fpupack.vhd" in Library work.

Architecture fpupack of Entity fpupack is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/comppack.vhd" in Library work.

Compiling vhdl file "D:/soni_ise/Soni_fpu/pre_norm_addsub.vhd" in Library work.

Architecture rtl of Entity pre_norm_addsub is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/addsub_28.vhd" in Library work.

Architecture rtl of Entity addsub_28 is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/post_norm_addsub.vhd" in Library work.

Architecture rtl of Entity post_norm_addsub is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/pre_norm_mul.vhd" in Library work.

Architecture rtl of Entity pre_norm_mul is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/mul_24.vhd" in Library work.

Architecture rtl of Entity mul_24 is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/serial_mul.vhd" in Library work.

Architecture rtl of Entity serial_mul is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/post_norm_mul.vhd" in Library work.

Architecture rtl of Entity post_norm_mul is up to date.

Compiling vhdl file "D:/soni_ise/Soni_fpu/fpu.vhd" in Library work.

Entity <fpu> compiled.

Entity <fpu> (Architecture <rtl>) compiled.

===

* Design Hierarchy Analysis *

===

Analyzing hierarchy for entity <fpu> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <pre_norm_addsub> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <addsub_28> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <post_norm_addsub> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <pre_norm_mul> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <mul_24> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <serial_mul> in library <work> (architecture <rtl>).

Analyzing hierarchy for entity <post_norm_mul> in library <work> (architecture <rtl>).

===

* HDL Analysis *

===

Analyzing Entity <fpu> in library <work> (Architecture <rtl>).

Set user-defined property "RD_WIDTH = 52" for instance <i_sqrt> in unit <fpu>.

Set user-defined property "SQ_WIDTH = 26" for instance <i_sqrt> in unit <fpu>.

Entity <fpu> analyzed. Unit <fpu> generated.

Analyzing Entity <pre_norm_addsub> in library <work> (Architecture <rtl>).

INFO:Xst:1561 - "D:/soni_ise/Soni_fpu/pre_norm_addsub.vhd" line 128: Mux is complete : default of case is discarded

INFO:Xst:1561 - "D:/soni_ise/Soni_fpu/pre_norm_addsub.vhd" line 148: Mux is complete : default of case is discarded

Entity <pre_norm_addsub> analyzed. Unit <pre_norm_addsub> generated.

Analyzing Entity <addsub_28> in library <work> (Architecture <rtl>).

Entity <addsub_28> analyzed. Unit <addsub_28> generated.

Analyzing Entity <post_norm_addsub> in library <work> (Architecture <rtl>).

Entity <post_norm_addsub> analyzed. Unit <post_norm_addsub> generated.

Analyzing Entity <pre_norm_mul> in library <work> (Architecture <rtl>).

Entity <pre_norm_mul> analyzed. Unit <pre_norm_mul> generated.

Analyzing Entity <mul_24> in library <work> (Architecture <rtl>).

Entity <mul_24> analyzed. Unit <mul_24> generated.

Analyzing Entity <serial_mul> in library <work> (Architecture <rtl>).

Entity <serial_mul> analyzed. Unit <serial_mul> generated.

Analyzing Entity <post_norm_mul> in library <work> (Architecture <rtl>).

Entity <post_norm_mul> analyzed. Unit <post_norm_mul> generated.

===

* HDL Synthesis *

===

Performing bidirectional port resolution...

Synthesizing Unit <pre_norm_addsub>.

Related source file is "D:/soni_ise/Soni_fpu/pre_norm_addsub.vhd".

Found 28-bit register for signal <fracta_28_o>.

Found 8-bit register for signal <exp_o>.

Found 28-bit register for signal <fractb_28_o>.

Found 8-bit register for signal <s_exp_diff>.

Found 8-bit adder for signal <s_exp_diff$addsub0000>.

Found 8-bit subtractor for signal <s_exp_diff$mux0001> created at line 144.

Found 8-bit register for signal <s_exp_o>.

Found 8-bit comparator greater for signal <s_expa_lt_expb$cmp_gt0000> created at line 110.

Found 28-bit shifter logical right for signal <s_fract_shr_28$shift0001> created at line 158.

Found 1-bit xor2 for signal <s_mux_diff$xor0000> created at line 139.

Found 6-bit adder for signal <s_rzeros$addsub0000> created at line 105.

Found 8-bit comparator greater for signal <s_sticky$cmp_gt0000> created at line 162.

Found 6-bit adder for signal <v_count$add0000> created at line 105.

Found 6-bit adder for signal <v_count$add0001> created at line 105.

Found 6-bit adder for signal <v_count$add0002> created at line 105.

Found 6-bit adder for signal <v_count$add0003> created at line 105.

Found 6-bit adder for signal <v_count$add0004> created at line 105.

Found 6-bit adder for signal <v_count$add0005> created at line 105.

Found 6-bit adder for signal <v_count$add0006> created at line 105.

Found 6-bit adder for signal <v_count$add0007> created at line 105.

Found 6-bit adder for signal <v_count$add0008> created at line 105.

Found 6-bit adder for signal <v_count$add0009> created at line 105.

Found 6-bit adder for signal <v_count$add0010> created at line 105.

Found 6-bit adder for signal <v_count$add0011> created at line 105.

Found 6-bit adder for signal <v_count$add0012> created at line 105.

Found 6-bit adder for signal <v_count$add0013> created at line 105.

Found 6-bit adder for signal <v_count$add0014> created at line 105.

Found 6-bit adder for signal <v_count$add0015> created at line 105.

Found 6-bit adder for signal <v_count$add0016> created at line 105.

Found 6-bit adder for signal <v_count$add0017> created at line 105.

Found 6-bit adder for signal <v_count$add0018> created at line 105.

Found 6-bit adder for signal <v_count$add0019> created at line 105.

Found 6-bit adder for signal <v_count$add0020> created at line 105.

Found 6-bit adder for signal <v_count$add0021> created at line 105.

Found 6-bit adder for signal <v_count$add0022> created at line 105.

Found 6-bit adder for signal <v_count$add0023> created at line 105.

Found 6-bit adder for signal <v_count$add0024> created at line 105.

Found 6-bit adder for signal <v_count$add0025> created at line 105.

Summary:

inferred 80 D-type flip-flop(s).

inferred 29 Adder/Subtractor(s).

inferred 2 Comparator(s).

inferred 1 Combinational logic shifter(s).

Unit <pre_norm_addsub> synthesized.

Synthesizing Unit <addsub_28>.

Related source file is "D:/soni_ise/Soni_fpu/addsub_28.vhd".

Found 1-bit register for signal <sign_o>.

Found 28-bit register for signal <fract_o>.

Found 28-bit comparator greater for signal <fracta_lt_fractb$cmp_gt0000> created at line 100.

Found 1-bit xor2 for signal <s_addop$xor0000> created at line 103.

Found 28-bit addsub for signal <s_fract_o>.

Found 1-bit xor2 for signal <s_sign_o$xor0000> created at line 106.

