INSTITUTE OF NATURAL AND APPLIED SCIENCES
UNIVERSITY OF CUKUROVA

Ph.D. THESIS

Metin Mete ÖZBİLEN

FLOATING-POINT HARDWARE DESIGNS FOR MULTIMEDIA PROCESSING

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ADANA, 2009

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ÇOKLU-ORTAM İŞLEME İÇİN KAYAN-NOKTA DONANIM TASARIMLARI

Metin Mete ÖZBİLEN

DOKTORA TEZİ
ELEKTRİK VE ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

Bu tez 08.07.2009 tarihinde aşağıdaki jüri üyeleri tarafından oybirliği ile kabul edilmiştir.

İmza.....
Doç.Dr. Mustafa GÖK
DANIŞMAN

İmza.....
Prof.Dr. Mehmet TÜMAY
ÜYE

İmza.....
Yrd.Doç.Dr Mutlu AVCI
ÜYE

İmza.....
Yrd.Doç.Dr. Ulus ÇEVİK
ÜYE

İmza.....
Yrd.Doç.Dr. Süleyman TOSUN
ÜYE

Bu tez Enstitümüz Elektrik ve Elektronik Mühendisliği Anabilim Dalında hazırlanmıştır.
Kod No:

Prof.Dr. Aziz ERTUNÇ
Enstitü Müdürü
İmza ve Mühür

Not: Bu tezde kullanılan özgün ve başka kaynaktan yapılan bildirişlerin, çizelge, şekil ve fotoğrafların kaynak gösterilmeden kullanımı, 5846 sayılı Fikir ve Sanat Eserleri Kanunundaki hükümlere tabidir.

Sevgili aileme,

ABSTRACT

Ph.D. THESIS

FLOATING-POINT HARDWARE DESIGNS FOR MULTIMEDIA PROCESSING

Metin Mete ÖZBİLEN

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
INSTITUTE OF NATURAL AND APPLIED SCIENCES
UNIVERSITY OF ÇUKUROVA

Supervisor: Assoc.Prof.Dr. Mustafa GÖK
Year: 2009, Pages: 120
Jury: Assoc.Prof.Dr. Mustafa GÖK
Prof.Dr. Mehmet TÜMAY
Assist.Prof.Dr. Mutlu AVCI
Assist.Prof.Dr. Ulus ÇEVİK
Assist.Prof.Dr. Süleyman TOSUN

In this dissertation floating-point arithmetic circuits for multimedia processing are designed. The arithmetic operations floating-point add, floating-point multiply, floating-point multiply-add and floating-point division are researched and specific hardware designs for them are implemented. The multimedia instructions are single instruction multi data (SIMD) type instructions. Hardware designs that perform operations on packed data increase the speed of the execution of floating-point multimedia instructions. In this dissertation, multiplication, addition, subtraction and reciprocal operations are speed up and additional functionalities are added using packet floating-point numbers.

Key Words: multimedia, hardware, design, floating-point, SIMD.

ÖZ

DOKTORA TEZİ

ÇOKLU-ORTAM İŞLEME İÇİN KAYAN-NOKTA DONANIM TASARIMLARI

Metin Mete ÖZBİLEN

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
ELEKTRİK VE ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

Danışman: Doç.Dr. Mustafa GÖK
Yıl: 2009, Sayfa: 120
Jüri: Doç.Dr. Mustafa GÖK
Prof.Dr. Mehmet TÜMAY
Yrd.Doç.Dr Mutlu AVCI
Yrd.Doç.Dr. Ulus ÇEVİK
Yrd.Doç.Dr. Süleyman TOSUN

Bu tezde çoklu ortamlarda için kayan-nokta aritmetik devreleri tasarımları yapılmıştır. Bu amaçla kayan-nokta toplama, kayan-nokta çarpma, kayan nokta çarp-topla ve kayan-nokta bölme aritmetik işlemleri araştırıldı ve özel donanım tasarımları gerçekleştirildi. Çoklu ortam yönergeleri tek yönerge çoklu veri tipi (SIMD) yönergeleridir. Paketlenmiş veri üzerinde işlem gerçekleştiren donanımlar kayan nokta çoklu ortam yönergelerinin işletilme hızını artırır. Bu tezde de çarpma, toplama çıkartma ve bire bölme işlemlerinde paketlenmiş kayan-nokta sayılar kullanılarak çoklu ortam işlemlerin gerçekleştirilmesinin hızlandırılması ve beraberinde fonksiyonel geliştirmeler sağlanmıştır.

Anahtar Kelimeler: çoklu-ortam, kayan-nokta, donanım, tasarım, SIMD.

TABLE OF CONTENTS **PAGE**

ABSTRACT I

ÖZ II

TABLE OF CONTENTS III

LIST OF TABLES VI

LIST OF FIGURES VII

1 INTRODUCTION 1

2 PREVIOUS RESEARCH 7

2.1 Floating Point Description 7

2.2 Floating Point Rounding 9

2.2.1 Round to Nearest Mode 9

2.2.2 Round to Positive-Infinity 10

2.2.3 Round to Negative-Infinity 10

2.2.4 Round to zero 10

2.3 Floating Point Special Cases 11

2.4 Floating Point Operations 11

2.4.1 Floating Point Addition and Subtraction 11

2.4.2 Floating Point Multiplication 13

2.4.3 Floating-Point Multiply-Add Fused (FPMAF) 17

2.4.4 Floating-Point Division 18

2.5 Floating-Point Packed Data 22

2.5.1 Packed Floating Point Addition and Subtraction 24

2.5.2 Packed Floating Point Multiplication 25

2.5.3 Packed Floating Point Division and Reciprocal 26

2.5.4 Packed Floating Point Multiply Add Fused(MAF) 27

2.6 Floating Point Packed Instruction Extensions 29

2.7 Benchmarking SIMD 31

2.8 Previous Packed Floating Point Designs 34
2.8.1 Packed Floating Point Multiplication Designs 34
2.8.2 Packed Floating Point Multiplier Add Fused Designs 37
2.9 Previous Patented Packed Floating Point Designs 39
2.9.1 Multiple-Precision MAF Algorithm 39
2.9.2 Shared Floating Point and SIMD 3D Multiplier 42
2.10 Method and Apparatus For Performing Multiply-Add Operation on Packed Data 44
2.11 Multiplier Structure Supporting Different Precision Multiplication Operations 47
2.12 Method and Apparatus for Calculating:Reciprocals and Reciprocal Square Roots 49

3 THE PROPOSED FLOATING POINT UNITS 51
3.1 The Multi-Precision Floating-Point Adder 51
3.2 The Single/Double Precision Floating-Point Multiplier Design 55
3.3 The Multi-Functional Double-Precision FPMAF Design 58
3.3.1 The Mantissas Preparation step 60
3.3.2 The Implementation Details for Multi-Functional Double-Precision *FPMAF* Design 65
3.4 Multi-Functional Quadruple-Precision *FPMAF* 70
3.4.1 The Preparation of Mantissas 71
3.4.2 The Implementation Details for The Multi-Functional Quadruple-Precision *FPMAF* Design 77
3.5 Multi-Precision Floating-Point Reciprocal Unit 81
3.5.1 Derivation of Initial Values 81
3.5.2 Newton-Raphson Iteration 83
3.5.3 The Implementation Details for Double/Single Precision Floating Reciprocal Unit 86

