DEVELOPING AN EFFICIENT IEEE 754 COMPLIANT FPU IN VERILOG

A Thesis Submitted For The Partial

Fulfilment Of Requirements For Degree Of

Bachelor Of Technology
In
Computer Science and Engineering

BY

RUBY DEV (108CS069)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

ROURKELA - 769008, INDIA

DEVELOPING AN EFFICIENT IEEE 754 COMPLIANT FPU IN VERILOG

A Thesis Submitted For The Partial

Fulfilment Of Requirements For Degree Of

Bachelor Of Technology
In
Computer Science and Engineering

BY

RUBY DEV (108CS069)

UNDER THE GUIDANCE OF

Prof. P. M. KHILAR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY ROURKELA

ROURKELA - 769008, INDIA

NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA-769008, ODISHA, INDIA

<u>CERTIFICATE</u>

This is to certify that the thesis entitled, "**DEVELOPING AN EFFICIENT IEEE754 COMPLIANT FPU IN VERILOG**" submitted by Ms. Ruby Dev (108CS069) in partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Computer Science and Engineering at National Institute of Technology, Rourkela is an authentic work carried out by her under my supervision and guidance.

Prof. P. M. KHILAR

Department of Computer Science and Engineering

National Institute of Technology

Rourkela- 769008

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Date:

Place:

ACKNOWLEDGEMENT

I wish to express my sincere and heartfelt gratitude towards my guide Prof. P. M. KHILAR, Computer Science Engineering Department, for his supervision, sympathy, and inspiration and above all help in all regards during the entire duration of my project without which, completion of the project was not possible at all. His guidance has been crucial for giving me a deep insight into the project.

I would also like to thank all the professors of the department of Computer Science and Engineering, National Institute of Technology, Rourkela, for their constant motivation and guidance.

I am really thankful to all my friends. My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism or their valuable time, I am truly indebted.

I must also acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, support, patience, understanding and guidance.

Ruby Dev
(108CS069)

ABSTRACT

A **floating-point unit** (**FPU)** colloquially is a **math coprocessor,** which is a part of a computer system specially designed to carry out operations on floating point numbers. Typical operations that are handled by FPU are addition, subtraction, multiplication and division. The aim was to build an efficient FPU that performs basic as well as transcendental functions with reduced complexity of the logic used reduced or at least comparable time bounds as those of x87 family at similar clock speed and reduced the memory requirement as far as possible. The functions performed are handling of Floating Point data, converting data to IEEE754 format, perform any one of the following arithmetic operations like addition, subtraction, multiplication, division and shift operation and transcendental operations like square Root, sine of an angle and cosine of an angle. All the above algorithms have been clocked and evaluated under Spartan 3E Synthesis environment. All the functions are built by possible efficient algorithms with several changes incorporated at our end as far as the scope permitted. Consequently all of the unit functions are unique in certain aspects and given the right environment(in terms of higher memory or say clock speed or data width better than the FPGA Spartan 3E Synthesizing environment) these functions will tend to show comparable efficiency and speed ,and if pipelined then higher throughput.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

- **FPU** Floating Point Unit
- **FP** Floating Point
- **GRFPU** Gaisler Research Floating Point Unit
- **CC** Clock Cycles
- **CLA** carry look ahead
- **NRD** non Restoring division
- **RTL** Register Transfer Level

CHAPTER 1

INTRODUCTION

Floating Point Unit

IEEE 754 Standards

Motivation

Literature Review

Floating-point units (FPU) colloquially are a math coprocessor which is designed specially to carry out operations on floating point numbers [1]. Typically FPUs can handle operations like addition, subtraction, multiplication and division. FPUs can also perform various transcendental functions such as exponential or trigonometric calculations, though these are done with software library routines in most modern processors.

1.1 FLOATING POINT UNIT

When a CPU executes a program that is calling for a floating-point (FP) operation, there are three ways by which it can carry out the operation. Firstly, it may call a floating-point unit emulator, which is a floating-point library, using a series of simple fixed-point arithmetic operations which can run on the integer ALU. These emulators can save the added hardware cost of a FPU but are significantly slow. Secondly, it may use an add-on FPUs that are entirely separate from the CPU, and are typically sold as an optional add-ons which are purchased only when they are needed to speed up math-intensive operations. Else it may use integrated FPU present in the system [2].

The FPU designed by us is a single precision IEEE754 compliant integrated unit. It can handle not only basic floating point operations like addition, subtraction, multiplication and division but can also handle operations like shifting, square root determination and other transcendental functions like sine, cosine and tangential function.

1.2 IEEE 754 STNDARDS

IEEE754 standard is a technical standard established by IEEE and the most widely used standard for floating-point computation, followed by many hardware (CPU and FPU) and software implementations [3]. Single-precision floating-point format is a computer number format that occupies 32 bits in a computer memory and represents a wide dynamic range of values by using a floating point. In IEEE 754-2008, the 32-bit with base 2 format is officially referred to as single precision or binary32. It was called single in IEEE 754-1985. The IEEE 754 standard specifies a single precision number as having sign bit which is of 1 bit length, an exponent of width 8 bits and a significant precision of 24 bits out of which 23 bits are explicitly stored and 1 bit is implicit 1.

Sign bit determines the sign of the number where 0 denotes a positive number and 1 denotes a negative number. It is the sign of the mantissa as well. Exponent is an 8 bit signed integer from −128 to 127 (2's Complement) or can be an 8 bit unsigned integer from 0 to 255 which is the accepted biased form in IEEE 754 single precision definition. In this case an exponent with value 127 represents actual zero. The true mantissa includes 23 fraction bits to the right of the binary point and an implicit leading bit (to the left of the binary point) with value 1 unless the exponent is stored with all zeros. Thus only 23 fraction bits of the mantissa appear in the memory format but the total precision is 24 bits.

For example:

```
  S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
31 30         23 22                            0
```

IEEE754 also defines certain formats which are a set of representation of numerical values and symbols. It may also include how the sets are encoded.

The standard defines [4]:

- **Arithmetic formats** which are sets of binary and decimal floating-point numbers, which consists of finite numbers including subnormal number and signed zero, a special value called "not a number" (NaN) and infinity.
- **Interchange formats** which are bit strings (encodings) that are used to exchange a floating-point data in a compact and efficient form.

- **Rounding rules** which are the properties that should be satisfied while doing arithmetic operations and conversions of any numbers on arithmetic formats.
- **Exception handling** which indicates any exceptional conditions (like division by zero, underflow, overflow, etc.) occurred during the operations.

The standard defines the following five rounding rules:

- Round to the nearest even which rounds to the nearest value with an even (zero) least significant bit.
- Round to the nearest odd which rounds to the nearest value above (for positive numbers) or below (for negative numbers)
- Round towards positive infinity which is a rounding directly towards a positive infinity and it is also called rounding up or ceiling.
- Round towards negative infinity which is rounding directly towards a negative infinity and it is also called rounding down or floor or truncation.

The standard also defines five exceptions, and all of them return a default value. They all have a corresponding status flag which are raised when any exception occurs, except in certain cases of underflow. The five possible exceptions are:

- **Invalid** operation are like square root of a negative number, returning of qNaN by default, etc., output of which does not exist.
- **Division by zero** is an operation on a finite operand which gives an exact infinite result for e.g., 1/0 or log(0) that returns positive or negative infinity by default.
- **Overflow** occurs when an operation results a very large number that can’t be represented correctly i.e. which returns ±infinity by default (for round-to-nearest mode).

- **Underflow** occurs when an operation results very small i.e. outside the normal range and inexact (denormalised value) by default.
- **Inexact** occurs when any operation returns correctly rounded result by default.

1.3 MOTIVATION

Floating-point calculation is considered to be an esoteric subject in the field of Computer Science [5]. This is obviously surprising, because floating-point is omnipresent in computer systems. Floating-point (FP) data type is almost present in every language. From PCs to supercomputers, all have FP accelerators in them. Most compilers are called from time to time to compile the floating-point algorithms and virtually every OS have to respond to all FP exceptions during operations such as overflow. Also FP operations have a direct effect on designs as well as designers of computer systems. So it is very important to design an efficient FPU such that the computer system becomes efficient. Further, FPU can be improvised by using efficient algorithm for the basic as well as transcendental functions, which can be handled by any FPU, with reduced complexity of the logic used. This FPU further can be worked upon to improvise further complex operations-viz. exponent, etc. It can be designed so that it can handle different data types like character, strings etc, can serve as a backbone for designing a fault tolerant IEEE754 compliant FPU on higher grounds and such that pipeline can be implemented.

1.4 LITERATURE REVIEW

When a CPU is executing a program that calls for a FP operation, a separate FPU is called to carry out the operation. So, the efficiency of the FPU is of great importance. Though, not many have had great achievements in this field, but the work by the following two are appreciable.

Open Floating Point Unit – This was the open source project done by Rudolf Usselmann [6]. His FPU described a single precision floating point unit which could perform add, subtract, multiply, divide, and conversion between FP number and integer. It consists of two pre-normalization units that can adjust the mantissa as well as the exponents of the given numbers. One unit is for addition and subtraction operation and the other one is for multiplication and division operations. It also has different units for different operations that perform an actual addition subtraction, multiplication and division. It also has a shared post normalization unit that normalizes the fraction part. The final result after post-normalization is directed to a valid result which is in accordance to single precision FP format. The main drawback of this model was that most of the codes were written in MATLAB and due to this it is non-synthesizable.