Summary:

inferred 29 D-type flip-flop(s).

inferred 1 Adder/Subtractor(s).

inferred 1 Comparator(s).

Unit <addsub_28> synthesized.

Synthesizing Unit <post_norm_addsub>.

Related source file is "D:/soni_ise/Soni_fpu/post_norm_addsub.vhd".

Found 32-bit register for signal <output_o>.

Found 1-bit register for signal <ine_o>.

Found 10-bit subtractor for signal <s_exp10>.

Found 10-bit adder for signal <s_exp10$addsub0000> created at line 132.

Found 9-bit register for signal <s_expo9_1>.

Found 9-bit subtractor for signal <s_expo9_2$addsub0000> created at line 172.

Found 9-bit adder for signal <s_expo9_3$addsub0000> created at line 188.

Found 28-bit register for signal <s_fracto28_1>.

Found 28-bit shifter logical right for signal <s_fracto28_1$shift0004> created at line 165.

Found 28-bit shifter logical left for signal <s_fracto28_1$shift0005> created at line 167.

Found 28-bit adder for signal <s_fracto28_rnd$addsub0000> created at line 182.

Found 1-bit xor3 for signal <s_nan_op$xor0000> created at line 198.

Found 1-bit 4-to-1 multiplexer for signal <s_roundup>.

Found 6-bit register for signal <s_shl1>.

Found 6-bit subtractor for signal <s_shl1$addsub0000> created at line 140.

Found 6-bit register for signal <s_shr1>.

Found 6-bit adder for signal <s_zeros$add0000> created at line 90.

Found 6-bit adder for signal <s_zeros$addsub0000> created at line 90.

Found 6-bit adder for signal <v_count$add0000> created at line 90.

Found 6-bit adder for signal <v_count$add0001> created at line 90.

Found 6-bit adder for signal <v_count$add0002> created at line 90.

Found 6-bit adder for signal <v_count$add0003> created at line 90.

Found 6-bit adder for signal <v_count$add0004> created at line 90.

Found 6-bit adder for signal <v_count$add0005> created at line 90.

Found 6-bit adder for signal <v_count$add0006> created at line 90.

Found 6-bit adder for signal <v_count$add0007> created at line 90.

Found 6-bit adder for signal <v_count$add0008> created at line 90.

Found 6-bit adder for signal <v_count$add0009> created at line 90.

Found 6-bit adder for signal <v_count$add0010> created at line 90.

Found 6-bit adder for signal <v_count$add0011> created at line 90.

Found 6-bit adder for signal <v_count$add0012> created at line 90.

Found 6-bit adder for signal <v_count$add0013> created at line 90.

Found 6-bit adder for signal <v_count$add0014> created at line 90.

Found 6-bit adder for signal <v_count$add0015> created at line 90.

Found 6-bit adder for signal <v_count$add0016> created at line 90.

Found 6-bit adder for signal <v_count$add0017> created at line 90.

Found 6-bit adder for signal <v_count$add0018> created at line 90.

Found 6-bit adder for signal <v_count$add0019> created at line 90.

Found 6-bit adder for signal <v_count$add0020> created at line 90.

Found 6-bit adder for signal <v_count$add0021> created at line 90.

Found 6-bit adder for signal <v_count$add0022> created at line 90.

Found 6-bit adder for signal <v_count$add0023> created at line 90.

Summary:

inferred 82 D-type flip-flop(s).

inferred 32 Adder/Subtractor(s).

inferred 1 Multiplexer(s).

inferred 2 Combinational logic shifter(s).

inferred 1 Xor(s).

Unit <post_norm_addsub> synthesized.

Synthesizing Unit <pre_norm_mul>.

Related source file is "D:/soni_ise/Soni_fpu/pre_norm_mul.vhd".

Found 10-bit register for signal <exp_10_o>.

Found 10-bit subtractor for signal <s_exp_10_o>.

Found 10-bit adder for signal <s_exp_10_o$addsub0000> created at line 101.

Found 10-bit adder for signal <s_expa_in>.

Found 10-bit adder for signal <s_expb_in>.

Summary:

inferred 10 D-type flip-flop(s).

inferred 4 Adder/Subtractor(s).

Unit <pre_norm_mul> synthesized.

Synthesizing Unit <mul_24>.

Related source file is "D:/soni_ise/Soni_fpu/mul_24.vhd".

Found 24-bit adder for signal <$add0000> created at line 208.

Found 6x6-bit multiplier for signal <$mult0000> created at line 194.

Found 6x6-bit multiplier for signal <$mult0001> created at line 195.

Found 6x6-bit multiplier for signal <$mult0002> created at line 196.

Found 6x6-bit multiplier for signal <$mult0003> created at line 197.

Found 24-bit adder for signal <add0000$addsub0000> created at line 208.

Found 24-bit adder for signal <add0000$addsub0001> created at line 208.

Found 3-bit up counter for signal <count>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0000<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0001<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0002<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0003<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0004<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0005<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0006<0>>.

Found 1-bit 4-to-1 multiplexer for signal <prod$mux0007<0>>.

Found 96-bit register for signal <prod2<0>>.

Found 96-bit register for signal <prod2<1>>.

Found 96-bit register for signal <prod2<2>>.

Found 96-bit register for signal <prod2<3>>.

Found 48-bit adder for signal <prod_a_b<4>>.

Found 48-bit adder for signal <prod_a_b_4$addsub0000> created at line 223.

Found 48-bit adder for signal <prod_a_b_4$addsub0001> created at line 223.

Found 24-bit register for signal <s_fracta_i>.

Found 24-bit register for signal <s_fractb_i>.

Found 1-bit register for signal <s_ready_o>.

Found 1-bit xor2 for signal <s_sign_o>.

Found 1-bit register for signal <s_signa_i>.

Found 1-bit register for signal <s_signb_i>.

Found 1-bit register for signal <s_start_i>.

Found 1-bit register for signal <s_state<0>>.

Found 96-bit register for signal <sum>.

Summary:

inferred 1 Counter(s).

inferred 533 D-type flip-flop(s).

inferred 6 Adder/Subtractor(s).

inferred 4 Multiplier(s).

inferred 8 Multiplexer(s).

Unit <mul_24> synthesized.

Synthesizing Unit <serial_mul>.

Related source file is "D:/soni_ise/Soni_fpu/serial_mul.vhd".

Found 1-bit register for signal <sign_o>.

Found 48-bit register for signal <fract_o>.

Found 1-bit register for signal <ready_o>.

Found 1-bit 24-to-1 multiplexer for signal <$varindex0000> created at line 127.

Found 5-bit up counter for signal <s_count>.

Found 48-bit up accumulator for signal <s_fract_o>.

Found 24-bit register for signal <s_fracta_i>.

Found 24-bit register for signal <s_fractb_i>.

Found 1-bit register for signal <s_ready_o>.

Found 1-bit xor2 for signal <s_sign_o>.

Found 1-bit register for signal <s_start_i>.

Found 1-bit register for signal <s_state<0>>.

Found 48-bit shifter logical left for signal <v_prod_shl$shift0000> created at line 135.