4 RESULTS 90
4.1 The Results for Multi-Precision Floating-Point Adder Design 90

4.2 The Results for Single/Double Precision Floating-Point Multiplier Design 91
4.3 The Results for Multi-functional Double-precision FPMAF design 92
4.4 The Results for Multi-Functional Quadruple-Precision *FPMAF* 95
4.5 The Multi-Precision Floating-Point Reciprocal Unit 97

5 CONCLUSIONS 99

BIBLIOGRAPHY 101

CURRICULUM VITAE 106

LIST OF TABLES **PAGE**

Table 2.1 Rounding Modes Examples 11
Table 2.2 Effective Operation 12
Table 2.3 Operations of Packed MAF 28
Table 2.4 Word-lengths in Single/Double Precision MAF 39
Table 2.5 Multiply-Accumulate Patent 46
Table 2.6 Packed Multiply-Add Patent 46
Table 2.7 Packed Multiply-Subtract Patent 46

Table 3.1 The Execution Modes 68
Table 3.2 The Logic Equations for The Generation of The Modified Mantissas . 73
Table 3.3 Quadruple Precision Execution Modes 78

Table 4.1 Area and Delay Estimates for Multi-Precision Floating Point Adder . 90
Table 4.2 Additional Components in Multi-Precision Adder Design 91
Table 4.3 Area and Delay Estimates for Single/Double-Precision Multiplier Design 92
Table 4.4 Additional Components in Single/Double-Precision Multiplier Design 92
Table 4.5 Area Estimates for Double-Precision *FPMAF* Design 93
Table 4.6 Delay Estimates for Double-Precision *FPMAF* Design 94
Table 4.7 Additional Components in Multi-Functional Double-Precision *FPMAF* Design 94
Table 4.8 Area Estimates for Quadruple-Precision *FPMAF* Design 96
Table 4.9 Delay Estimates for Quadruple-Precision *FPMAF* Design 96
Table 4.10 Additional Components in Multi-Functional Quadrable-Precision *FPMAF* Design 97
Table 4.11 The Comparison of the Standard and Proposed Reciprocal Design . . 97
Table 4.12 Additional Components in Multi-Precision Reciprocal Design 98

LIST OF FIGURES **PAGE**

Figure 1.1 SSID vs SIMD Structure 2

Figure 2.1 Floating Point Number Parts 7
Figure 2.2 Single and Double Precision Formats 8
Figure 2.3 Single Precision Floating Point Representation 9
Figure 2.4 Additional Bits Used for Rounding 13
Figure 2.5 Floating Point Adder/Subtracter 14
Figure 2.6 Floating Point Multiplier. 16
Figure 2.7 Floating-Point Multiply Add Fused. 19
Figure 2.8 Newton-Raphson Iteration. 21
Figure 2.9 Floating-Point Divider. 23
Figure 2.10 SIMD Type Data Alignment 23
Figure 2.11 SIMD Type Data Alignment Example 24
Figure 2.12 SIMD Addition Alignment Example 24
Figure 2.13 SIMD Addition Numerical Example. 25
Figure 2.14 SIMD Multiplication Alignment Example 25
Figure 2.15 SIMD Multiplication Numerical Example 26
Figure 2.16 SIMD Division Alignment Example 26
Figure 2.17 SIMD Reciprocal Numerical Example 27
Figure 2.18 SIMD Division Numerical Example 27
Figure 2.19 Packed Single Precision Floating Point Dot Product Results. 28
Figure 2.20 3DNow! Technology Floating-Point Data Type 29
Figure 2.21 SIMD Extensions, Register Layouts, and Data Types. 30
Figure 2.22 Motorola Altivec Vector Register. 30
Figure 2.23 Benchmark Result of with out SIMD and with SIMD. 35
Figure 2.24 Dual Mode Quadruple Precision Multiplier 36
Figure 2.25 The Divide-and-Conquer Technique 37
Figure 2.26 Two Single-Precision Numbers Packed in One Double-Precision Register 38
Figure 2.27 General structure of multipleprecision MAF unit 40
Figure 2.28 Shared Floating Point and SIMD 3D Multiplier 43

Figure 2.29 Multiply-Add Design for Packed Data 45
Figure 2.30 Multiplier Structure Supporting Different Precision Multiplication Operations 48
Figure 2.31 Reciprocal and Reciprocal Square Root Apparatus 50

Figure 3.1 The Alignments of Floating-Point Numbers in Multi-Precision Adder 52
Figure 3.2 The Block Diagram of Multi-Precision Floating-Point Adder 54
Figure 3.3 The Alignments for Double and Single Precision Numbers 56
Figure 3.4 The Multiplication Matrix for Single and Double Precision Mantissas 57
Figure 3.5 The Block Diagram for the Proposed Floating Point Multiplier 59
Figure 3.6 The Alignments of Double and Single Precision Floating-Point Operands in 64-bit Registers 61
Figure 3.7 The Partial Product Matrices Generated for (*DPM*) and (*SPM*) 63
Figure 3.8 The Matrix Generated for (*DOP*) Mode. 64
Figure 3.9 The Mantissa Modifier Unit in the Double Precision *FPMAF* 66
Figure 3.10 The Block Diagram for Multi-Functional Double Precision *FPMAF* Design. 67
Figure 3.11 The Alignments of Operands in 128-bit Registers 72
Figure 3.12 The Partial Product Matrices Generated for *SPM* Mode 75
Figure 3.13 The Matrix Generated for Single Precision Dot Product (*SDOP*) Mode 76
Figure 3.14 The Block Diagram for the Proposed Quadruple Precision *FPMAF* Design 82
Figure 3.15 Simple Reciprocal Unit that uses Newton-Raphson Method 84
Figure 3.16 Alignment of Double Precision and Single Precision Mantissas . . . 85
Figure 3.17 Multiplication Matrix for Single and Double Precision Mantissas . . 86
Figure 3.18 Alignment of Double and Single Precision Floating Point Numbers . 87
Figure 3.19 The proposed Single/Double Precision Reciprocal Unit 89

1. INTRODUCTION

Multimedia can be defined as multiples of media integrated together (Buford, 2007). The term media can be text, graphics, audio, animation, video or data. Other than media integration, multimedia sometimes used for interactive types of media like video games. Multimedia has become an important in industry, education and entertainment. The information from televisions, magazines, web pages to movies can be thought as multimedia streams. Advertising may be one of the largest industry using multimedia to convey their messages to people (Buford, 2007). Another popular use of multimedia is interactive education. Human beings can learn with their senses, especially with sight and hearing. A lecture that uses pictures and videos can help an individual learn and retain information much more effectively. Online learning applications replaces the physical contact of the teacher by multimedia content and offers more accessible learning environment.

One of the most popular multimedia application area is the graphics. At the beginning 2D graphics applications were considered quite satisfying. However new applications raised the bar to 3D graphics(Hillman, 1997). Engineering CAD(Computer Aided Design)/CAM(Computer Aided Manufacturing), scientific visualization and 3D animation becomes important aspects of multimedia. Graphic processing requires large computations that can be performed via specialized hardware in general purpose microprocessors extensions. Graphics processing applications are supported via extensions. These extensions consist of instructions that operate on packets of data. This type of instructions perform a single operation on all the data in the packet, which is known as SIMD.