GRFPU –This high Performance IEEE754 FPU was designed at Gaisler Research for the improvement of FP operations of a LEON based systems [7]. It supports both single precision and double precision operands. It implements all floating point arithmetic operations defined by the IEEE754 standard in hardware. All operations are dealt with the exception of denormalized numbers which are flushed to zero and supports all rounding modes. This advanced design combines low latency and high throughput. The most common operations such as addition, subtraction and multiplication are fully pipelined which has throughput of one CC and a latency of three CC. More complex divide and square root operation takes between 1 to 24 CC to complete and execute in parallel with other FP operations. It can also perform operations like converse and compliment. It supports all SPARC V8 FP instructions. The main drawback of this model is that it is very expensive and complex to implement practically.

CHAPTER 2

BACKGROUND

Insight To Our FPU

Features Implemented in the Design Of FPU

Implementation In A Nutshell

Performing The Selected Operation

2.1 INSIGHT TO OUR FLOATING POINT UNIT

Our Floating Point Unit is a single precision IEEE754 compliant integrated unit. It incorporates various basic operations like addition, subtraction, multiplication, division, shifting and other transcendental functions like square root determination and trigonometric operations like sine, cosine and tangential value evaluation.

2.2 FEATURES IMPLEMENTED IN THIS DESIGN OF THE FPU

This document describes a single precision floating point unit. The floating point unit is fully IEEE 754 compliant. The design implemented here incorporates the following modules. Both the module name and its functionality have been specified here in sequence of the manner they appear in the attached code:-

Module Name	**Functionality**
Cnvrt_2_integral_form	Converts 32 bit integral and 32 fractional part into single novel integral representation
cnvrt_2_ieee	Converts 32 bit binary to its equivalent IEEE-754 format
pre_normalization	Adjusts the operands by performing the necessary shifts before an add or subtract operation
add	Performs addition
sub	Performs subtraction
post_normalization	Normalizes the result of add/sub operation to its IEEE754 form
multiplication	Performs pre-normalization and multiplication of the operands intended to be multiplied and finally post-normalization of the result
Division	Performs pre-normalization and division of the operands intended to be divided, determines the remainder and finally post-normalization of the result
Squareroot determination	Evaluates the square root of the first operand op1_ieee
Shifting	Performs the shifting of the operand to the specified bit in specified direction
Cordic	Performs the trigonometric evaluation

Table 2.1: Modules and its Functionalities

2.2.1 OPERATION MODES

fpu_op	Operation
0	Add
1	Subtract
2	Multiply
3	Divide
4	Shifting
5	Find Square Root
6	Find Trigonometric values

Table 2.2 Operation Modes

2.2.3 ROUNDING MODES

Since the input is taken initially without consideration of the decimal point the only rounding method used is truncation.

Rmode	Rounding Mode
0	Truncation

Table 2.3 Rounding Mode

However by allocating two special parts i.e. INTEGER_OP and FRACTIONAL_OP, we have introduced the working for fractional parts too and include any one of the following rounding techniques:-

- Round to nearest even
- Round to nearest odd
- Round to zero
- Round to infinity

2.2.4 INTERFACES

This table lists all inputs and outputs of the FPU and provides a general description of their functions.

Signal Name	Width	Type	Description
Clk	1 bit	Input	System Clock
Rst	1 bit	Input	Reset values for initializing
Op1	32 bit	Input	Operand 1
Op2	32 bit	Input	Operand 2
Oper	2 bit	Input	Mode of operation
Rmode	2 bit	Input	Mode of rounding
Op1_ieee	32 bit	Output	IEEE-754 format of Operand 1
Op2_ieee	32 bit	Output	IEEE-754 format of Operand 2
Oper_result	32 bit	Output	Result of the selected operation in IEEE format
Underflow	1 bit	Output	If operand or result is below range of representation
Overflow	1 bit	Output	If operand or result is above range of representation
Div_by_0	1 bit	Output	If the divisor is zero then this exception is raised

Table 2.4 Interfaces

2.3 IMPLEMENTATION IN A NUTSHELL

The entire design is implemented by the following steps in progression.

- Conversion of the Floating Point Number into a novel integral representation.
- Conversion of the binary integer to its IEEE754 format.
- Pre-normalization of the operands
- Performing the selected operation.
- Post-normalize the output obtained.
- Detecting and handling the exceptions encountered.

2.3.1 CONVERSION OF FLOATING POINT NUMBER INTO A NOVEL INTEGRAL REPRESENTATION

As our FPU works with floating point numbers, the operations, intermediate calculations and output are conventionally in the same floating point structure. But this invariably increases the complexity of calculation and the number of adjustments required at each level to obtain the correct result. Our proposal is to convert the floating point number into a simple yet quite precise integral representation and perform the calculations on the same, followed by the final conversion of the output into its expected floating point result format.

The floating point data is inputted in two parts. The first part is a 32 bit binary value of the **integer part** of the floating point operand and other is a 32 bit binary value of **fractional part** of the floating point operand. This is done because Verilog cannot deal with floating point numbers. So we need to consolidate the two parts (integral and fractional) of the operand into a single 32 bit **effective operand.** This is done by the following algorithm:

Step 1: The sign bit (31st bit) of the input **integer part** becomes the sign bit of the **effective operand**.

Step 2: Then the position of 1st significant 1 is searched in the input **integer part** from RHS. This **position** is stored.

Step 3: All the bits from this position to the end of the input **integer part** (i.e. till the 0th bit) is taken and inserted into the **effective operand** from its 30th bit onward.(This step stores the actual useful bits of the **integer part** as not all the 32 bits are used to accommodate the **integer part**.)

Step 4: If there are still positions in the **effective operand** that are not filled, then it is filled with the bits from the input **fractional part** from its MSB down to the number of bits equal to places left to be filled.(This step stores the just requisite number of bits from the fractional part to complete the 32 bit representation)

This can be explained with the help of an example.

Float_op_int = 32’b00000010101000110101000011100000

Float_op_frc = 32’b11111111111110000000000111111111

Step 1: Assign output[31] = Float_op_int[31]

Step 2: Pos of 1st 1 from LHS of Float_op_int = 25(pos counted from RHS)

Step 3: Assign output = Float_op_int[25:0]

Step 4: Remaining bits left to be assigned in

remaning = 32-26-1 = 5

Step 5: output[4:0] = Float_op_frc[31:27]

$$\text{Output} = \underbrace{0}_{\text{(From Integer part)}}\ \underbrace{10101000110101000011100000}_{\text{(From Integer part)}}\ \underbrace{11111}_{\text{(From Fraction part)}}$$

So, basically our technique gives preference to the fractional part for smaller numbers and the integer part for larger ones thus keeping intact the effective precision of the floating point number.

2.3.2 CONVERSION OF THE BINARY INTEGER TO ITS IEEE754 FORMAT

As our FPU is IEEE754 compliant, the next step is to convert the input (here the **effective operand** into the IEEE specified format.

IEEE754 single precision can be encoded into 32 bits using 1 bit for the sign bit (the most significant i.e. 31st bit), next eight bits are used for the exponent part and finally rest 23 bits are used for the mantissa part.

```
         S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
`       31 30      23 22                      0
```

However, it uses an implicit bit, so the significant part becomes 24 bits, even though it usually is encoded using 23 bits.

This conversion can be done using the below algorithm:

Step 1: Sign bit of the binary number becomes the sign bit (31st bit) of the IEEE equivalent.

Step 2: 30th bit to 8th bit of the binary number becomes the mantissa part of the IEEE equivalent.

Step 3: The exponent part is calculated by subtracting the **position** of the 1st one obtained in the algorithm described in section 2.2.1.

Step 4: A bias of 127 is added to the above exponent value.

This can be explained with the help of an example.

Output = 01010100011010100001110000011111

Pos = 25 (from above calculation)

Step 1: op_ieee[31] = output[31]

Step 2: op_ieee[30:23] = 29-pos+127 = 131

Step 3: op_ieee[22:0] = 01010001101010000111000
(bits that follows the 1st 1 from LHS of output)

```
Op_ieee = 0 10000011 01010001101010000111000
          S Exponent      Mantissa
```

2.3.3 PRE-NORMALIZATION OF THE OPERANDS

Pre-normalization is the process of equalizing the exponents of the operands and accordingly adjusting the entire IEEE754 expression of the inputs to produce correct results maintaining the IEEE754 standard throughout all calculation steps inclusive of the intermediate calculations and their outputs.

This conversion can be done using the below algorithm:

Step 1: Insert the implicit 1 in the mantissa part of each of the operands.

Step 2: Find positive **difference** between the exponents of the operands

Step 3: Set the lower operand’s exponent same as that of the operand with higher exponent.

Step 4: Right shift mantissa of the lower operand by steps equal to **difference** calculated.

This can be explained with the help of an example.

Op1_ieee = 0 10000011 01010001101010000111000

Op2_ieee = 0 10000010 01010001101011100111000

Temp_op1_ieee = 101010001101010000111000 (After adding implicit 1 to Op1_ieee’s mantissa)

Temp_op2_ieee = 101010001101011100111000 (After adding implicit 1 to Op2_ieee’s mantissa)

Exponent of Temo_op1_ieee(10000011)> Exponent of Temp_op2_ieee(10000010)

Difference = 1 (10000011-10000010)

Temp_op2_ieee = 01010100011010111001100

Note: This algorithm for normalization is used only for addition and subtraction. Pre-normalization for other operations are done separately along with their calculation.