Summary:

inferred 1 Counter(s).

inferred 1 Accumulator(s).

inferred 101 D-type flip-flop(s).

inferred 1 Multiplexer(s).

inferred 1 Combinational logic shifter(s).

Unit <serial_mul> synthesized.

Synthesizing Unit <post_norm_mul>.

Related source file is "D:/soni_ise/Soni_fpu/post_norm_mul.vhd".

Found 32-bit register for signal <output_o>.

Found 1-bit register for signal <ine_o>.

Found 10-bit register for signal <s_exp_10_i>.

Found 10-bit adder for signal <s_exp_10a>.

Found 10-bit subtractor for signal <s_exp_10b>.

Found 8-bit register for signal <s_expa>.

Found 8-bit register for signal <s_expb>.

Found 9-bit register for signal <s_expo1>.

Found 9-bit subtractor for signal <s_expo2b$addsub0000> created at line 192.

Found 9-bit adder for signal <s_expo3$addsub0000> created at line 233.

Found 48-bit register for signal <s_frac2a>.

Found 48-bit shifter logical right for signal <s_frac2a$shift0002> created at line 185.

Found 48-bit shifter logical left for signal <s_frac2a$shift0003> created at line 187.

Found 25-bit register for signal <s_frac_rnd>.

Found 25-bit adder for signal <s_frac_rnd$addsub0000> created at line 222.

Found 48-bit register for signal <s_fract_48_i>.

Found 6-bit adder for signal <s_lost$addsub0000> created at line 197.

Found 6-bit comparator greater for signal <s_lost$cmp_gt0000> created at line 197.

Found 32-bit register for signal <s_opa_i>.

Found 32-bit register for signal <s_opb_i>.

Found 6-bit register for signal <s_r_zeros>.

Found 6-bit adder for signal <s_r_zeros$addsub0000> created at line 105.

Found 2-bit register for signal <s_rmode_i>.

Found 1-bit 4-to-1 multiplexer for signal <s_roundup>.

Found 6-bit register for signal <s_shl2>.

Found 6-bit register for signal <s_shr2>.

Found 1-bit register for signal <s_sign_i>.

Found 6-bit register for signal <s_zeros>.

Found 6-bit adder for signal <s_zeros$addsub0000> created at line 90.

Found 6-bit adder for signal <v_count$add0000> created at line 90.

Found 6-bit adder for signal <v_count$add0001> created at line 90.

Found 6-bit adder for signal <v_count$add0002> created at line 90.

Found 6-bit adder for signal <v_count$add0003> created at line 90.

Found 6-bit adder for signal <v_count$add0004> created at line 90.

Found 6-bit adder for signal <v_count$add0005> created at line 90.

Found 6-bit adder for signal <v_count$add0006> created at line 90.

Found 6-bit adder for signal <v_count$add0007> created at line 90.

Found 6-bit adder for signal <v_count$add0008> created at line 90.

Found 6-bit adder for signal <v_count$add0009> created at line 90.

Found 6-bit adder for signal <v_count$add0010> created at line 90.

Found 6-bit adder for signal <v_count$add0011> created at line 90.

Found 6-bit adder for signal <v_count$add0012> created at line 90.

Found 6-bit adder for signal <v_count$add0013> created at line 90.

Found 6-bit adder for signal <v_count$add0014> created at line 90.

Found 6-bit adder for signal <v_count$add0015> created at line 90.

Found 6-bit adder for signal <v_count$add0016> created at line 90.

Found 6-bit adder for signal <v_count$add0017> created at line 90.

Found 6-bit adder for signal <v_count$add0018> created at line 90.

Found 6-bit adder for signal <v_count$add0019> created at line 90.

Found 6-bit adder for signal <v_count$add0020> created at line 90.

Found 6-bit adder for signal <v_count$add0021> created at line 90.

Found 6-bit adder for signal <v_count$add0022> created at line 90.

Found 6-bit adder for signal <v_count$add0023> created at line 90.

Found 6-bit adder for signal <v_count$add0024> created at line 90.

Found 6-bit adder for signal <v_count$add0025> created at line 90.

Found 6-bit adder for signal <v_count$add0026> created at line 90.

Found 6-bit adder for signal <v_count$add0027> created at line 90.

Found 6-bit adder for signal <v_count$add0028> created at line 90.

Found 6-bit adder for signal <v_count$add0029> created at line 90.

Found 6-bit adder for signal <v_count$add0030> created at line 90.

Found 6-bit adder for signal <v_count$add0031> created at line 90.

Found 6-bit adder for signal <v_count$add0032> created at line 90.

Found 6-bit adder for signal <v_count$add0033> created at line 90.

Found 6-bit adder for signal <v_count$add0034> created at line 90.

Found 6-bit adder for signal <v_count$add0035> created at line 90.

Found 6-bit adder for signal <v_count$add0036> created at line 90.

Found 6-bit adder for signal <v_count$add0037> created at line 90.

Found 6-bit adder for signal <v_count$add0038> created at line 90.

Found 6-bit adder for signal <v_count$add0039> created at line 90.

Found 6-bit adder for signal <v_count$add0040> created at line 90.

Found 6-bit adder for signal <v_count$add0041> created at line 90.

Found 6-bit adder for signal <v_count$add0042> created at line 90.

Found 6-bit adder for signal <v_count$add0043> created at line 90.

Found 6-bit adder for signal <v_count0$add0000> created at line 105.

Found 6-bit adder for signal <v_count0$add0001> created at line 105.

Found 6-bit adder for signal <v_count0$add0002> created at line 105.

Found 6-bit adder for signal <v_count0$add0003> created at line 105.

Found 6-bit adder for signal <v_count0$add0004> created at line 105.

Found 6-bit adder for signal <v_count0$add0005> created at line 105.

Found 6-bit adder for signal <v_count0$add0006> created at line 105.

Found 6-bit adder for signal <v_count0$add0007> created at line 105.

Found 6-bit adder for signal <v_count0$add0008> created at line 105.

Found 6-bit adder for signal <v_count0$add0009> created at line 105.

Found 6-bit adder for signal <v_count0$add0010> created at line 105.

Found 6-bit adder for signal <v_count0$add0011> created at line 105.

Found 6-bit adder for signal <v_count0$add0012> created at line 105.

Found 6-bit adder for signal <v_count0$add0013> created at line 105.

Found 6-bit adder for signal <v_count0$add0014> created at line 105.

Found 6-bit adder for signal <v_count0$add0015> created at line 105.

Found 6-bit adder for signal <v_count0$add0016> created at line 105.

Found 6-bit adder for signal <v_count0$add0017> created at line 105.

Found 6-bit adder for signal <v_count0$add0018> created at line 105.

Found 6-bit adder for signal <v_count0$add0019> created at line 105.

Found 6-bit adder for signal <v_count0$add0020> created at line 105.

Found 6-bit adder for signal <v_count0$add0021> created at line 105.

Found 6-bit adder for signal <v_count0$add0022> created at line 105.