SIMD instructions entered to personal computing world with Intel's MMX(Multimedia Extension) instructions added to the x86 instruction set (Lempel, Peleg, Weiser, 1997). Motorola introduces the Altivec instructions to PowerPC G3 and later a better version to PowerPC G4 processor (Diefendorff, Dubey, Hochsprung, Scale, 2000).

The term SIMD(Single Instruction Multiple Data) is a processor structure which a single instruction manipulates multi data structure. As it can be seen from the Figure 1.1, a SIMD processor uses a property of the data stream called *data parallelism*. When large amount of uniform data, that needs same instruction performed on it, requires data parallelism. For example an application which fit to SIMD operation is applying a filter

to an image. When a raster-based image has to filtered, the same filter has to be applied to all pixels of image. The computation of filter equations for each pixel is the same. That means there is single operation to be performed on multiple data.

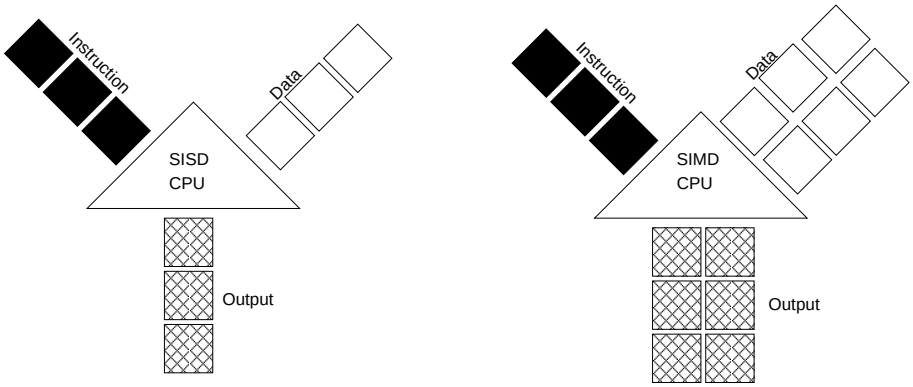


Figure 1.1 SSID vs SIMD Structure.

Today, many general-purpose processors have multimedia extensions that increase the performance for 3D applications. Processors from AMD(Advanced Micro Devices) support 3DNow! and 3DNow!+ (AMD, 2000). These extensions have additional 21 instruction for supporting packed floating point arithmetic and packed floating point comparison (Oberman, Favor, Weber, 1999). Intel has implemented SSE (Streaming SIMD Extension) since the Pentium 3 processor with support to SIMD single precision floating point operations, 64 bit integer SIMD operations also cache ability control, prefetch and instruction ordering operations The SSE2 and SSE3 were introduced with the Pentium 4 processor (Singhal, 2004) with support to on packed double-precision floating operations and packed byte,word doubleword and quadword operations and the SSE4 were introduced with the Core platform (Varghese, 2007), giving support to packed doubleword multiplies, floating-point dot products, simplify packed blending, packed integer operations, integer format conversions (Intel, 2007).

Another trend to increase the performance of graphics processing is the use of graphical processing units'(GPU) computational power (Macedonia, 2003). With the introduction of GeForce 256 processor from NVIDIA in 1999, graphics card's processor can be used as co-processor in graphics calculations. Since these cards are designed to execute fast graphics operations, they have high performance parallel processing units. The GeForce 3 has the first programmable vertex processor executing vertex shaders, along

with a configurable 32-bit floating-point fragment pipeline, programmed with Microsoft DirectX8 and OpenGL. The Radeon 9700, introduced in 2002, featured a programmable 24-bit floating- point pixel-fragment processor programmed with Microsoft Direct X9 (Charles, 2007) and OpenGL (Open Graphics Library) (Cole, 2005). The GeForce FX added 32-bit floating-point pixel-fragment processors. These GPUs has has a register-based instruction set including floating-point, integer, bit, conversion, transcendental, flow control, memory load/store,and texture operations. Floating-point and integer operations include add, multiply, multiply-add, minimum, maximum, compare, set predicate, and conversions between integer and floating-point numbers. (Lindholm, Nickolls, Oberman, Montrym, 2008). Recently, Nvidia has introduced CUDA(Compute Unified Device Architecture), which is a general purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA graphics processing units (GPUs) to solve many complex computational problems in a fraction of the time required on a CPU (Garland, Le Grand, Nickolls, Anderson, Hardwick, Morton, Phillips, Yao, Volkov, 2008).

This dissertation presents multi precision and multi functional floating point units that can be efficiently used in graphics processing. The cited previous work shows that there is a considerable research effort on increasing the performance of multimedia applications. Leading chip manufacturers introduces a new extension almost every year. The presented units also support dot product modes, which have never been implemented on any FPMAF(Floating Point Multiply Add Fused) design. The quad-precision FPMAF has two dot product modes: One of these modes performs two double-precision floating-point multiplications and adds their products with another double-precision floating-point operand; the other mode performs four single-precision floating-point multiplications and adds their products with an other single-precision floating-point operand. The double-precision FPMAF has only one dot product mode that performs two single-precision floating-point multiplications and adds their products with another single-precision floating-point operand. The proposed designs achieve significant hardware savings by supporting these functions in one unit instead of using a separate circuit for each mode.

The Dot product is also called scalar product. It takes to real number vectors and generate a real scalar value. It is inner product of orthonormal Euclidean space (Arfken,

1985). From the definition, dot product is very useful in geometric and physics calculation. Two and Three dimension computer graphics deal with both of them. Our design simplifies and also speed up these type of calculation. There exists instruction making similar calculation in todays popular processor multimedia extensions. The Intel Pentium 4 has single precision dot product instruction begin from the SSE 4 (Intel, 2007). The AMD processor has an accumulate multiplication in its 3Dnow multimedia extension which also do similar computation. (Amd, 2007)

A multi-precision floating-point adder design that overcomes the performance degradation caused by format conversion operations. The proposed multi-precision floating-point adder design can perform four half-precision (in NVIDIA format)(NVidia, 2007) floating-point additions or two-single precision floating-point additions or a single double-precision floating-point addition. In low-precision operation modes, the results are generated in parallel. A floating-point adder with the proposed functionality is not reported in the literature. Floating point addition is used in many places hence it is one of the most common operation. Packed floating point can speed up filtering operation of images by accessing multiple data. Both popular general purpose processors have packed single precision floating point addition instructions in their multimedia extension instructions sets (AMD and INTEL, 2007).

The following contributions are made by this dissertations:

- A multi-precision floating point adder/subtractor is designed that support half, single and double precision floating-point additions (Ozbilen, Gok, 2008). Compared to a single-precision floating point adder the proposed multi-precision design can compute four half precision or two single precision addition simultaneously. Therefore the performance of a single-precision addition can be doubled and half precision addition can be quadrupled with the proposed design. In addition to these advantages, to our best of knowledge, the proposed adder is the only multi-precision adder that supports half precision addition reported in the literature.

- A floating point multiplier design method that supports single and double precision multiplication is designed (Gok and Ozbilen, 2009b). Beside double precision multiplication, the proposed multiplier can simultaneously perform two single precision multiplication within the delay of a standard double precision multiplication.

One of the main advantage of the proposed design method is it can be applicable to all kind of floating point multipliers.