2.4 PERFORMING THE SELECTED OPERATION

After completion of the preliminary steps the next step is to perform the actual operation.The choice of operation is taken as input via a 4 bit wire **oper**. Following is the table of the functions and their corresponding operation code.

fpu_op	Operation
0	Add
1	Subtract
2	Multiply
3	Divide
4	Shifting
5	Find Square Root
6	Find Trigonometric values

Table 2.5 Operations

2.4.1 MODULE ADD

Addition is a mathematical operation which represents combining a collection of objects together to form larger collection. The process of developing an efficient addition module in our FPU was an iterative process and with gradual improvement at each attempt.

2.4.1.1 ADD USING THE "+" OPERATOR

The initial attempt was to add using the simple in-built "+" operator available in Verilog library. It used a 23 bit register sum and a 1 bit register Co (for carry). The algorithm for the addition can be described as follows:

Step 1: Check if oper = 4'b0000

Step 2: {Co,Sum} = Temp_op1_ieee[22:0] + Temp_op2_ieee[22:0]

Step 3: If carry is 1, then

- Resultant_exponent = Larger_exponent + 1;

Else if carry is 0, then do

- Resultant_exponent = Larger_exponent – (21-difference) (difference as in sec.2.2.3)

Step 4: Check for overflow and underflow-

- If for any of the operands (sign(operand with greater exponent)==0 & (exp_greater + 1 > 255)) then, Set the overflow flag to 1.
- Else if (sign(operand with lesser exponent==0) & (exp_lesser<0)), then set the underflow flag to 1

Step 5: Aggregate the **result** as concatenation of {Sign_bit,Resultant_exponent,Sum}

2.4.2 SUBTRACT MODULE

Subtraction is an operation which is treated as inverse of addition operation. The process of developing an efficient SUB module followed the iterative development of the ADD module.

2.4.2.1 SUB USING THE "-" OPERATOR

The initial attempt was to subtract using the simple in-built "-" operator available in Verilog library. It used a 23 bit register diff and a 1 bit register borrow (for borrow). The algorithm for the subtraction module can be described as follows:

Step 1: Check if oper = 4'b0001

Step 2: {borrow.diff} = Temp_op1_ieee[22:0] - Temp_op2_ieee[22:0]

Step 3: Resultant_exponent = Larger_exponent + (21-difference) (difference as in sec.2.2.3)

Step 4: Check for overflow and underflow-

- If for any operand (sign(operand with greater exponent)==1 AND (exp_greater + 1 < 0))

  Set the overflow flag to 1

- If for any operand (sign(operand with exponent)==1'b1 AND (exp_lesser>8'd255))

  Set the underflow flag to 1

Step 5: Aggregate the **result** as concatenation of {Sign_bit,Resultant_exponent,diff}

2.4.3 MULTIPLICATION MODULE

The process of developing an efficient multiplication module was iterative and with gradual improvement at each attempt. The product of two n-digit operands can be accommodated in 2n-digit operand.

2.3.3.1 MULTIPLICATION USING “*” OPERATOR

It used a 47 bit register to store the product

Step 1: Check if oper = 4’b0010

Step 2: product = Temp_op1_ieee[22:0] * Temp_op2_ieee[22:0]

Step 3: Resultant_exponent = op1_ieee[30:23] + op2_ieee[30:23] - 127

Step 4: Check for overflow

```
        If for product ( Resultant_exponent >255 ), then do,

            • Set the overflow flag to 1

Step 5: Sign_bit = op1_ieee[31] ^op2_ieee[31]

Step 6: Aggregate the **result** as concatenation of { Sign_bit, Resultant_exponent, product }
```

2.4.4 MODULE DIVISION

Division is regarded as the most complex and time-consuming of the four basic arithmetic operations. Given two inputs, a dividend and a divisor, division operation has two components as its result, quotient and a remainder.

<u>2.3.4. DIVISION USING '/' OPERATOR</u>

The initial attempt was to divide two numbers using the simple in-built "/" operator available in Verilog library. It used a 32 bit result_div_ieee register to store the quotient and register remainder to store the remainder of the division operation.

Algorithm:

```
Step 1: Check if the oper = 4 bit 0100
Step 2: result_div_ieee = temp_op1_ieee[22:0] / temp_op2_ieee[22:0]
Step 3: Check for the exception
        If op2_ieee[30:0] is all 0
            • Set div_bby_zero flag to 1
Step 4: Aggregate the result as concatenation of {Sign_bit, Resultant_exponent, result_div_ieee}
```

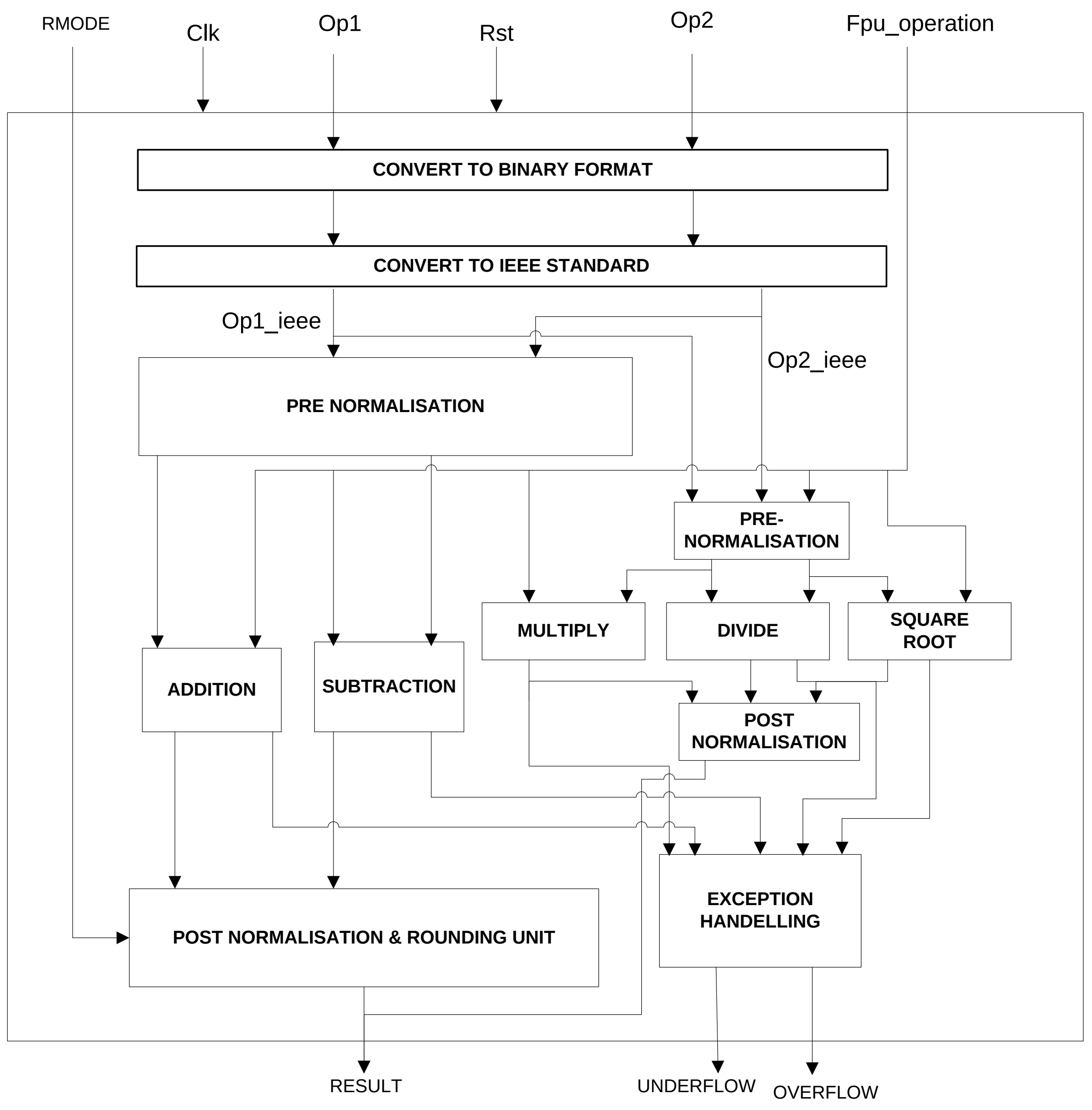

Figure 2.1 Block Diagram Of FPU

CHAPTER 3

EFFICIENT ALGORITHMS

Efficient Addition Algorithm

Efficient Subtraction Algorithm

Efficient Multiplication Algorithm

Efficient Division Algorithm

Efficient Shifting Algorithm

Efficient Square Root Determination Algorithm

As the efficiency of the FP operation carried out by the FPU is very much responsible for the efficiency of the Computer System, It is very much necessary to implement not only efficient algorithms, but to reduce the memory requirement, reduce the clock cycles for any operations, and to reduce the complexity of the logic used.

In the path to make a better and efficient FPU, we have tried to use the preexisting efficient algorithms and incorporate few changes in them or combine different positive aspects of already existing algorithms. This has resulted in positive and better or at least comparable results than that of preexisting FPUs results of which has been provided in the last chapter.