Found 6-bit adder for signal <v_count0$add0023> created at line 105.

Found 6-bit adder for signal <v_count0$add0024> created at line 105.

Found 6-bit adder for signal <v_count0$add0025> created at line 105.

Found 6-bit adder for signal <v_count0$add0026> created at line 105.

Found 6-bit adder for signal <v_count0$add0027> created at line 105.

Found 6-bit adder for signal <v_count0$add0028> created at line 105.

Found 6-bit adder for signal <v_count0$add0029> created at line 105.

Found 6-bit adder for signal <v_count0$add0030> created at line 105.

Found 6-bit adder for signal <v_count0$add0031> created at line 105.

Found 6-bit adder for signal <v_count0$add0032> created at line 105.

Found 6-bit adder for signal <v_count0$add0033> created at line 105.

Found 6-bit adder for signal <v_count0$add0034> created at line 105.

Found 6-bit adder for signal <v_count0$add0035> created at line 105.

Found 6-bit adder for signal <v_count0$add0036> created at line 105.

Found 6-bit adder for signal <v_count0$add0037> created at line 105.

Found 6-bit adder for signal <v_count0$add0038> created at line 105.

Found 6-bit adder for signal <v_count0$add0039> created at line 105.

Found 6-bit adder for signal <v_count0$add0040> created at line 105.

Found 6-bit adder for signal <v_count0$add0041> created at line 105.

Found 6-bit adder for signal <v_count0$add0042> created at line 105.

Found 6-bit adder for signal <v_count0$add0043> created at line 105.

Found 6-bit adder for signal <v_count0$add0044> created at line 105.

Found 6-bit adder for signal <v_count0$add0045> created at line 105.

Found 10-bit subtractor for signal <v_shl1$addsub0000> created at line 154.

Found 10-bit adder for signal <v_shr1$addsub0000> created at line 148.

Found 10-bit adder for signal <v_shr1$addsub0001> created at line 148.

Summary:

inferred 280 D-type flip-flop(s).

inferred 101 Adder/Subtractor(s).

inferred 1 Comparator(s).

inferred 1 Multiplexer(s).

inferred 2 Combinational logic shifter(s).

Unit <post_norm_mul> synthesized.

Synthesizing Unit <fpu>.

Related source file is "D:/soni_ise/Soni_fpu/fpu.vhd".

Found 32-bit register for signal <output_o>.

Found 1-bit register for signal <ready_o>.

Found 1-bit register for signal <ine_o>.

Found 1-bit register for signal <overflow_o>.

Found 1-bit register for signal <underflow_o>.

Found 1-bit register for signal <div_zero_o>.

Found 1-bit register for signal <inf_o>.

Found 1-bit register for signal <zero_o>.

Found 1-bit register for signal <qnan_o>.

Found 1-bit register for signal <snan_o>.

Found 32-bit up counter for signal <s_count>.

Found 3-bit register for signal <s_fpu_op_i>.

Found 1-bit register for signal <s_ine_o>.

Found 32-bit register for signal <s_opa_i>.

Found 32-bit register for signal <s_opb_i>.

Found 32-bit register for signal <s_output1>.

Found 1-bit xor2 for signal <s_output_o$xor0000> created at line 397.

Found 2-bit register for signal <s_rmode_i>.

Found 1-bit register for signal <s_start_i>.

Found 1-bit register for signal <s_state<0>>.

Summary:

inferred 1 Counter(s).

inferred 145 D-type flip-flop(s).

Unit <fpu> synthesized.

INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that some arithmetic operations in this design can share the same physical resources for reduced device utilization. For improved clock frequency you may try to disable resource sharing.

===

HDL Synthesis Report

===

Macro Statistics

Multipliers : 4

6x6-bit multiplier : 4

Adders/Subtractors : 173

10-bit adder : 7

10-bit subtractor : 4

24-bit adder : 3

25-bit adder : 1

28-bit adder : 1

28-bit addsub : 1

48-bit adder : 3

6-bit adder : 146

6-bit subtractor : 1

8-bit adder : 1

8-bit subtractor : 1

9-bit adder : 2

9-bit subtractor : 2

Counters : 3

3-bit up counter : 1

32-bit up counter : 1

5-bit up counter : 1

Accumulators : 1

48-bit up loadable accumulator : 1

Registers : 84

1-bit register : 26

10-bit register : 2

2-bit register : 2

24-bit register : 24

25-bit register : 1

28-bit register : 4

3-bit register : 1

32-bit register : 8

48-bit register : 3

6-bit register : 6

8-bit register : 5

9-bit register : 2

Comparators : 4

28-bit comparator greater : 1

6-bit comparator greater : 1

8-bit comparator greater : 2

Multiplexers : 11

1-bit 24-to-1 multiplexer : 1

1-bit 4-to-1 multiplexer : 10

Logic shifters : 6

28-bit shifter logical left : 1

28-bit shifter logical right : 2

48-bit shifter logical left : 2

48-bit shifter logical right : 1

Xors : 7

1-bit xor2 : 6

1-bit xor3 : 1

==

* Advanced HDL Synthesis *

==

INFO:Xst:2261 - The FF/Latch <s_shr1_1> in Unit <i_postnorm_addsub> is equivalent to the following 4 FFs/Latches, which will be removed : <s_shr1_2> <s_shr1_3> <s_shr1_4> <s_shr1_5>

INFO:Xst:2261 - The FF/Latch <prod2<3>_0_0> in Unit <i_mul_24> is equivalent to the following 47 FFs/Latches, which will be removed : <prod2<3>_0_1> <prod2<3>_0_2> <prod2<3>_0_3> <prod2<3>_0_4> <prod2<3>_0_5> <prod2<3>_0_6> <prod2<3>_0_7> <prod2<3>_0_8> <prod2<3>_0_9> <prod2<3>_0_10> <prod2<3>_0_11> <prod2<3>_3_12> <prod2<3>_3_13> <prod2<3>_3_14> <prod2<3>_3_15> <prod2<3>_3_16> <prod2<3>_3_17> <prod2<3>_3_18> <prod2<3>_3_19> <prod2<3>_3_20> <prod2<3>_3_21> <prod2<3>_3_22> <prod2<3>_3_23> <prod2<3>_1_0> <prod2<3>_1_1> <prod2<3>_1_2> <prod2<3>_1_3> <prod2<3>_1_4> <prod2<3>_1_5> <prod2<3>_1_18> <prod2<3>_1_19> <prod2<3>_1_20> <prod2<3>_1_21> <prod2<3>_1_22> <prod2<3>_1_23> <prod2<3>_2_0> <prod2<3>_2_1> <prod2<3>_2_2> <prod2<3>_2_3> <prod2<3>_2_4> <prod2<3>_2_5> <prod2<3>_2_18> <prod2<3>_2_19> <prod2<3>_2_20> <prod2<3>_2_21> <prod2<3>_2_22> <prod2<3>_2_23>