- A multi-precision floating point multiply add fused design method is introduced and using this method a double precision multiply add and a quadruple precision multiply add designs are implemented (Gok and Ozbilen, 2008). The proposed double precision multiply add fused supports single and double precisions multiply-add operations and single precision dot-product operation. The proposed quadruple precision multiply add fused supports single, double, and quadruple precision multiply add fused operations and single and double dot product operations. Compared to the previous state of the art double-precision multiply-add fused designs presented in (Huang, Shen, Dai, and Wang, 2007) and (Jessani and Putrino, 1998) and the proposed double-precision designs have the following advantages: The dot product operation mode which may double the performance of a matrix multiplication. Another novelty of this design is in dot product mode the rounding error is decreased since only one rounding operation is performed whereas a dot product operation with a multiply add design requires rounding as much as the number of iterations.

- The quadruple precision multiply add fused design is very rare in academic research though there exist recent designs by main chip manufacturers. Therefore the design is compared with a quadruple multiplier presented in (Akkas, Schulte, 2006) the proposed quad-MAF has 3% more area and approximately the same delay compared to the reference design however the functionality of the design far exceeds it.

- A floating point reciprocal unit design method that is based on the previous design methods is presented (Ozbilen, Gok, 2008). The double precision reciprocal units designed with this method supports two single precision reciprocal operation with nearly same delay. This unit can be also enhanced by coupling with proposed double precision multiply add fused unit to support division operation, divide and sum or divide by subtract. This design is compared with the design presented in (Kucukkabak, Akkas, 2004). Compared to the reference design the proposed design can perform two reciprocal operation in the same critical delay.

- In general, all the proposed designs overcomes the additional delay due to the format conversion. Format conversion adds extra delay to computation if a smaller precision operation is performed using a larger precision unit. In that case smaller precision operands are converted to the larger precision and after the operation the large precision result is converted back to the small precision format.

2. PREVIOUS RESEARCH

This section explains floating point number formats, floating point addition, subtraction, multiplication, multiplication-add fused, division and reciprocal operations and describes basic implementation methods for those operation. This section also presents some of the significant previous work on floating point circuits described for multimedia operations based on patents and/or research papers.

2.1 Floating Point Description

The floating point format is used to represent very big or very small real numbers in computers or calculators. A floating point number consists of three parts: A sign bit that shows whether the number is positive or negative, an exponent which represents the position of the radix point, and a mantissa which represents the digits of the number's magnitude. The sign, exponent and mantissa are placed as shown in Figure 2.1 where sign is the most significant bit. This placement makes comparison of the numbers easier.

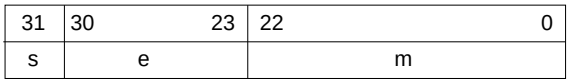


Figure 2.1 Floating Point Number Parts.

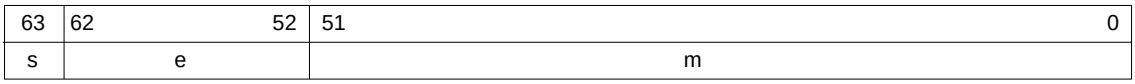
Since the acceptance of the IEEE standard in late 80s, floating point hardware in modern processors abide the rules dictated by IEEE-754 standard (IEEE, 1985). This has increased the portability of the floating-point applications. Due to general demand the standard is undergoing modifications (Microprocessor Standards Committee, 2006). The current draft of the standard can be accessed from ANSI(American National Standards Institute) -IEEE Standard 754. The main differences between the current draft and the IEEE-754 standard are the inclusion of decimal floating point number formats and quadruple precision format and exclusion of extended precision formats. The single and double precision formats are kept unchanged. The advantage of this notation is that the point can be placed so that long strings of leading or trailing zeros can be avoided. The specific place for the point is typically just after the leftmost nonzero digit. Because of this the leftmost digit of the significant can't be zero. This is called *normalization*. So,

there is no need to express the point explicitly which is hidden. Popular general purpose processors such as, the Intel Pentium and the Motorola 68000 series provide 80 bit extended precision format, which has 15 bit exponent and 64 bit mantissa, with no hidden bit.

The IEEE-754 standard has two different precision types: the single, which has 32 bits data width, with 8 bit exponent and 23 bit mantissa and the double, which has 64 bits data width, with 11 bit exponent and 52 bit mantissa. The single and double formats are shown in Figure 2.2.



(a) Single Precision



(b) Double Precision

Figure 2.2 Single and Double Precision Formats

The exponent is biased by $2^{8-1}-1=127$, so that exponent's range is -126 to +127. For the normalized numbers, the number has value

$$V = s \times 2^e \times 1.m \tag{2.1}$$

where

$s = +1$ for positive numbers when the sign bit is 0

$s = -1$ for negative numbers when the sign bit is 1

$e = exponent - 127$ exponent is stored with 127 added to it, also called biased with 127.

$m =$ the mantissa with leading one, where $1 \leq 1.m < 2$

Since both formats have finite area for representing real numbers, the numbers have to be approximated while they are converted to floating-point representation. Through out the text IEEE-754 format floating point numbers are referred as floating-point.

The single-precision format representation of 0.156255 is shown in Figure 2.3.

S	Exponents (8 Bits)	Mantissa (23 Bits)
0	0 1 1 1 1 1 0 0	0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
31	30 23	22 00

Figure 2.3 Single Precision Floating Point Representation of Real Number 0.15625

2.2 Floating Point Rounding

Floating point numbers is used to represent the real numbers, but sometimes these numbers can not be represented exactly. In this case the floating point number is rounded. For example the real number 0.1 can not be represented exactly in IEEE-754 format (IEEE, 1985).

$$0.1 = 00011001100110011001100... \tag{2.2}$$

When it is rounded to single precision format it is represented as

$$s = 10011001100110011001100 \quad e = 01111011\text{(-4)} \quad s = 0 \tag{2.3}$$

The exact decimal value after conversion is

$$s = 0.09999994 \tag{2.4}$$

The difference between two consecutive floating-point numbers which have same exponent is called unit in last place (*ulp*). For numbers which has exponent of 0, an *ulp* is exactly 2^{-23} or about 10^{-7} in single precision, and about 10^{-16} in double precision.The IEEE-754 standard has four rounding modes: Round to nearest even, round up or round toward positive infinity, round down or round to negative infinity and round toward zero. The IEEE-754 standard accepts round-to-nearest even as default rounding for all fundamental algebraic operations (IEEE, 1985). Consider a floating number, x is between to real numbers R_1 and R_2, that means $R_1 \leq x \leq R_2$, has to be rounded.

2.2.1 Round to Nearest Mode

In this mode, the inexact result is rounded to the nearer of the two adajent values. If the result is in the middle, then the even alternative is chosen. This rounding is also known

as *round to even*. It can be formulated as

$$Rnd\,(x) = \begin{cases} R_1 & \text{if } |x-R_1| < |x-R_2| \\ R_2 & \text{if } |x-R_1| > |x-R_2| \\ Even(R_1,R_2) & \text{if } |x-R_1| = |x-R_2| \end{cases} \tag{2.5}$$

For example, 0.016 is rounded to 0.02 because the next digit '6' is 6 or more; 0.013 is rounded to 0.01 because the next digit '3' is 4 or less; 0.015 is rounded to 0.02 because the next digit is 5, and the hundredths digit '1' is odd; 0.045 is rounded to 0.04 because the next digit is 5, and the hundredths digit '4' is even; 0.04501 is rounded 3.05 because the next digit is 5, but it is followed by non-zero digits.