3.1 EFFICIENT ADDITION ALGORITHM

We initially tried to implement Carry Look Ahead (CLA) addition algorithm for the addition operation of 24 bits, using four 6-bit adders. But since CLA has fan-in problem due the large no. of inputs required to generate a carry bit esp. for higher bit carries, we had implemented block CLA where output carry of one block is input to the other adder block. Further, to reduce the number of gate required, we have implemented further variations in the CLA algorithm which has been explained in section 3.1.3.

3.1.2 ADD USING THE CLA

This adder works on the principle of generating and propagating a carry. [8] The structure of this adder is simplest and theoretically the most efficient in terms of time required for generation of carry for every single bit of the operand pair. It uses two function called **Generate Function** and **Propagate Function.** If the generate function for any stage (say i) is 1 then, carry for stage i+1 will be 1 independent of the input carry for the stage i. Propagate function means that, if either x_i or y_i is 1, then carry for that stage will be produced.

Generate function → G_i = op1[i] & op2[i]

Propagate function → P_i = op1[i] ^ op2[i]

Sum for the i^{th} bit pair of operand1 and operand2 → $S_i = P_i \wedge C_{i-1}$

Carry for the i^{th} bit pair of operand1 and operand2 → $C_i = G_{i-1} + P_{i-1} C_{i-1}$

Thus in general:-

$C_1 = G_0 + P_0.C_0$ [Where C_0 is the initial Carry-in bit]

$C_2 = G_1 + G_0.P_1 + P_1.P_0.C_0$

……………………………………………………….

$C_{24} = G_{23} + G_{22}.P_{23} + G_{21}.P_{23}.P_{22} + G_{20}.P_{23}.P_{22}.P_{21} + \ldots\ldots\ldots\ldots\ldots\ldots +$

$P_{23}.P_{22}.P_{21}.P_{20}.P_{19}\ldots\ldots\ldots\ldots\ldots\ldots P_1.P_0.C_0$

The algorithm can be described as follows:

Step 1: Check if oper = 4’b0000

Step 2: Generate all the G_i‘s and P_i’s.

Step 3: Generate al the C_i’s and S_i’s

Step 4: Consolidate all the S_i’s to **Sum**.

Step 5: **Co** = C_{24}

Step 6: Set sign bit (as we consider only same sign nos. sign bit is and of the individual sign bits of the operands.)

Step 7: Check for underflow and overflow same as in Section 2.3.1.1.

Step 8: Same as Step 5 in Section 2.3.1.1.

3.1.3 ADD USING THE BLOCK CLA

The initial algorithm has a fan-in problem due the large no. of inputs required to generate a carry bit esp. for higher bit carries. A solution to this is to divide the bits into blocks that propagate carry at block level as in Ripple Carry Adder and at intra-block level perform the CLA add structure [8]. We have a 24 bit add and this is divided into 4 blocks of 6 bits the formula for calculation from G_i to G_{i+5} remains the same as above.

The algorithm is described as follows:

Step 1: Check if oper = 4’b0000

Step 2: Generate all the G_i‘s and P_i’s.

Step 3: Generate al the C_i’s and S_i’s of a block.

Step 4: Propagate the final carry.

Step 5: Repeat steps 3 and 4 for every block.

Step 6: Consolidate all the S_i’s to **Sum**.

Step 7: **Co** = C_{24}

Step 8: Set sign bit (as we consider only same sign nos. sign bit is and of the individual sign bits of the operands.)

Step 9: Check for underflow and overflow same as in Section 2.3.1.1.

Step 8: Same as Step 5 in Section 2.3.1.1.

3.1.4 ADD USING THE BLOCK CLA ADDER WITH REDUCED FAN IN

Our motivation was to reduce the no. of AND/OR gates used in the sub-expressions for each block further reducing the fan-in. So at the expense of a little propagation delay we tried to reduce the gate nos. thereby considerably reducing the GATE DELAYS. Thus we can achieve the reduced gate requirement which has been explained in the following example.

For example, in block 1→

$C_1 = G_0 + P_0.C_0$

$C_2 = G_1 + P_1 (G_0 + P_0.C_0)$ (Saves 1 gate & causes 1 gate delay)

$C_3 = G_2 + P_2.G_1 + P_2.P_1 (G_0 + P_0.C_0)$ (Saves 2 gates & causes 1 gate delay)

$C_4 = G_3 + P_3 (G_2 +P_2.G_1) + P_3.P_2.P_1 (G_0 + P_0.C_0)$ (Saves 4 gates & causes 2 gate delay)

$C_5 = G_4 + P_4.G_3 + P_4.P_3 (G_2 + P_2.G_1) + P_4.P_3.P_2.P_1 (G_0 + C_0)$

(Saves 6 gates & causes 2 gate delay)

$C_6 = G_5 + P_5.G_3 + P_5.P_4.G_2 + P_5.P_4 (G_2 + P_3.G_1) + P_5.P_4.P_3.P_2 (G_0 + P_1.C_0)$

(Saves 6 gates & causes 2 gate delay)

Total gates saved in block 1 = 1+2+4+6+6 = 19

Total delay caused by gate saving = 1+1+2*4 = 10

So total time saved = 19*0.5-10*0.5=4.5 units

So basically it's a faster technique which not only eliminates fan-in problem of CLA but reduces the required number of gates too. The algorithm is described as follows:

Step 1: Check if oper = 4'b0000

Step 2: Generate all the G_i 's and P_i's.

Step 3: Generate al the C_i's and S_i's of a block using the new formula.

Step 4: Propagate the final carry.

Step 5: Repeat steps 3 and 4 for every block.

Step 6: Consolidate all the S_i's to **Sum**.

Step 7: **Co** = C_{24}

Step 8: Set sign bit (as we consider only same sign nos. sign bit is and of the individual sign bits of the operands.)

Step 9: Check for underflow and overflow same as in Section 2.3.1.1.

Step 8: Same as Step 5 in Section 2.3.1.1.

3.2 EFFICIENT SUBTRACTION ALGORITHM

Subtraction can be interpreted as addition of a positive and a negative number. So using the same algorithm as that of addition, we can complete the subtraction operation by taking complement of the negative number and adding 1 to the complement. This is same as taking the 2's complement of the negative number. Doing this we interpreted the negative number as positive and carry the addition operation.

3.2.1 SUB USING THE CLA ADDER

Basically subtraction can be implemented using same CLA, which was used for the addition operation and now will work for the subtraction of two operands, one is a positive operand and other will be 2's complement of the second operand. The algorithm can be explained in the following way:

Step 1: Check if oper = 4'b0001

Step 2: Two's complement the 2nd operand

Step 3: Now consider the operand1 and the one obtained in step2 as the summands.

Step 4: Generate all the G_i's and P_i's.

Step 5: Generate al the C_i's and S_i's

Step 6: Consolidate all the S_i's to **diff**.

Step 7: **borrow** = C_{24}

Step 8: Set sign bit (as we consider only same sign nos. sign bit is and of the individual sign bits of the operands.)

Step 9: Check for underflow and overflow same as in Section 2.3.2.1.

Step 10: Same as Step 5 in Section 2.3.2.1.

3.2.2 SUB USING THE BLOCK CLA ADDER

Works the same way as CLA block, generates values for a 6 bit block where there are 4 such blocks. Similarly as in addition operation, here the carry output of i^{th} block will be the carry input of the $(i+1)^{th}$ block, where the carry propagation at block level is similar to Ripple Carry Adder but at intra-block level is similar to the CLA add structure. Here the second operand is use in its two's compliment form. The subtraction operation using the CLA can be explained using the following algorithm:

Step 1: Check if oper = 4'b0001

Step 2: Two's complement the 2^{nd} operand

Step 3: Now consider the operand1 and the one obtained in step2 as the summands.

Step 4: Generate all the G_i's and P_i's.

Step 5: Generate al the C_i's and S_i's of a block.

Step 6: Propagate the final carry.

Step 7: Repeat steps 3 and 4 for every block.

Step 8: Consolidate all the S_i's to **diff**.

Step 9: **borrow** = C_{24}

Step 10: Set sign bit (as we consider only same sign nos. sign bit is and of the individual sign bits of the operands.)

Step 11: Check for underflow and overflow same as in Section 2.3.2.1.

Step 12: Same as Step 5 in Section 2.3.2.1.

2.3.3 SUB USING THE BLOCK CLA ADDER WITH REDUCED FAN IN

This algorithm works in the same way as CLA block used in addition operation which generates values for a 6 bit block where there are 4 such blocks using the compound common taking expression obtained in section 3.1.4. We take the two's compliment of the second operand to carry out the subtraction operation. The algorithm used can be described in the following way:

Step 1: Check if oper = 4'b0001

Step 2: Two's complement the 2nd operand

Step 3: Now consider the operand1 and the one obtained in step2 as the summands.

Step 4: Generate all the G_i's and P_i's.

Step 5: Generate al the C_i's and S_i's of a block using the new formula.

Step 6: Propagate the final carry.

Step 7: Repeat steps 3 and 4 for every block.

Step 8: Consolidate all the S_i's to **diff**.

Step 9: **borrow** = C_{24}

Step 10: Set sign bit (as we consider only same sign nos. sign bit is and of the individual sign bits of the operands.)

Step 11: Check for underflow and overflow same as in Section 2.3.2.1.

Step 12: Same as Step 5 in Section 2.3.2.1.