INFO:Xst:2261 - The FF/Latch <prod2<1>_0_0> in Unit <i_mul_24> is equivalent to the following 47 FFs/Latches, which will be removed : <prod2<1>_0_1> <prod2<1>_0_2> <prod2<1>_0_3> <prod2<1>_0_4> <prod2<1>_0_5> <prod2<1>_0_6> <prod2<1>_0_7> <prod2<1>_0_8> <prod2<1>_0_9> <prod2<1>_0_10> <prod2<1>_0_11> <prod2<1>_3_12> <prod2<1>_3_13> <prod2<1>_3_14> <prod2<1>_3_15> <prod2<1>_3_16> <prod2<1>_3_17> <prod2<1>_3_18> <prod2<1>_3_19> <prod2<1>_3_20> <prod2<1>_3_21> <prod2<1>_3_22> <prod2<1>_3_23> <prod2<1>_1_0> <prod2<1>_1_1> <prod2<1>_1_2> <prod2<1>_1_3> <prod2<1>_1_4> <prod2<1>_1_5> <prod2<1>_1_18> <prod2<1>_1_19> <prod2<1>_1_20> <prod2<1>_1_21> <prod2<1>_1_22> <prod2<1>_1_23> <prod2<1>_2_0> <prod2<1>_2_1> <prod2<1>_2_2> <prod2<1>_2_3> <prod2<1>_2_4> <prod2<1>_2_5> <prod2<1>_2_18>

<prod2<1>_2_19> <prod2<1>_2_20> <prod2<1>_2_21> <prod2<1>_2_22> <prod2<1>_2_23>
INFO:Xst:2261 - The FF/Latch <prod20>_2_21> <prod2<0>_2_22> <prod2<0>_2_23> <prod2<0>_0_0> <prod2<0>_0_1> <prod2<0>_0_2> <prod2<0>_0_3> <prod2<0>_0_4> <prod2<0>_0_5> <prod2<0>_0_6> <prod2<0>_0_7> <prod2<0>_0_8> <prod2<0>_0_9> <prod2<0>_0_10> <prod2<0>_0_11> <prod2<0>_1_0> <prod2<0>_1_1> <prod2<0>_1_2> <prod2<0>_1_3> <prod2<0>_1_4> <prod2<0>_1_5> <prod2<0>_1_18> <prod2<0>_1_19> <prod2<0>_1_20> <prod2<0>_1_21> <prod2<0>_1_22> <prod2<0>_1_23> <prod2<0>_3_12> <prod2<0>_3_13> <prod2<0>_3_14> <prod2<0>_3_15> <prod2<0>_3_16> <prod2<0>_3_17> <prod2<0>_3_18> <prod2<0>_3_19> <prod2<0>_3_20> <prod2<0>_3_21> <prod2<0>_3_22> <prod2<0>_3_23>
INFO:Xst:2261 - The FF/Latch <prod2<2>_0_0> in Unit <i_mul_24> is equivalent to the following 47 FFs/Latches, which will be removed : <prod2<2>_0_1> <prod2<2>_0_2> <prod2<2>_0_3> <prod2<2>_0_4> <prod2<2>_0_5> <prod2<2>_0_6> <prod2<2>_0_7> <prod2<2>_0_8> <prod2<2>_0_9> <prod2<2>_0_10> <prod2<2>_0_11> <prod2<2>_1_0> <prod2<2>_1_1> <prod2<2>_1_2> <prod2<2>_1_3> <prod2<2>_1_4> <prod2<2>_1_5> <prod2<2>_1_18> <prod2<2>_1_19> <prod2<2>_1_20> <prod2<2>_1_21> <prod2<2>_1_22> <prod2<2>_1_23> <prod2<2>_2_0> <prod2<2>_2_1> <prod2<2>_2_2> <prod2<2>_2_3> <prod2<2>_2_4> <prod2<2>_2_5> <prod2<2>_2_18> <prod2<2>_2_19> <prod2<2>_2_20> <prod2<2>_2_21> <prod2<2>_2_22> <prod2<2>_2_23> <prod2<2>_3_12> <prod2<2>_3_13> <prod2<2>_3_14> <prod2<2>_3_15> <prod2<2>_3_16> <prod2<2>_3_17> <prod2<2>_3_18> <prod2<2>_3_19> <prod2<2>_3_20> <prod2<2>_3_21> <prod2<2>_3_22> <prod2<2>_3_23>
INFO:Xst:2261 - The FF/Latch <s_expb_0> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_23>
INFO:Xst:2261 - The FF/Latch <s_expb_1> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_24>
INFO:Xst:2261 - The FF/Latch <s_opa_i_23> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_0>
INFO:Xst:2261 - The FF/Latch <s_expb_2> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_25>

INFO:Xst:2261 - The FF/Latch <s_expb_7> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_30>
INFO:Xst:2261 - The FF/Latch <s_opa_i_24> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_1>
INFO:Xst:2261 - The FF/Latch <s_expb_3> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_26>
INFO:Xst:2261 - The FF/Latch <s_opa_i_25> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_2>
INFO:Xst:2261 - The FF/Latch <s_opa_i_30> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_7>
INFO:Xst:2261 - The FF/Latch <s_expb_4> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_27>
INFO:Xst:2261 - The FF/Latch <s_opa_i_26> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_3>
INFO:Xst:2261 - The FF/Latch <s_expb_5> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_28>
INFO:Xst:2261 - The FF/Latch <s_opa_i_27> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_4>
INFO:Xst:2261 - The FF/Latch <s_expb_6> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_29>
INFO:Xst:2261 - The FF/Latch <s_opa_i_28> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_5>
INFO:Xst:2261 - The FF/Latch <s_opa_i_29> in Unit <i_post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_6>
WARNING:Xst:1710 - FF/Latch <s_expo9_1_8> (without init value) has a constant value of 0 in block <i_postnorm_addsub>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1710 - FF/Latch <s_shr1_1> (without init value) has a constant value of 0 in block <i_postnorm_addsub>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1710 - FF/Latch <prod2<3>_0_0> (without init value) has a constant value of 0 in block <i_mul_24>. This FF/Latch will be trimmed during the optimization process.

WARNING:Xst:1710 - FF/Latch <prod2<0>_2_0> (without init value) has a constant value of 0 in block <i_mul_24>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1710 - FF/Latch <prod2<1>_0_0> (without init value) has a constant value of 0 in block <i_mul_24>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1710 - FF/Latch <prod2<2>_0_0> (without init value) has a constant value of 0 in block <i_mul_24>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1710 - FF/Latch <s_expo1_8> (without init value) has a constant value of 0 in block <i_post_norm_mul>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:2677 - Node <s_opa_i_31> of sequential type is unconnected in block <i_post_norm_mul>.
WARNING:Xst:2677 - Node <s_opb_i_31> of sequential type is unconnected in block <i_post_norm_mul>.
WARNING:Xst:2404 - FFs/Latches <s_shr1<5:1>> (without init value) have a constant value of 0 in block <post_norm_addsub>.
WARNING:Xst:2404 - FFs/Latches <prod2<3>_2<23:18>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<3>_1<23:18>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<3>_3<23:12>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<0>_1<23:18>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<0>_3<23:12>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<0>_2<23:18>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<1>_1<23:18>> (without init value) have a constant value of 0 in block <mul_24>.
WARNING:Xst:2404 - FFs/Latches <prod2<1>_2<23:18>> (without init value) have a constant value of 0 in block <mul_24>.