2.2.2 Round to Positive-Infinity

This mode rounds inexact results to the possible value closer to positive infinity. It can be formulated as

$$Rnd\,(x) = R_2 \tag{2.6}$$

For example, 0.016 rounded to hundredths is 0.02; 0.013 rounded to hundredths is 0.02.

2.2.3 Round to Negative-Infinity

This mode rounds inexact results to the possible value closer to negative infinity. It can be formulated as

$$Rnd\,(x) = R_1 \tag{2.7}$$

For example, 0.016 rounded to hundredths is 0.01; 0.013 rounded to hundredths is 0.01.

2.2.4 Round to zero

This mode rounds inexact results to the possible value closer to zero. In other way the result is truncated. It can be formulated as

$$Rnd\,(x) = \begin{cases} R_1 & \text{if } x \geq 0 \\ R_2 & \text{if } x \leq 0 \end{cases} \tag{2.8}$$

For example, 0.016 rounded to hundredths is 0.0;

Examples for rounding modes are summarized in the Table 2.1. The real number 1.0016 is digitized to 40 bits and its rounded to single precision 23 bits versions are given in binary and decimal.

Table 2.1 Rounding Modes Examples

No Round	1000000000110100011011011100010111010 11	1.0016
Round-to-Nearest	10000000001101000110111 0	1.0016
Round-to-Positive Infinity	10000000001101000110111 0	1.0016
Round-to-Negative Infinity	10000000001101000110110 1	1.0015999
Round-to-Zero	10000000001101000110110 1	1.0015999

2.3 Floating Point Special Cases

Following special are cases usually indicated by flags for floating-point operations: *Overflow*, when the exponent is incremented during normalization and rounding step. If exponent $E \geq 255$, the overflow flag is set and result is set to $\pm\infty$. *Underflow*, when the exponent is decremented during normalization and if exponent $E = 0$, the underflow flag is set and fraction is left unnormalized. *Zero*, when the mantissa is zero, $E = 0$ and $F = 0$ then zero flag is set. *Inexact*, when the guard bits are all one than inexact flag is set. Not a number (*NAN*), when one of the operand or both is a NAN, then result is set to NAN.

2.4 Floating Point Operations

2.4.1 Floating Point Addition and Subtraction

The most popular floating point operation is floating point addition. The addition of two floating point numbers $X = S_x \cdot 2^{E_x} \cdot M_x$ and $Y = S_y \cdot 2^{E_y} \cdot M_y$ can be formulated as

$$M_z = \begin{cases} (-1)^{S_x} \cdot M_x \pm \left((-1)^{S_y} \cdot M_y \times 2^{(E_y - E_x)} \right) & \text{if } E_x \geq E_y \\ \left((-1)^{S_x} \cdot M_x \times 2^{(E_y - E_x)} \right) \pm (-1)^{S_y} \cdot M_y & \text{if } E_x < E_y \end{cases} \tag{2.9}$$

$$E_z = \max(E_x, E_y) \tag{2.10}$$

where $Z = S_z \cdot 2^{E_z} \cdot M_z$ is the result.

The floating point addition operation begins with equalization of the exponents of operands. The number with small exponent is equalized by right shifting the mantissa while increasing the exponent by one with each shift. This operation is known as *alignment*. After alignment of the mantissas the effective operation takes place. The effective operation base on the signs is shown in Table 2.2. The exponent of the result is chosen

Table 2.2 Effective Operation

Floating-Point Operation	**Signs of Operation**	**Effective Operation(EOP)**
Add	equal	add
Add	different	subtract
Subtract	equal	subtract
Subtract	different	add

from one of the equalized exponents. The sign of the result is determined by selecting largest operand. After the operation the result might require normalization operation. There results might be in one of these three forms:

1. The result is already normalized.
2. When the effective operation is addition, there might be an overflow in the mantissa.
3. When the effective operation is subtraction, there might be leading zeros.

For the second and third forms of result has to be normalized and according to the normalization shift amount the exponent has to be updated. Leading One Dedector (LOD) determines the position of the leading one in the result.

After normalization and exponent update, rounding takes place for the result. The alignment of mantissa may increase the operand size of the result. To obtain correct result only three additional fractional bits are sufficient. These bits are called *guard bits*. They are guard (G), round (R), and sticky (T), which are shown in Figure 2.4, where F denotes the fractional part of mantissa. In *Round to nearest even* mode result is rounded up if $G = 1$ and R and T are not both 0 and round to even if $G = 1$ and $R = T = 0$. In *Round towards zero* mode result is truncated. In *Round toward positive infinity* mode result is

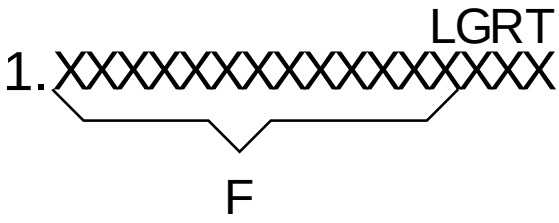


Figure 2.4 Additional Bits Used for Rounding

rounded up if *G*,*R*, and T are not all zero. In *Round toward positive infinity* mode result is rounded up if *G*,*R*, and T are all zero.

The basic floating point adder is shown as a block diagram in Figure 2.5. The function of each block is explained as follows. The *Exponent Difference* unit computes difference of the exponents. The sign bit of the difference is used to select the greatest exponent which realizes Equation 2.10. This sign bit is also used by the *Swap* unit to decide which number has to be aligned. The *EOP* unit performs the effective operation given in Table 2.2. The *Alignment* unit right shifts by d digits. The *Add/Sub* unit performs the effective operation. The *Normalization* unit performs normalization based on the value generated by *LZA* unit. The *LZA* unit anticipates the number of leading zeros. The normalized result is rounded by the *Round* unit and the mantissa of the result is generated. Based on the *ovf* signal *Exponent Update* unit increments the exponent value and the exponent of the result is generated. The *Sign* unit determines the sign of operation depending on input signs and result of effective operation.

2.4.2 Floating Point Multiplication

Floating point multiplication is another popular operation used in floating point operations. The floating point multiplication is performed for floating point numbers x and y and product z as

$$M_z = 1.M_x \times 1.M_y \tag{2.11}$$

$$E_z = E_x + E_y \tag{2.12}$$

$$S_z = S_x \otimes S_y \tag{2.13}$$

where M_x, M_y, and M_z are mantissas, E_x, E_y, and E_z are exponents and S_x, S_y, and S_z are signs of the operands X, Y, and the result Z, respectively.