2.4 EFFICIENT MULTIPLICATION ALGORITHM

Multiplication of negative number using 2’s complement is more complicated than multiplication of a positive number. This is because performing a straightforward unsigned multiplication of the 2's complement representations of the inputs does not give the correct result. Multiplication can be designed in such that it first converts all their negative inputs to positive quantities and use the sign bit of the original inputs to determine the sign bit of the result. But this increases the time required to perform a multiplication, hence decreasing the efficiency of the whole FPU. Here initially we have used Bit Pair Recoding algorithm which increases the efficiency of multiplication by pairing. To further increase the efficiency of the algorithm and decrease the time complexity, we have combined the Karatsuba algorithm with the bit pair recoding algorithm.

2.4.1 MULTIPLICATION USING BIT PAIR RECODING

This technique divides the maximum number of summands into two halves. It is directly derived from the Booth’s algorithm [9]. It basically works on the principle of finding the cumulative effect of two bits of the multiplier at positions i and i+1 when performed at position i. This is further clarified in the following table.

Multiplier bit pair		Multiplier bit on the right	Multiplicand selected at position i
i+1	i	i-1	Effective oper.x M
0	0	0	0 x M
0	0	1	+1 x M
0	1	0	+1 x M
0	1	1	+2 x M
1	0	0	-2 x M
1	0	1	-1 x M
1	1	0	-1 x M
1	1	1	0 x M

Table 3.1Bit Pair Recoding

The algorithm can be described as follows:

Step 1: Pair the bits of the multiplicand.

Step 2: Refer the table and operate on M accordingly find summands at i^{th} level

Step 3: Increase by 2 value of i.

Step 4: Repeat steps 2 & 3 till the last possible value of i (here 22)

Step 5: Add the summands obtained in each step.

Step 6: Execute steps 3-5 of algorithm in section 2.4.1.

Further, the algorithm is being explained with the help of an example:-

```
            0 1 1 0 1 (+13)
X          1 1 0 1 0 (-6)
-----------------------------------
-----------------------------------


            0  1  1  0  1       (+13)
X          1 -1 +1 -1 0
-----------------------------------
 0 0 0 0 0 0 0 0 0 0
 1 1 1 1 1 0 0 1 1
 0 0 0 0 1 1 0 1
 1 1 1 0 0 1 1
 0 0 0 0 0 0
-------------------------------------
 1 1 1 0 1 1 0 0 1 0    (-78)



            0  1  1  0  1       (+13)
X           0     -1   -2
----------------------------------------
1 1 1 1 1 0 0 1 1 0
1 1 1 1 0 0 1 1
0 0 0 0 0 0
----------------------------------------
1 1 1 0 1 1 0 0 1 0    (-78)
```

2.4.2 MULTIPLICATION USING BIT PAIR RECODING AND KARATSUBA ALGORITHM

The **Karatsuba algorithm** is a fast multiplication algorithm that reduces the multiplication of two *n*-digit numbers to at most $3n^{\log_2 3} \approx 3n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3}$ when *n* is a power of 2) [10].

The basic step of this algorithm is a formula that allows us to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts.

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can split the two given numbers as follows

$X = x_1B^m + x_0$

$Y = y_1B^m + x_0$

Where x_0 and y_0 are less than B^m. The product is then

$xy = (x_1B^m + x_0) + (y_1B^m + x_0)$

$= z_2B^{2m} + z_1B^m + z_0$

Where

$Z_2 = x_1y_1$

$Z_1 = x_1y_0 + x_0y_1$

$Z_0 = x_0y_0$

Ad we can see these formulae require 4 smaller multiplications. Karatsuba observed that xy can be calculated in only 3 multiplications, at the cost of few extra additions operations:

Let $Z_2 = x_1y_1$

Let $Z_0 = x_0y_0$

Let $z1 = (x_1 + x_0) * (y_0 + y_1) – z_2 – z_0$

Since

$Z_1 = x_1y_0 + x_0y_1$

$= (x_1 y_1 + x_0 y_1 + x_0y_0 + x_1y_0) - x_1y_1 - x_0y_0$

$= (x_1 + x_0) * (y_0 + y_1) - x_1y_1 - x_0y_0$

Example:

To compute the product of 1234 and 5678, choose *B* = 10 and *m* = 2. Then

$12\ 34 = 12 \times 10^2 + 34$

$56\ 78 = 56 \times 10^2 + 78$

$z_2 = 12 \times 56 = 672$

$z_0 = 34 \times 78 = 2652$

$z_1 = (12 + 34)(56 + 78) - z_2 - z_0 = 46 \times 134 - 672 - 2652 = 2840$

Result = $z_2 \times 10^{2\times 2} + z_1 \times 10^2 + z_0 = 672 \times 10000 + 2840 \times 100 + 2652 = 7006652$.

We implemented an algorithm combining both Karatsuba and bit pair recoding and hence, reducing the simultaneous solving of summands by ¼ th of the normal multiplication. Moreover the number of summands and also the size of the multiplicand were found to be reduced by half further facilitating quick and smaller multiplications. The algorithm can be described as follows:

Step 1: Divide the multiplicand into two equal halves. (Let them be A and B each 12 bits)

Step 2: Divide the multiplier into two halves. (Let them be C and D each 12 bits)

Step 3: Perform bit recoding and find Z_2

Step 4: Perform bit recoding and find Z_1.

Step 5: Perform bit recoding and find Z_0.

Step 6: Calculate $Z_2 \times 2^{2m} + Z_1 \times 2^m + Z_0$ (Here m=12)

So basically:

- The time complexity of bit pair recoding = O(n/2)
- The time complexity of Karatsuba = $O(n^{\log3/\log2})$
- The time complexity of bit pair recoding = O(n/4)

2.5 EFFICIENT DIVISION ALGORITHM

As already discussed before, division is the most complex and time-consuming operation of the four basic arithmetic operations. Given two inputs, a dividend and a divisor, division operation has two components as its result i.e. quotient and a remainder.

2.5.1 DIVISION USING NON-RESTORING DIVIION (NRD)

The division that has been used in our FPU is based on the Non-restoring division algorithm. It is considered as a sequence of addition or subtraction and shifting operations [10]. Here, correction of the quotient bit, when final remainder and the dividend has different sign, and restoration of the remainder are postponed to later steps of the algorithm, unlike restoration division. In this algorithm, restoration of the operation is totally avoided. Main advantage of this NRD algorithm is the compatibility with the 2's complement notation used for the division of negative numbers. The algorithm follows in the following manner:

Step 1: Check if oper = 4'b0100

Step 2: Set the value of register A as 24 bit 0

Step 3: Set the value of register M as Divisor (24 bit)

Step 4: Set the value of register Q as Dividend (24 bit)

Step 5: Concatenate A with Q

Step 6: Repeat the following "**n**" number of times (here n is no. of bits in divisor):

- If the sign bit of A equals 0, shift A and Q combined, left by 1 bit, and subtract M from A. Else shift A and Q combined, left by 1 bit and add M to A
- Now if sign bit of A equals 0, then set q0 as 1, else set q0 as 0

Step 7: Finally if the sign bit of A equals 1 then add M to A.

Step 8: Check for division by zero exception as in section 2.3.4.1

Step 9: Assign value of register A to output register remainder and value of register Q[22:0] to output register result_div_ieee[22:0]

For negative numbers, the approach is little bit different. We convert the negative operand into its 2's complement form. 2's complement of any number is determined by taking complement of the number and then adding 1 to that number. If both of the numbers are negative, we perform normal NRD using the two numbers. But if only one of the operand is negative and other is positive then, following algorithm is carried out:

Step 1: Check if oper = 4'b0100

Step 2: Set the value of register A as 24 bit 0

Step 3: Set the value of register M as 2's compliment of the Divisor (24 bit)

Step 4: Set the value of register Q as Dividend (24 bit)

Step 5: Concatenate A with Q

Step 6: Perform the normal NRD using the positive number and the 2's complement of the negative number.

Step 7: If the remainder is not equal to zero, then perform:

- Increment the quotient by one.
- The value of the remainder is calculated using the formula

Remainder = divisor * quotient – dividend (all three are positive)

Step 8: Finally set the sign bit of the quotient as 1.

Step 9: Check for division by zero exception as in section 2.3.4.1

Step 10: Assign value of register A to output register remainder and value of register Q[22:0] to output register result_div_ieee[22:0]

2.6 EFFICIENT SHIFTING ALGORITHM

Barrel shifters are a combinational logic circuit that can shift a data input in a single clock cycle. It has three inputs i.e. the number to be shifted (32 bit register op1), the direction where the number is shifted (1 bit register direction-1 for left and 0 for right) and the value by which the input number is shifted (5 bit register shift_val) and one output (32 bit register result) giving the value after the input number is shifted to the direction by the input value.

The algorithm used for shifting operation is described as follows:

Step 1: Check if the oper is 4 bit 0101.

Step 2: Do the following for n number of times (n is the number of bits in shift_val)

- Check the MSB of the bit 5 of the register shift_valIf it is 1, we copy bits [15:0] of register op1 and save it in bits [31:16] of register result and rest [15:0] as 0 if direction is 1shift and if direction is 0, copy the 0 bit from bits [31:16] of register result and rest part will consist of the [31:16] bits of the op1.
- If it is 0, we do not alter anything and use the same value for next iteration

If the shift_val is 01000, as normal shift operator (>>, << or >>>) does 1 bit shifting per clock cycle, it will take 8 clock cycle to complete the shifting. But our algorithm shifts the operand in a single clock cycle as it directly copies bits [23:0] of register op1 to bits [31:8] of result and rest bits of register result are assigned 0 for left shift and copies bits [31:8] of register op1 to bits [23:0] of register result and rest bits of result are assigned 0 for right shift, in a single clock cycle. Thus our algorithm is time efficient.