WARNING:Xst:2404 - FFs/Latches <prod2<1>_3<23:12>> (without init value) have a constant value of 0 in block <mul_24>.

WARNING:Xst:2404 - FFs/Latches <prod2<2>_2<23:18>> (without init value) have a constant value of 0 in block <mul_24>.

WARNING:Xst:2404 - FFs/Latches <prod2<2>_1<23:18>> (without init value) have a constant value of 0 in block <mul_24>.

WARNING:Xst:2404 - FFs/Latches <prod2<2>_3<23:12>> (without init value) have a constant value of 0 in block <mul_24>.

ARNING:Xst:2677 - Node <s_opa_i_31> of sequential type is unconnected in block <post_norm_mul>.

WARNING:Xst:2677 - Node <s_opb_i_31> of sequential type is unconnected in block <post_norm_mul>.

===

Advanced HDL Synthesis Report

===

Macro Statistics

Multipliers : 4

6x6-bit multiplier : 4

Adders/Subtractors : 170

10-bit adder : 3

10-bit adder carry in : 1

10-bit subtractor : 3

24-bit adder : 3

25-bit adder : 1

28-bit adder : 1

28-bit addsub : 1

48-bit adder : 3

6-bit adder : 144

6-bit adder carry in : 1

6-bit subtractor : 2

7-bit adder carry in : 1

8-bit adder : 1

8-bit subtractor : 1

9-bit adder : 2

9-bit subtractor : 2

Counters : 3

3-bit up counter : 1

32-bit up counter : 1

5-bit up counter : 1

Accumulators : 1

48-bit up loadable accumulator : 1

Registers : 1157

Flip-Flops : 1157

Comparators : 4

28-bit comparator greater : 1

6-bit comparator greater : 1

8-bit comparator greater : 2

Multiplexers : 11

1-bit 24-to-1 multiplexer : 1

1-bit 4-to-1 multiplexer : 10

Logic shifters : 6

28-bit shifter logical left : 1

28-bit shifter logical right : 2

48-bit shifter logical left : 2

48-bit shifter logical right : 1

Xors : 7

1-bit xor2 : 6

1-bit xor3 : 1

===

* Low Level Synthesis *

===

INFO:Xst:2261 - The FF/Latch <s_expb_0> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_23>

INFO:Xst:2261 - The FF/Latch <s_expb_1> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_24>

INFO:Xst:2261 - The FF/Latch <s_opa_i_23> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_0>

INFO:Xst:2261 - The FF/Latch <s_expb_2> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_25>
INFO:Xst:2261 - The FF/Latch <s_expb_7> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_30>
INFO:Xst:2261 - The FF/Latch <s_opa_i_24> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_1>
INFO:Xst:2261 - The FF/Latch <s_expb_3> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_26>
INFO:Xst:2261 - The FF/Latch <s_opa_i_25> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_2>
INFO:Xst:2261 - The FF/Latch <s_opa_i_30> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_7>
INFO:Xst:2261 - The FF/Latch <s_expb_4> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_27>
INFO:Xst:2261 - The FF/Latch <s_opa_i_26> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_3>
INFO:Xst:2261 - The FF/Latch <s_expb_5> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_28>
INFO:Xst:2261 - The FF/Latch <s_opa_i_27> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_4>
INFO:Xst:2261 - The FF/Latch <s_expb_6> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_opb_i_29>
INFO:Xst:2261 - The FF/Latch <s_opa_i_28> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_5>
INFO:Xst:2261 - The FF/Latch <s_opa_i_29> in Unit <post_norm_mul> is equivalent to the following FF/Latch, which will be removed : <s_expa_6>
WARNING:Xst:1710 - FF/Latch <fractb_28_o_27> (without init value) has a constant value of 0 in block <pre_norm_addsub>. This FF/Latch will be trimmed during the optimization process.
WARNING:Xst:1895 - Due to other FF/Latch trimming, FF/Latch <fracta_28_o_27> (without init value) has a constant value of 0 in block <pre_norm_addsub>. This FF/Latch will be trimmed during the optimization process.
Optimizing unit <fpu> ...
Optimizing unit <pre_norm_addsub> ...

Optimizing unit <addsub_28> ...

Optimizing unit <post_norm_addsub> ...

Optimizing unit <pre_norm_mul> ...

Optimizing unit <mul_24> ...

Optimizing unit <serial_mul> ...

Optimizing unit <post_norm_mul> ...

Mapping all equations...

Building and optimizing final netlist ...

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_10> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_10>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_11> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_11>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_12> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_12>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_13> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_13>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_14> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_14>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_20> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_20>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_15> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_15>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_21> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_21>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_16> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_16>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_22> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_22>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_17> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_17>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_18> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_18>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_19> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_19>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_10> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_10>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_11> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_11>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_12> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_12>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_13> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_13>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_14> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_14>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_20> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_20>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_15> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_15>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_21> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_21>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_16> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_16>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_17> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_17>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_22> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_22>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_18> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_18>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_19> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_19>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_0> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_0>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_1> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_1>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_2> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_2>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_3> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_3>

INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_4> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_4>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_5> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_5>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_6> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_6>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_7> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_7>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_8> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_8>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fractb_i_9> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opb_i_9>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_0> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_0>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_1> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_1>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_2> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_2>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_3> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_3>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_4> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_4>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_5> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_5>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_6> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_6>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_7> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_7>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_8> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_8>
INFO:Xst:2261 - The FF/Latch <i_mul_24/s_fracta_i_9> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_post_norm_mul/s_opa_i_9>
INFO:Xst:2261 - The FF/Latch <s_start_i> in Unit <fpu> is equivalent to the following FF/Latch, which will be removed : <i_mul_24/s_start_i>

Found area constraint ratio of 100 (+ 5) on block fpu, actual ratio is 306.

Optimizing block <fpu> to meet ratio 100 (+ 5) of 960 slices :

FlipFlop i_post_norm_mul/s_fract_48_i_1 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_2 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_3 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_39 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_4 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_40 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_41 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_42 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_43 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_44 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_45 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_46 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_5 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_6 has been replicated 1 time(s)

FlipFlop i_post_norm_mul/s_fract_48_i_7 has been replicated 1 time(s)

Final Macro Processing ...

Processing Unit <fpu> :

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_6>.

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_5>.

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_4>.

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_3>.

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_2>.

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_1>.

Found 2-bit shift register for signal <i_prenorm_addsub/exp_o_0>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_9>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_8>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_7>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_6>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_5>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_4>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_3>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_2>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_1>.

Found 2-bit shift register for signal <i_post_norm_mul/s_exp_10_i_0>.

Unit <fpu> processed.