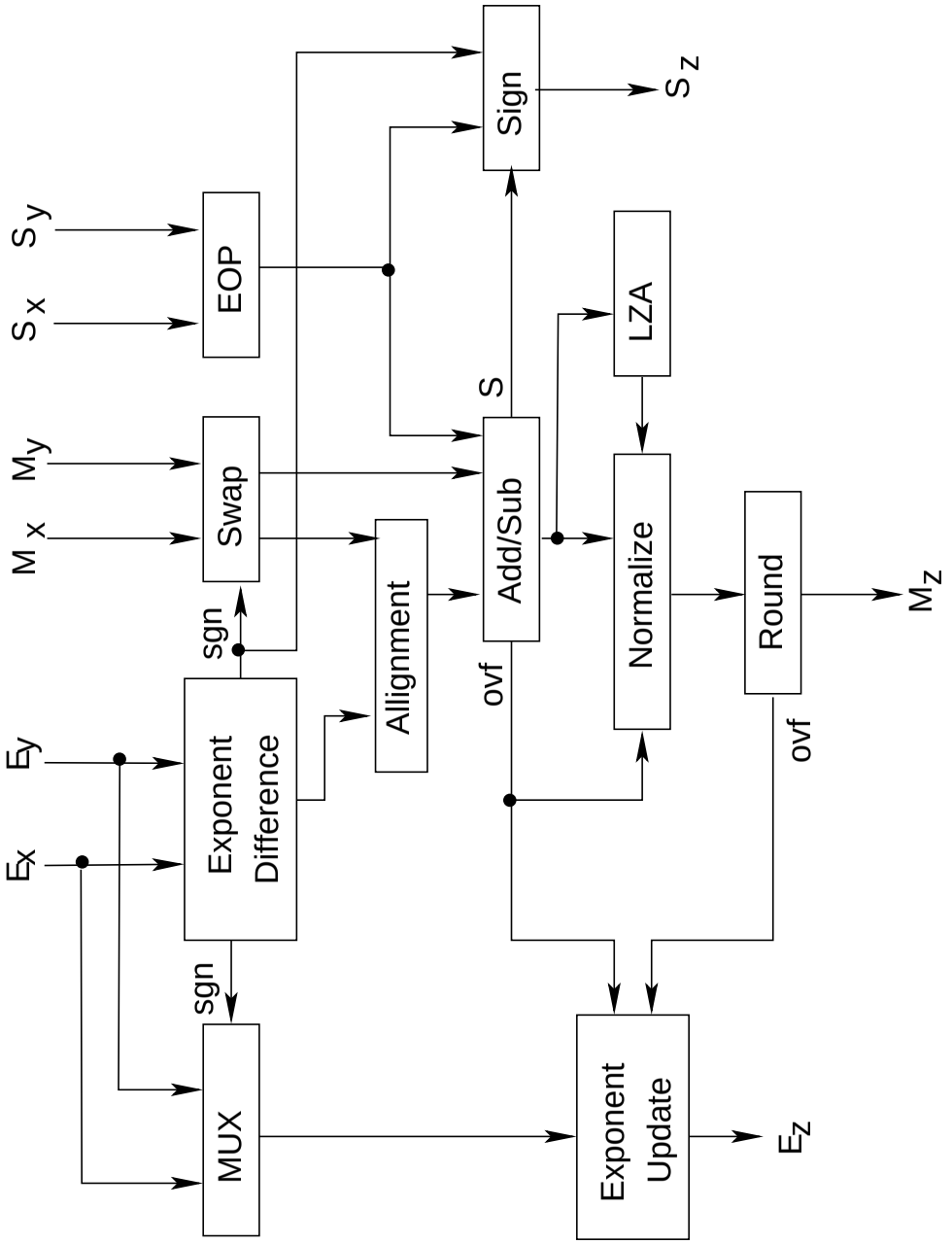


Figure 2.5: Floating Point Adder/Subtracter.

The computation of Equations 2.11-2.13 can be performed in parallel. The addition of the exponents in biased representation is performed by adding the exponents and subtracting the extra bias that comes from second operand. The operation is expressed as

$$E_{B,z} = E_{B,x} + E_{B,y} - B \tag{2.14}$$

where B is the bias value. Exponent addition can be performed by using fast carry propagate adder (CPA) (Koren, 2002).

The sign of the result is evaluated with an XOR gate. The mantissa multiplication is usually performed by a fast parallel multiplier. Some of the popular multipliers used in mantissa multiplication are unsigned-radix-2, signed Baugh-Wooley (Baugh and Wooley, 1973) and signed Booth (Booth, 1951). These methods are used for generating multiplication matrix. Then these values are reduced to carry-save vectors using reduction methods like Wallace (Wallace, 1964) or Dadda (Dadda, 1965) reduction. The final result is obtained by using a final CPA. The multiplication of n-bit mantissas generates a $2n$-bit product, P. But, only the n-bits are needed in results, others are used in generation of guard bits. The sticky-bit is computed in parallel with the multiplication. The $n-2$ least significant bits of P are not returned as a part of the rounded P, but for rounding it is important to know if any of the discarded bits is a one. The sticky-bit represents this situation (Gok ve Ozbilen, 2008). The trivial method for generating sticky simply ORs all the $n-2$ least significant bits of P. The sticky-bit can also be determined from the second half of the carry-save representation of the product (Bewick, 1994; Yu and Zyner, 1995). In Bewicks design a 1 is added into the partial product tree, and later is corrected during the addition of sum and carry vectors by setting the carry-in input of the CPA to one (Bewick, 1994). Yu and Zyner presented a method that determines whether the sum of sum and carry vectors is a zero, without performing a carry- propagate addition (Yu and Zyner, 1995).

After the multiplication step, normalization of the mantissa step is performed. When $M_x \geq 1$ and $M_y < 2$,the result is in the range $[1,4)$ so a normalization by shifting right one position might be needed. There is no left shift normalization is needed in floating point multiplication. Mantissa is rounded as in the floating point addition.

The block diagram of a simple floating point multiplier can be seen in Figure 2.6. In the figure, the *Exponent Addition* unit computes the Equation 2.14. The *Multiplier*