2.7 EFFICIENT SQUARE ROOT DETERMINATION ALGORITHM

The non-restoring square root determination algorithm focuses on the "partial remainder" with every iteration and not on "each bit of the square root" [11]. At each iteration, this algorithm requires only one traditional adder or subtractor, i.e., it does not require other hardware components, such as multipliers, or even multiplexors. It generates the correct result even for the last bit position. Based on the result of the last bit, a precise remainder is obtained immediately without any addition or correction operation. It can be implemented at very fast clock rate as it has very simple operations at each iteration [12]. The algorithm is described as follows:

Initial condition:

- Set value of register Remainder as value 0
- Set the value of register Quotient as value 0
- Set the register D as the value of the number whose square root is to be obtained

Do the following for n 15 till n value decreases to 0 (Done for every root bit)

Step 1: If the value of register Remainder is greater than or equal to 0, do

- Set the value of register Remainder as (Remainder<<2)|((D>>(i+1))&3)
- Then set the value of register Remainder as Remainder–((Quotient<<2)|1)

Step 2: Else do

- Set the value of register Remainder as (Remainder<<2)|((D >>(i+1))&3)
- Then set the value of register Remainder as Remainder+((Quotient<<2)|3)

Step 3: If the value of register Remainder is greater than or equal to 0 then do

- Set the value of Quotient as ((Quotient<<1)|1)

Step 4: Else do

- Set the value of Quotient as ((Quotient<<1)|0)

Step 5: If the value of register Remainder is less than 0 then do,

- Set the value of register Remainder as Remainder+((Quotient<<1)|1)

Finally the value of square root is obtained from the register Q and the value of remainder is obtained from the register Remainder. The algorithm is generating a correct bit of result in each iteration including the last one. For each iteration addition or subtraction is based on the sign of the result obtained from previous iteration. The partial remainder is generated in each iteration which is used in the successive iteration even if it is negative (satisfying the meaning of non-restoring our new algorithm). In the last iteration, if the partial remainder is positive, it will become the final remainder. Otherwise, we can get the final remainder by addition to the partial remainder.

CHAPTER 4

TRANSCENDENTAL FUNCTIONS

Efficient Trigonometric Algorithm

A **transcendental function** is a function whose coefficients are themselves polynomials and which does not satisfy any polynomial equation. In other words, it is a function that transcends the algebra in the sense that it is not able to express itself in terms of any finite sequence of the algebraic operations like addition, multiplication, and root extraction. Examples of this function may include the exponential function, the logarithm, and the trigonometric functions. In the approach of developing an efficient FPU, we have tried to implement some transcendental functions such as sine function, cosine and tangential functions. The operation involves usage of large memory storage, has large number of clock cycles and needs expensive hardware organization. To reduce the effect of the above mentioned disadvantages, we have implemented CORDIC algorithm [13]. It is an effective algorithm to be used in our FPU as it can fulfill the requirements of rotating a real and an imaginary pair of a numbers at any angle and uses only bit-shift operations and additions and subtractions operation to compute any functions.

4.1 EFFICIENT TRIGONOMETRIC ALGORITHM

Evaluation of trigonometric value viz. sine, cosine and tangent is generally a complex operation which requires a lot of memory, has complex algorithms, and requires large number of clock cycles with expensive hardware organization. So usually it is implemented in terms of libraries. But the algorithm that we use here is absolutely simple, with very low memory requirements, faster calculation and commendable precision which use only bit-shift operations and additions and subtractions operation to compute any functions.

4.1.1 CORDIC FUNCTION

CORDIC (COordinate Rotation DIgital Computer algorithm) is a hardware efficient algorithm [14]. It is iterative in nature and is implemented in terms of Rotation Matrix. It can perform a rotation with the help of a series of incremental rotation angles each of which is performed by a shift and add/sub operation. The basic ideas that is incorporated is that -

- It embeds elementary function calculation as a generalized rotation step.
- Uses incremental rotation angles.
- Each of these basic rotation is performed by shift or and/sub operation

Principles of calculation-

- If we rotate point (1,0) by angle Ø then the coordinates say (X,Y) will be

  $X= \cos Ø$ and $Y= \sin Ø$

- Now if we rotate (X.Y) we get say (X´, Y´), then it is expressed as-

  $X´= X.\cos Ø - Y.\sin Ø$

  $Y´= Y.\cos Ø + X.\sin Ø$

- Rearranging the same-

  $X´= \cos Ø\,[X - Y.\tan Ø]$

  $Y´= \cos Ø\,[Y + X.\tan Ø]$

  Where tan is calculated as steps-

  $\tan Ø = \pm 2^{-I}$

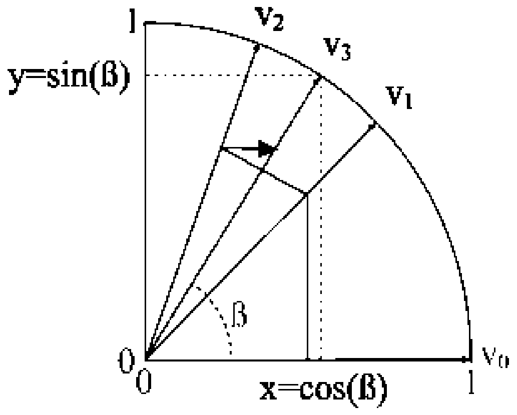

Figure 4.1 Cordic Angle Determination [15]

So, basically CORDIC is an efficient algorithm where we would not prefer use of a hardware based multiplier and we intend to save gates as in FPGA.

Now, since our conventional input is in degrees we built a look-up table in degrees. We are working towards a 12-bit precision structure. Moreover since all our floating point numbers have been converted to integers thus we satisfy the criteria of fixed-point format. But since our calculations are all integer based we need a look-up table that is integral in nature. So we multiply the values in table by a value = 2048 (= 2^{11} as we need a precision of 12 bits). So our look-up table is as follows-

Index	Ø	Ø * 2048
0	45	92160
1	26.565°	54395
2	14.036°	28672
3	7.125°	14592
4	3.576°	7824
5	1.789°	3664
6	0.895°	1833
7	0.4476°	917
8	0.2241°	459
9	0.1123°	230
10	0.0561°	115
11	0.0278°	57

Table 4.1 Look Up Table

We will assume a 12-step system so that it will yield 12 bits of accuracy in the final answer. Note that the Cos Ø constant for a 12 step algorithm is 0.60725. We also assume that the 12 values of Atan ($1/2^i$) have been calculated before run time and stored along with the rest of the algorithm. If true FP operations are used then the shift operations must be modified to divide by 2 operations.

4.1.2 INITIAL APPROACH:

The initialization specifies the total angle of rotation and sets the initial value of the point at (1,0) and multiplied by the constant 0.60725.

- Set register A to the desired angle.
- Set register Y to value 0
- Set register X to value 0.60725

4.1.2.1 COMPUTATION

The algorithm is described below. Do the following for i<12 times:

Step 1: Set dx to value after shifting X right by i places (It effectively calculates X*tan Ø for this step)

Step 2: Set dy to value after shifting Y right by i places (effectively calculates Y*tan Ø for this step)

Step 3: Set da to value Atan ($1/2^i$) (From the small lookup table)

Step 4: if value of A >= 0 (to decide if next rotation would be clockwise or anti-clockwise) then do,

- Set value of X to value of X - dy (to compute X-Y*Tan Ø)

- Set the value of Y to the value of Y + dx (To compute Y+X*Tan Ø)
- Set the value of A to the value of A - da (To update the current angle)

Step 5: if the values of A < 0 (to decide if next rotation would be clockwise or anti-clockwise) then do,

- Set value of X to value of X + dy (to compute X-Y*Tan Ø)
- Set the value of Y to the value of Y - dx (To compute Y+X*Tan Ø)
- Set the value of A to the value of A + da (To update the current angle)

The Sine of the desired angle is now present in the variable Y and the Cosine of the desired angle is in the variable X. This algorithm requires the use of non-integral numbers. This presents certain inconvenience so the algorithm is modified to work with only integral numbers. The modified algorithm is given below. As we have been working with an algorithm using 12 bits, our output angle ranges from –2048 to +2047. So, we will have to assume 16 bit calculations throughout.

4.1.3 EFFICIENT CORDIC IMPLEMENTATION

- Set register A to the desired angle*2048
- Set register Y to value 0
- Set register X to the value of 0.60725*2048
- Setup the lookup table to contain 2048*Atan ($1/2^i$)

4.1.3.1 COMPUTATION

The algorithm is described below. Do the following for i<12 times:

Step 1: Set the value of dx to the value of after shifting X right by i places (done to effectively calculate X*tan Ø)

Step 2: Set the value of dy to the value after shifting Y right by i places (done effectively to calculate Y*tan Ø)

Step 3: Set the value of da from the lookup ($1/2^i$) (From the small lookup table)

Step 4: if the value of A >= 0 (to decide if our next rotation is clockwise or anti clockwise), then do,

- Set the value of X to the value of X – dy (to compute value of X-Y*Tan Ø)
- Set the value of Y to the value of Y + dx (to compute value of Y+X*Tan Ø)
- Set the value of A to the value of A – da (to update the current angle)

Step 5: if the value of A < 0 (to decide if our next rotation is clockwise or anti clockwise), then do,

- Set the value of X to the value of X + dy (to compute value of X-Y*Tan Ø)
- Set the value of Y to the value of Y - dx (to compute value of Y+X*Tan Ø)
- Set the value of A to the value of A + da (to update the current angle)

The Sine of the desired angle is now present in the variable Y and the Cosine of the desired angle is in the variable X. These outputs are within the integer range –2048 to +2047.