===

Final Register Report

===

Macro Statistics

Registers : 908

Flip-Flops : 908

Shift Registers : 17

2-bit shift register : 17

===

* Partition Report *

===

Partition Implementation Status

No Partitions were found in this design.

===

* Final Report *

===

Final Results

RTL Top Level Output File Name : fpu.ngr

Top Level Output File Name : fpu

Output Format : NGC

Optimization Goal : Speed

Keep Hierarchy : NO

Design Statistics

IOs : 112

Cell Usage :

BELS : 8085

GND : 1

INV : 16

LUT1 : 153

LUT2 : 422

LUT2_D : 25

LUT2_L : 14

LUT3 : 1254

LUT3_D : 123

LUT3_L : 63

LUT4 : 3560

LUT4_D : 542

LUT4_L : 367

MULT_AND : 18

MUXCY : 406

MUXF5 : 807

VCC : 1

XORCY : 313

FlipFlops/Latches : 925

FD : 452

FDE : 289

FDR : 21

FDRE : 35

FDRS : 10

FDS : 116

FDSE : 2

Shift Registers : 17

SRL16 : 17

Clock Buffers : 1

BUFGP : 1

IO Buffers : 111

IBUF : 70

OBUF : 41

MULTs : 4

MULT18X18SIO : 4

Others : 6

post_norm_div : 1

post_norm_sqrt : 1

pre_norm_div : 1

pre_norm_sqrt : 1

serial_div : 1

sqrt : 1

===

Device utilization summary:

===

Selected Device : 3s100evq100-5

Number of Slices: 3414 out of 960 355% (*)

Number of Slice Flip Flops: 924 out of 1920 48%

Number of 4 input LUTs: 6556 out of 1920 341% (*)

Number used as logic: 6539

Number used as Shift registers: 17

Number of IOs: 112

Number of bonded IOBs: 112 out of 66 169% (*)

IOB Flip Flops: 1

Number of MULT18X18SIOs: 4 out of 4 100%

Number of GCLKs: 1 out of 24 4%

WARNING:Xst:1336 - (*) More than 100% of Device resources are used

===

Partition Resource Summary:

===

No Partitions were found in this design.

===

TIMING REPORT

===

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.

FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT

GENERATED AFTER PLACE-and-ROUTE.

Clock Information:

-----------------------------------+------------------------+-------+

Clock Signal | Clock buffer(FF name) | Load |

----------------------------------+------------------------+-------+

clk_i | BUFGP | 942 |

----------------------------------+------------------------+-------+

Asynchronous Control Signals Information:

No asynchronous control signals found in this design

Timing Summary:

Speed Grade: -5

Minimum period: 54.255ns (Maximum Frequency: 18.432MHz)

Minimum input arrival time before clock: 9.121ns

Maximum output required time after clock: 4.040ns

Maximum combinational path delay: 2.655ns

Timing Detail:

All values displayed in nanoseconds (ns)

===

Timing constraint: Default period analysis for Clock 'clk_i'

Clock period: 54.255ns (frequency: 18.432MHz)

Total number of paths / destination ports: 2141048702513967 / 1387

Delay: 54.255ns (Levels of Logic = 50)

Source: i_post_norm_mul/s_fract_48_i_0 (FF)

Destination: i_post_norm_mul/s_r_zeros_5 (FF)

Source Clock: clk_i rising

Destination Clock: clk_i rising

Data Path: i_post_norm_mul/s_fract_48_i_0 to i_post_norm_mul/s_r_zeros_5

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

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

FD:C->Q 19 0.514 0.925 i_post_norm_mul/s_fract_48_i_0 (i_post_norm_mul/s_fract_48_i_0)

LUT4_D:I3->O 16 0.612 0.948 i_post_norm_mul/v_count0_mux0003<0>1_1 (i_post_norm_mul/v_count0_mux0003<0>1)

LUT2_D:I1->LO 1 0.612 0.103 i_post_norm_mul/Madd_v_count0_add0004_cy<1>11_SW0 (N6989)

LUT4:I3->O 7 0.612 0.605 i_post_norm_mul/v_count0_mux0005<2>1 (i_post_norm_mul/Madd_v_count0_add0009_lut<2>)

LUT4:I3->O 4 0.612 0.529 i_post_norm_mul/v_count0_mux0006<3>1 (i_post_norm_mul/Madd_v_count0_add0008_lut<3>)

LUT4:I2->O 9 0.612 0.700 i_post_norm_mul/v_count0_mux0007<3>1 (i_post_norm_mul/Madd_v_count0_add0007_lut<3>)

LUT4_D:I3->O 11 0.612 0.796 i_post_norm_mul/v_count0_mux0008<3>1 (i_post_norm_mul/Madd_v_count0_add0006_lut<3>)

LUT4_D:I3->O 3 0.612 0.454 i_post_norm_mul/v_count0_mux0009<3>1 (i_post_norm_mul/Madd_v_count0_add0005_lut<3>)

LUT4_D:I3->O 12 0.612 0.820 i_post_norm_mul/v_count0_mux0010<3>157 (i_post_norm_mul/Madd_v_count0_add0010_lut<3>)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0012<3>183_G (N1449)

MUXF5:I1->O 5 0.278 0.541 i_post_norm_mul/v_count0_mux0012<3>183 (i_post_norm_mul/Madd_v_count0_add0012_lut<3>)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0014<3>2_F (N1214)

MUXF5:I0->O 15 0.278 0.867 i_post_norm_mul/v_count0_mux0014<3>2 (i_post_norm_mul/N835)

LUT4:I3->O 6 0.612 0.572 i_post_norm_mul/v_count0_mux0014<3>125 (i_post_norm_mul/v_count0_mux0014<3>125)

LUT4:I3->O 15 0.612 0.867 i_post_norm_mul/v_count0_mux0014<3>140 (i_post_norm_mul/Madd_v_count0_add0014_lut<3>)

LUT4:I3->O 13 0.612 0.839 i_post_norm_mul/v_count0_mux0015<4>1 (i_post_norm_mul/Madd_v_count0_add0015_lut<4>)

LUT4:I3->O 16 0.612 0.909 i_post_norm_mul/v_count0_mux0017<4>1 (i_post_norm_mul/Madd_v_count0_add0018_lut<4>)

LUT4:I2->O 4 0.612 0.529 i_post_norm_mul/v_count0_mux0019<4>1 (i_post_norm_mul/Madd_v_count0_add0016_lut<4>)

LUT3:I2->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0020<4>253_F (N5949)

MUXF5:I0->O 7 0.278 0.632 i_post_norm_mul/v_count0_mux0020<4>253 (i_post_norm_mul/N440)

LUT3:I2->O 1 0.612 0.387 i_post_norm_mul/v_count0_mux0021<4>80 (i_post_norm_mul/v_count0_mux0021<4>80)

LUT4:I2->O 5 0.612 0.541 i_post_norm_mul/v_count0_mux0021<4>119 (i_post_norm_mul/Madd_v_count0_add0021_lut<4>)

LUT4:I3->O 3 0.612 0.520 i_post_norm_mul/v_count0_mux0023<4>59 (i_post_norm_mul/v_count0_mux0023<4>59)