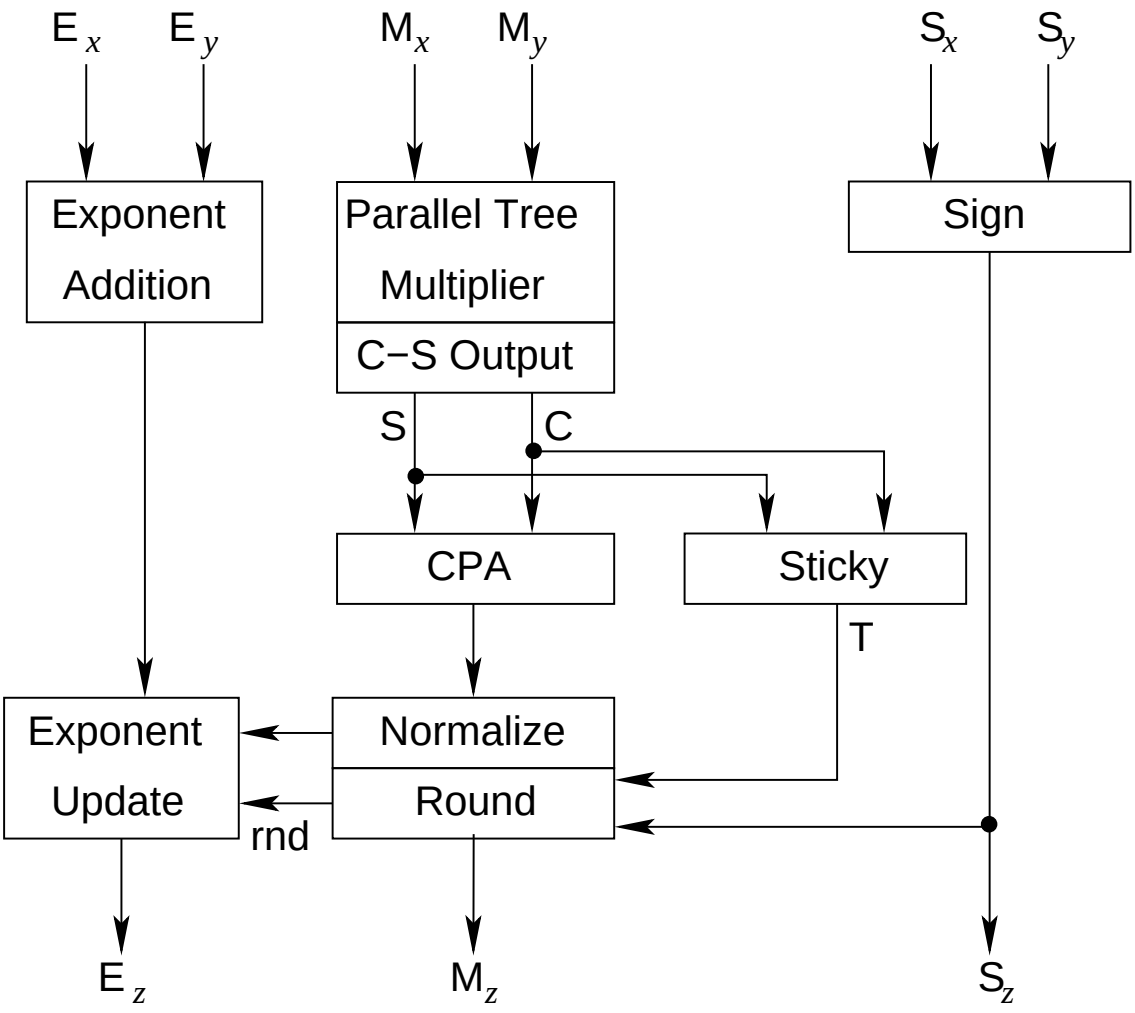


Figure 2.6 Floating Point Multiplier.

generates the product of the mantissas in carry-save format. The sign of the multiplication is computed by an XOR gate in the *Sign* unit. (Gurkayna, Leblebicit, Chaouati, McGuinness, 2000; Beaumont-Smith and Lim, 2001), Carry-Lookahead Adders (Yu-Ting and Yu-Kumg, 2004; Fu-Chiung, Unger and Theobald, 2000; Wang, Jullien, Miller and Wang, 1993) or Carry Skip Adder (Min and Swartzlander, 2000; Chirca, Schulte, Glossner, Horan, Mamidi, Balzola, Vassiliadis, 2004). At the same time these vectors are used by the *Sticky* unit for sticky bit computation. After the unnormalized results are normalized in the *Normalization* unit, the *Round* unit performs rounding. The *Exponent update* unit updates the exponent depending on the normalization and rounding operations (Even, Mueller and Seidel, 1997; Gok, 2007; Even and Seidel, 2000; Quach, Takagi and Flynn, 2004).

2.4.3 Floating-Point Multiply-Add Fused (FPMAF)

The FPMAF unit calculates

$$Z = (X \times Y) + W \tag{2.15}$$

where X,Y, W and Z are the operands represented with (M_x,E_x),(M_y,E_y) and (M_w,E_w) respectively and result Z is represented with (M_z,E_z). All the mantissas are signed and normalized. This reduces the number of interconnections between units and provides accuracy more than separate multiply and add units. The accuracy comes from single normalization and rounding step instead of two. The FPMAF can be also used to perform addition and multiplication by setting $Y = 1.0$ or $W = 0.0$, respectively (Ercegovac and Lang, 2004).Floating-Point multiplication add fused operation is defined as

$$M_z = (-1)^{(S_x \oplus S_y)} \cdot 1.M_x \times 1.M_y + (-1)^{S_w} \cdot M_w \cdot 2^{(E_x + E_y - B - E_w)} \tag{2.16}$$

$$E_z = max(E_x + E_y - B, E_w) \tag{2.17}$$

where the operands $X = S_x \cdot 2^{E_x} \cdot M_x$, $Y = S_y \cdot 2^{E_y} \cdot M_y$ and $W = S_w \cdot 2^{E_w} \cdot M_w$. B is bias value.

Mantissa multiplication of M_x and M_y is performed by a fast parallel multiplier similar to Floating-Point multiplication. Addition of exponents E_x and E_y and determination of alignment shift for operand M_w for biased exponents can be expressed in Equation 2.18

$$d = E_x + E_y - E_w - B + m + 3 \tag{2.18}$$

where d is distance, B is bias value, m is 1+length of fractional part, 3 is for extra guard bits.

The main part of FPMAF is the mantissa multiplier. After the generation of multiplication matrix and reducing them to carry and sum vectors. The final adder can be modified to add a third floating point number(W). This addition can be realized with a Carry-Save-Adder (CSA) and a Carry-Propagate-Adder (CPA) (Harris and Sutherland, 2003).

The alignment of the W can be performed in parallel with the multiplication of mantissas. The size of the shifter is $3m+2$ bits. $2m$ comes from the result of multiplication and $1m$ from the third floating point number. There are 2 more bits that can be used as

guard-bits. To avoid bidirectional shift operation. The addend is positioned at $m+3$ bit left to the product in the shifter. So, only right shifting is performed when necessary.

The $3m+2$-bit 3-2 Carry-Save-Adder (CSA) is used for addition of $2m$-bit Carry and Save vectors produced by multiplier with aligned M_w. The unnormalized resultant mantissa is obtained after 2-1 carry propagate adder (CPA). Since the leftmost $m+2$ bits of adder input are always 0, the adder can be divided into an adder and an incrementer. The normalization of FPMAF is performed as in Floating- Point Addition. The leading one detector locates the position of one. The left shifter can shift up to $2m$ position. Additional m positions comes form initial position of adder operands. The exponent is updated based on the shift amount. Rounding of mantissa is performed after normalization. The rounding is performed as in Floating-Point Addition with out any change. The determination of special values of Floating-Point Addition with out any change is applicable to FPMAF design.The FPMAFs are usually pipelined to increase the throughput. A typical pipelined FPMAF design, which has 3 stages is shown in Figure 2.7.

The description of the functional blocks is explained as follows: The *multiplication matrix* unit generates the products in parallel with the alignment of W. *Distance* unit computes the right shift amount d. Also the exponent with greater value is selected between sum of exponents E_x and E_y, and E_w in this unit. Then, the aligned additive and vectors carry and sum are added in the unit *CSA*. The resultant sum is obtained after the *CPA* unit, during this operation also sticky bit and leading zeros are generated by the units *LZA*(Leading Zero Anticipator) and *Sticky* respectively. The resultant sum is normalized with the value taken from the *LZA* unit, then rounded in *Round* unit to its final value. The exponent is also adjusted with the value from *LZA* and *Round* unit to final value. The sign bit is determined in *Sign* unit from the sum generated by the *CPA* unit