Thus we have implemented an efficient algorithm for evaluating trigonometric functions that is absolutely simple, which incurs very low memory usage, which is faster in calculation and incorporates commendable precision which use only bit-shift operations and additions and subtractions operation to compute any functions.

CHAPTER 5

RESULTS & DISCUSSION

5.1 INTRODUCTION

In this chapter we analyze the results of simulation, RTL results and synthesis results for all the algorithms that we have implemented in our FPU. Then we compared the performance of our FPU to that of X87 family at similar clock speed. The synthesis was done in FPGA Spartan 3E Synthesizing Environment. The comparison is done with respect to

- Memory Requirement
- Gates Used
- Clock Cycle
- Complexity of the logic

5.2 SIMULATION RESULTS

The code was simulated in Xilinx 13.3. We have given some of the screen shots of the simulations that were obtained as a result of simulation in Xilinx software.

5.2.1 FLOAT TO INTEGER CONVERSION

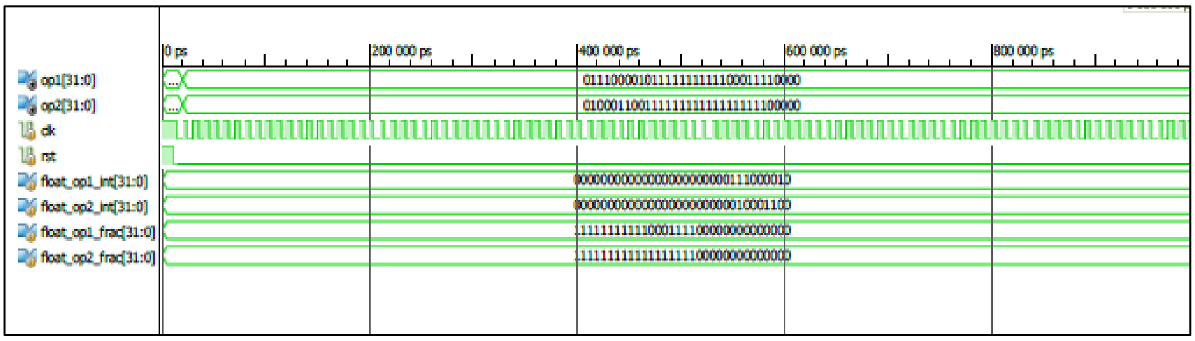

Figure 5.1 Float to Integer Conversion simulation result

5.2.2 ADDITION

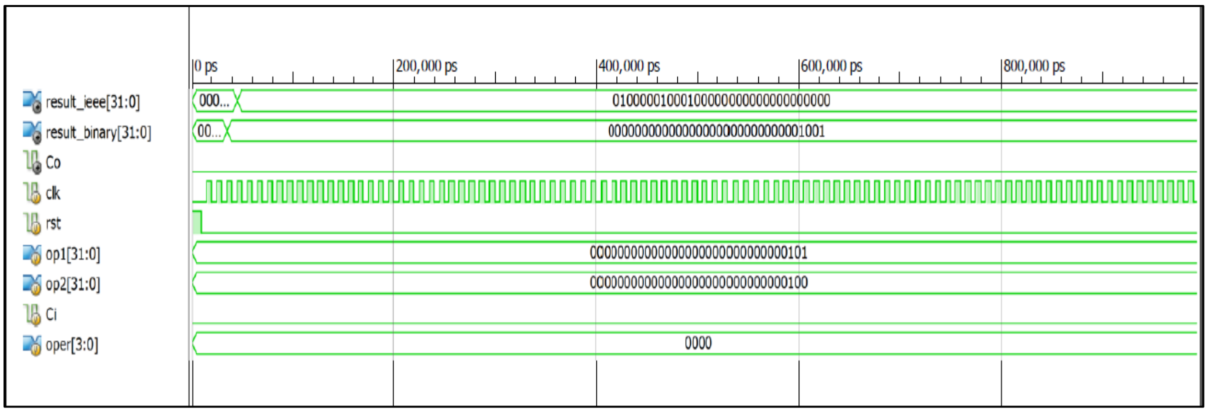

Figure 5.2 ADD simulation result

5.2.3 SUBTRACTION

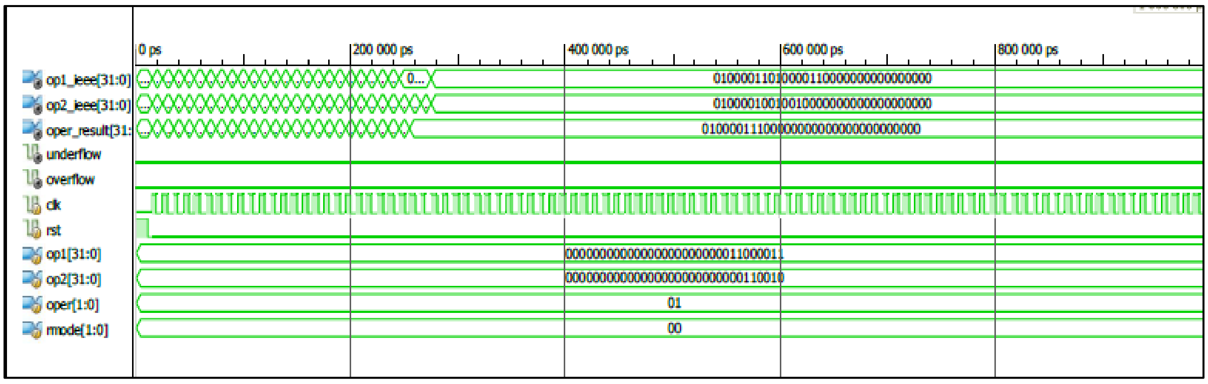

Figure 5.3 SUB simulation result

5.2.4 MULTIPLICATION

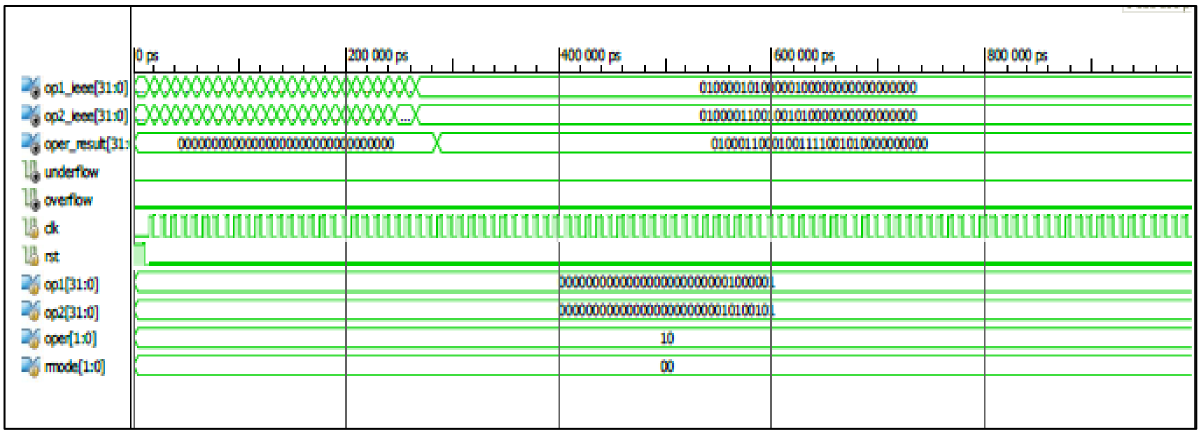

Figure 5.4 Multiplication simulation result

5.2.5 DIVISION

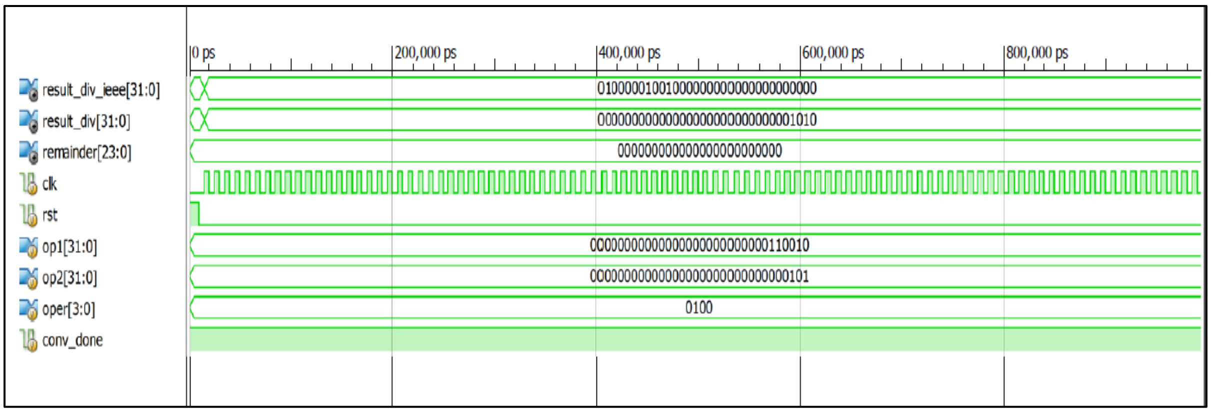

Figure 5.5 Division simulation result

5.2.6 SHIFTING

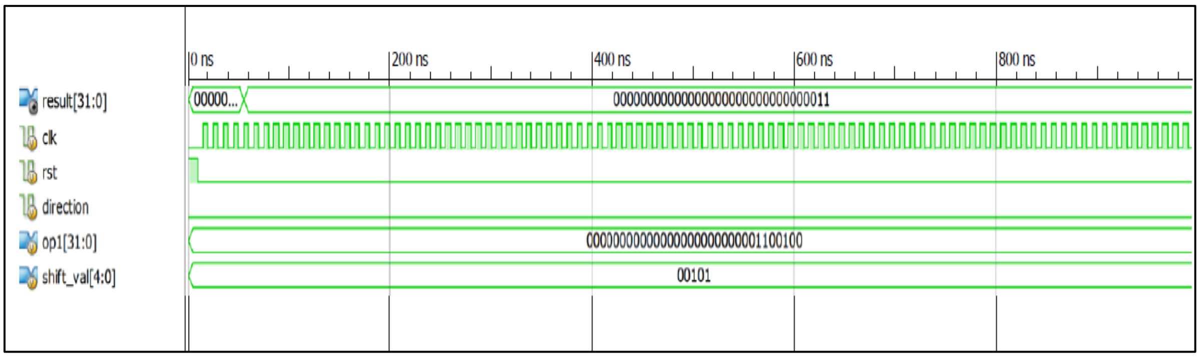

Figure 5.6 Shifting simulation result

5.2.7 SQUARE ROOT DETERMINATION

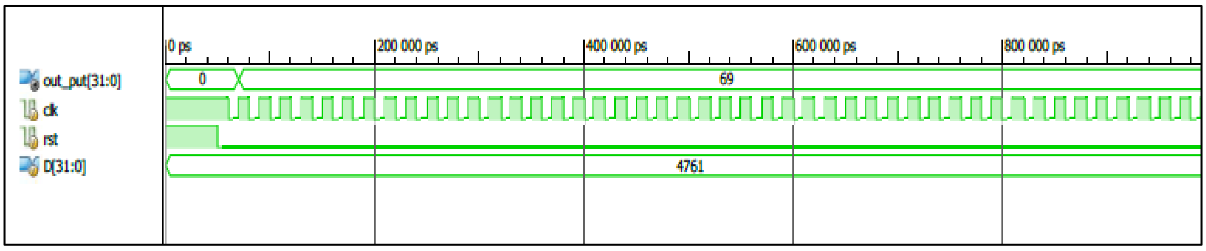

Figure 5.7 Square root simulation result

5.2.8TRIGONOMETRIC EVALUATION

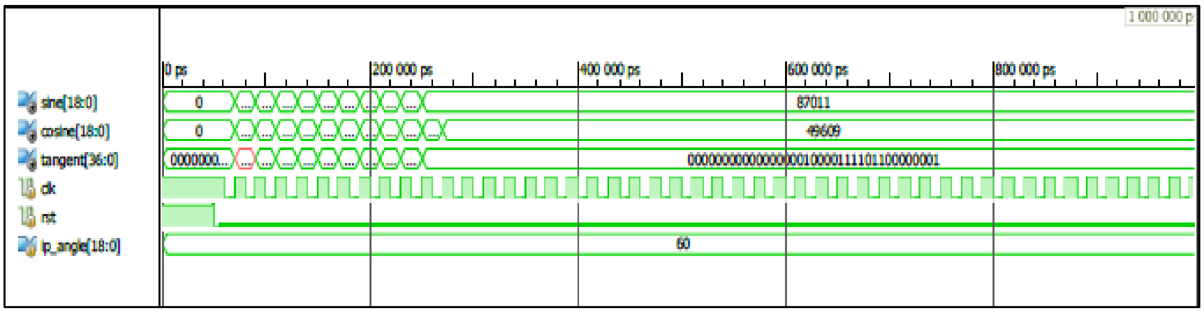

Figure 5.8 Trigonometric simulation result

5.3 SYNTHESIS RESULTS

After the simulation of the code was successful, we proceeded for the synthesis analysis. The simulation results gave a detailed description of the memory usage of the operation, i.e. the total number of registers required, total gates used, total multiplexers, LUTs, adders/subtractors, latches, comparator, flip--flops used. It also gives a detailed description of the device utilization summary, and detailed timing report which consists of time summary, timing constraints and delay. These details of the initial algorithm used which were discussed in chapter 2 were compared with that of the efficient algorithms discussed in chapter 3 and found that the efficient algorithms used less registers and gates. Number of IOs used was less in efficient algorithms and the delay were reduced too.

For an example, the addition algorithm which was implemented using block CLA adder with reduced fan-in was using less number of gates and registers than used by normal block CLA and delay was also reduced in CLA with reduced fan-in.

	Block CLA	Block CLA with reduced Fan-in
1 Bit Register	52	28
24-Bit Register	2	3
Flip-Flops	100	70
1 Bit XORs	24	2
24-Bit XORs	1	2
Number of IOs	136	96
Delay (ns)	8.040	4.734

Figure 5.1 Block CLA Vs. Block CLA with reduced fan-in

The synthesis report shows that the CLA with reduced fan-in is much more efficient than the normal CLA block algorithm. Thus proving the efficiency of the FPU designed.