LUT4:I1->O 8 0.612 0.646 i_post_norm_mul/v_count0_mux0023<4>94 (i_post_norm_mul/Madd_v_count0_add0023_lut<4>)

LUT4:I3->O 2 0.612 0.449 i_post_norm_mul/v_count0_mux0025<4>83 (i_post_norm_mul/v_count0_mux0025<4>83)

LUT4:I1->O 11 0.612 0.796 i_post_norm_mul/v_count0_mux0025<4>97 (i_post_norm_mul/Madd_v_count0_add0025_lut<4>)

LUT4_D:I3->O 12 0.612 0.886 i_post_norm_mul/v_count0_mux0030<4>19 (i_post_norm_mul/Madd_v_count0_add0029_lut<4>)

LUT4:I1->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0030<4>2_F (N6019)

MUXF5:I0->O 6 0.278 0.572 i_post_norm_mul/v_count0_mux0030<4>2 (i_post_norm_mul/N435)

LUT4_D:I3->O 9 0.612 0.700 i_post_norm_mul/v_count0_mux0030<4>129 (i_post_norm_mul/Madd_v_count0_add0030_lut<4>)

LUT4:I3->O 13 0.612 0.839 i_post_norm_mul/v_count0_mux0031<4>112 (i_post_norm_mul/Madd_v_count0_add0031_lut<4>)

LUT4:I3->O 2 0.612 0.383 i_post_norm_mul/v_count0_mux0033<4>61 (i_post_norm_mul/v_count0_mux0033<4>61)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0033<4>100_F (N1098)

MUXF5:I0->O 7 0.278 0.605 i_post_norm_mul/v_count0_mux0033<4>100 (i_post_norm_mul/Madd_v_count0_add0033_lut<4>)

LUT4:I3->O 9 0.612 0.727 i_post_norm_mul/v_count0_mux0034<4>184 (i_post_norm_mul/Madd_v_count0_add0034_lut<4>)

LUT4:I2->O 14 0.612 0.880 i_post_norm_mul/v_count0_mux0036<4>1 (i_post_norm_mul/Madd_v_count0_add0038_lut<4>)

LUT4:I2->O 5 0.612 0.541 i_post_norm_mul/v_count0_mux0038<4>1 (i_post_norm_mul/Madd_v_count0_add0036_lut<4>)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0040<4>2_F (N1564)

MUXF5:I0->O 6 0.278 0.572 i_post_norm_mul/v_count0_mux0040<4>2 (i_post_norm_mul/N938)

LUT4_D:I3->O 7 0.612 0.605 i_post_norm_mul/v_count0_mux0040<4>95 (i_post_norm_mul/Madd_v_count0_add0040_lut<4>)

LUT4:I3->O 6 0.612 0.572 i_post_norm_mul/v_count0_mux0044<4>2_SW1 (N569)

LUT4:I3->O 1 0.612 0.387 i_post_norm_mul/v_count0_mux0044<4>2_SW3 (N2273)

LUT4:I2->O 4 0.612 0.499 i_post_norm_mul/v_count0_mux0043<4>49 (i_post_norm_mul/Madd_v_count0_add0043_lut<4>)

MUXF5:S->O 1 0.641 0.360 i_post_norm_mul/v_count0_mux0044<5>144_F_SW0_SW0 (N3621)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/v_count0_mux0044<5>144_F (N940)

MUXF5:I0->O 5 0.278 0.568 i_post_norm_mul/v_count0_mux0044<5>144 (i_post_norm_mul/Madd_v_count0_add0044_lut<5>)

LUT3:I2->O 2 0.612 0.383 i_post_norm_mul/s_r_zeros_mux0000<5>147 (i_post_norm_mul/s_r_zeros_mux0000<5>147)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/s_r_zeros_mux0000<5>217_SW1_F (N5733)

MUXF5:I0->O 1 0.278 0.360 i_post_norm_mul/s_r_zeros_mux0000<5>217_SW1 (N2242)

LUT4_L:I3->LO 1 0.612 0.103 i_post_norm_mul/s_r_zeros_mux0000<5>254 (i_post_norm_mul/s_r_zeros_mux0000<5>254)

LUT4:I3->O 1 0.612 0.000 i_post_norm_mul/s_r_zeros_mux0000<5>524 (i_post_norm_mul/s_r_zeros_mux0000<5>)

FD:D 0.268 i_post_norm_mul/s_r_zeros_5

Total 54.255ns (28.739ns logic, 25.516ns route)

(53.0% logic, 47.0% route)

===

Timing constraint: Default OFFSET IN BEFORE for Clock 'clk_i'

Total number of paths / destination ports: 708 / 165

Offset: 9.121ns (Levels of Logic = 6)

Source: fpu_op_i<2> (PAD)

Destination: output_o_0 (FF)

Destination Clock: clk_i rising

Data Path: fpu_op_i<2> to output_o_0

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

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

IBUF:I->O 71 1.106 1.235 fpu_op_i_2_IBUF (fpu_op_i_2_IBUF)

LUT3:I0->O 2 0.612 0.532 s_div_zero_o_cmp_eq0000_inv1 (s_div_zero_o_cmp_eq0000_inv)

LUT4:I0->O 1 0.612 0.360 s_output_o_or0000_SW0 (N87)

LUT4:I3->O 4 0.612 0.651 s_output_o_or0000 (s_output_o_or0000)

LUT4:I0->O 1 0.612 0.360 s_output_o<0>2_SW0 (N98)

LUT4:I3->O 23 0.612 1.022 s_output_o<0>2 (N4)

FDS:S 0.795 output_o_0

Total 9.121ns (4.961ns logic, 4.160ns route)

(54.4% logic, 45.6% route)

===

Timing constraint: Default OFFSET OUT AFTER for Clock 'clk_i'

Total number of paths / destination ports: 241 / 241

Offset: 4.040ns (Levels of Logic = 1)

Source: div_zero_o (FF)

Destination: div_zero_o (PAD)

Source Clock: clk_i rising

Data Path: div_zero_o to div_zero_o

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

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

FDR:C->Q 1 0.514 0.357 div_zero_o (div_zero_o_OBUF)

OBUF:I->O 3.169 div_zero_o_OBUF (div_zero_o)

Total 4.040ns (3.683ns logic, 0.357ns route)

(91.2% logic, 8.8% route)

==

Timing constraint: Default path analysis

Total number of paths / destination ports: 235 / 235

Delay: 2.655ns (Levels of Logic = 1)

Source: clk_i (PAD)

Destination: i_pre_norm_div:clk_i (PAD)

Data Path: clk_i to i_pre_norm_div:clk_i

Gate Net

Cell:in->out fanout Delay Delay Logical Name (Net Name)

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

BUFGP:I->O 942 1.457 1.199 clk_i_BUFGP (clk_i_BUFGP)

pre_norm_div:clk_i 0.000 i_pre_norm_div

Total 2.655ns (1.457ns logic, 1.199ns route)

(54.9% logic, 45.1% route)