According to the simulation and synthesis results, we have compared the performance of our FPU with that of X87 family (PENTIUM/MMX). The following table shows the result of comparison.

FPU	MAX CLK FREQ	DATA WIDTH	FADD /FSUB	FMUL	FDIV	FSQRT	FSIN /FCOS
PENTIUM /MMX	160-300 MHz	8 bit	1-3	1-3	39-40	70	17-173
OUR FPU (12 bit precsion)	50-250 MHz	32 bit	2-3	2-3	72	75-80	31

Figure 5.2 OUR FPU Vs. PENTIUM/MMX

CHAPTER 6

CONCLUSION

Conclusion

6.1 CONCLUSION

We have proved in the last chapter that the performance of our FPU was comparable to that of the X87 family (PENTIUM/MMX). The algorithm that we have used for the final FPU was comparable or even better in some case than the already existing efficient algorithms like in the case of block CLA and CLA with reduced fan-in in terms of memory used, delay, and device utilization. Because we have built the FPU using possible efficient algorithms with several changes incorporated at our ends as far as the scope permitted, all the unit functions are unique in certain aspects and given the right environment (in terms of higher memory or clock speed or data width better than the FPGA Spartan 3E synthesizing environment), these functions will tend to show comparable efficiency and speed and if pipelined then higher throughput may be obtained.

REFERENCES

[1] http://searchwinit.techtarget.com/definition/floating-point-unit

[2] Formal Verification of a Fully IEEE Compliant Floating Point Unit, Christian Jacobi

[5, 6] What Every Computer Scientist Should Know About Floating-Point Arithmetic, by DAVID GOLDBERG, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

[3] Open Floating Point Unit, The Free IP Cores Projects, Author: Rudolf Usselmann

[4, 7] GRFPU – High Performance IEEE754 Floating Point Unit, Edvin Catovic, Revised by: Jan Andersson Gaisler Research, Första Långatan 19, SE413 27 Göteborg, Sweden

[8] The Fastest Carry Lookahead Adder, Yu-Ting Pai and Yu-Kumg Chen, Department of Electronic Engineering, Huafan University

[9] Efficient Hardware Architectures for Modular Multiplication by David Narh Amanor

[10] Generalizations of the Karatsuba Algorithm for Efficient Implementations, Andr¶e Weimerskirch and Christof Paar, Communication Security Group, Department of Electrical Engineering & Information Sciences, Ruhr-UniversitÄat Bochum, Germany

[11] A New Non-Restoring Square Root Algorithm and Its VLSI Implementations, Yamin Li and Wanming Chu, Computer Architecture Laboratory, The University of Aizu

[12] Faster floating-point square root, for integer processors Claude-Pierre Jeannerod∗, Herv´e nochel † , Christophe Monat † , Member, IEEE, and Guillaume Revy∗ ∗ Laboratoire LIP (CNRS, ENSL, INRIA, UCBL)

[13] Sine/Cosine using CORDIC Algorithm, Prof. Kris Gaj, Gaurav, Doshi, Hiren Shah

[14] Compact and Efficient Generation of Trigonometric Functions using a CORDIC algorithm, Samuel Ginsberg, Cape Town, South Africa

[15] http://teal.gmu.edu/courses/ECE645/projects_S06/talks/CORDIC.pdf

1. Design Trade-Offs in Floating-Point Unit, Implementation for Embedded and Processing-In-Memory Systems, Taek-Jun Kwon, Jeff Sondeen, Jeff Draper, USC Information Sciences Institute

2. Efficient Algorithms for Fixed-Point Arithmetic Operations In An Embedded PIM, Jinwoo Suh, Dong-In Kang, and Stephen P. Crago, University of Southern California/Information Sciences Institute