2.4.4 Floating-Point Division

Though, floating point division is not popular as much as floating point multiplication or floating point addition, this operation is also supported in hardware in modern processors. The operation is expressed with

$$Q = X/D \tag{2.19}$$

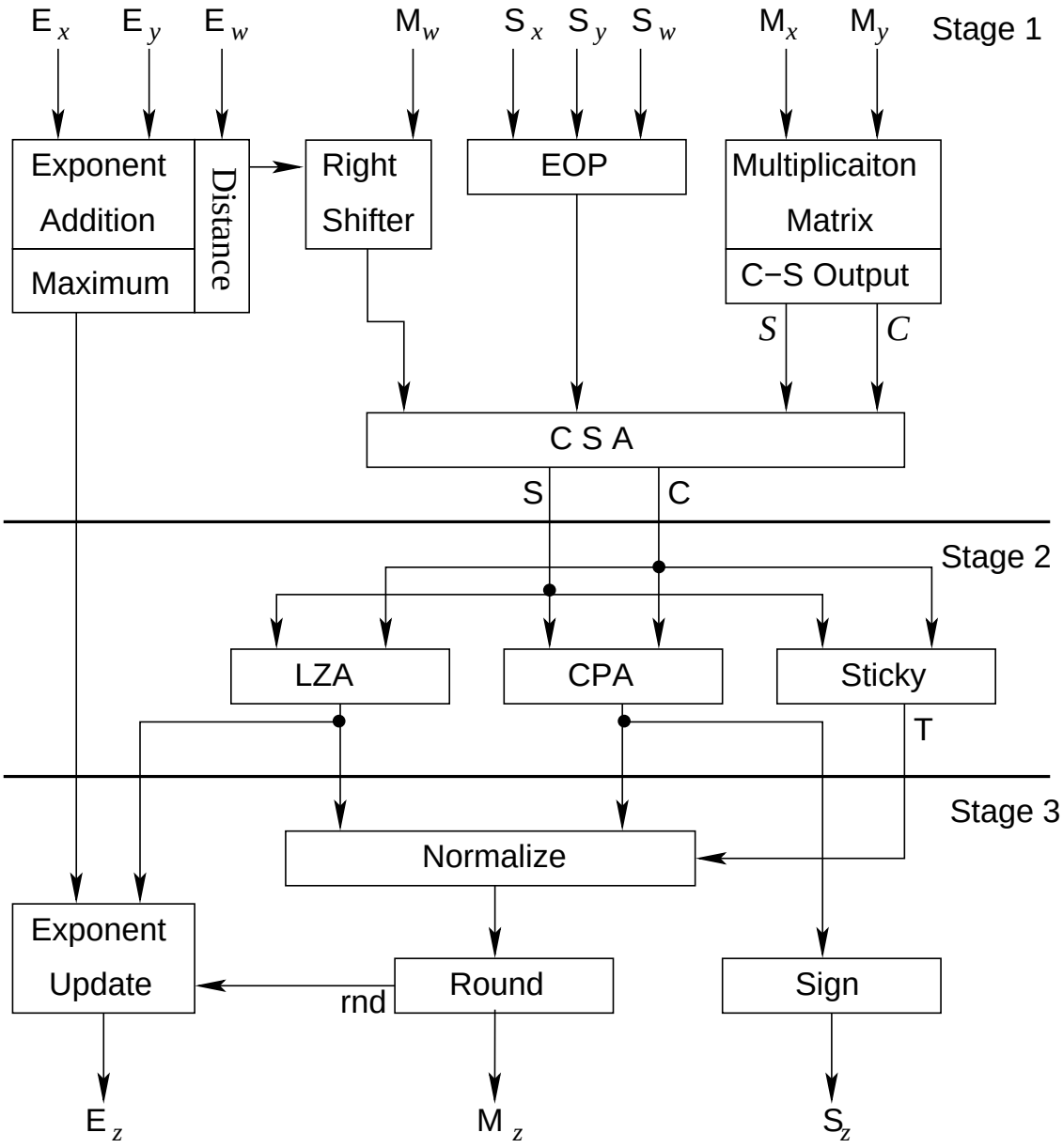


Figure 2.7 Floating-Point Multiply Add Fused.

where the operands $X = S_x \cdot 2^{E_x} \cdot M_x$ is dividend, $D = S_d \cdot 2^{E_d} \cdot M_d$ is divider and $Q = S_q \cdot 2^{E_q} \cdot M_q$ is quotient. All the mantissas are signed and normalized. The division of mantissas and exponent subtraction is performed with the Equations

$$M_q = 1.M_x / 1.M_d \tag{2.20}$$

$$E_q = E_x - E_d \tag{2.21}$$

The division of the mantissas is realized with either Radix-2 or 4 Digit Recurrence method or reciprocal of the divisor *d* is multiplied by the dividend *x*. In the Digit Recurrence method increase of radix makes quotient-digit selection more complicated. Beside, it reduces the number of iteration need for exact-quotient. For simplicity Radix-2 division algorithm is demonstrated below.(Ercegovac and Lang, 2004)

1. *Initialize*

   $WS[0] \leftarrow x/2$; $WC[0] \leftarrow 0$; $Q[-1] = 0$; $q_0 = 0$;

2. *Recurrence*

   for $j = 0 \cdots n+1$; ($n+2$ iterations because of initialization and guard bit)

   $q_{j+1} \leftarrow SEL(\widehat{y})$;

   $(WC[j+1], WS[j+1]) \leftarrow CSA\left(2WC[j], 2WS[j], -q_{j+1}d\right)$;

   $Q[j] \leftarrow CONVERT(Q[j-1], q_i)$;

   end for;

3. *Terminate*

   If $w[n+2] < 0$ then $q = 2(CONVERT(Q[n+1], q_{n+2} - 1))$

   else

   $q = 2(CONVERT(Q[n+1], q_{n+2}))$;

where (WS) and (WC) represent sum and carry vectors in the residual redundant form, i.e. $w[j] = (WC[j], WS[j])$ where w is residual of partial remainder, n is the precision in bits, $q_j \in \{-1, 0, 1\}$ is the j^{th} quotient digit, SEL is the quotient-digit selection function given in Equation 2.22 with $\widehat{y}$ the value of truncated carry-save shifted residual ($2w[j]$) with four bits. (three integer and one fractional bit). Because the range of $\widehat{y}$, $2w[j]$ requires also three integer bits and, therefore,$w[j]$ has two integer bits. *CSA* is carry- save adder,

$-q_{j+1}d$ is in two's complement form, $CONVERT$ is the on-the-fly conversion function producing the accumulated quotient in conventional representation.

$$q_{j+1} = SEL\left(\widehat{Y}\right) = \begin{cases} 1 & \text{if } 0 \leq \widehat{y} \leq 3/2 \\ 0 & \text{if } \widehat{y} = 1/2 \\ -1 & \text{if } -5/2 \leq \widehat{y} \leq -1 \end{cases} \tag{2.22}$$

In the later method, Newton-Raphson iteration is used for computation of divisor reciprocal. The main idea of this method is to find a zero point of function. The derivation can be carried out by Taylor series. It is shown in Figure 2.8. The Newton-Raphson

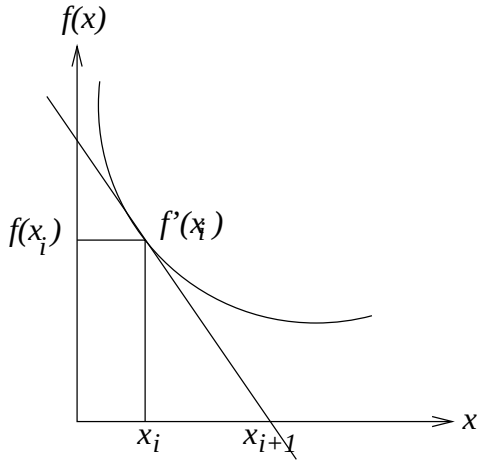


Figure 2.8 Newton-Raphson Iteration.

formula is

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) \tag{2.23}$$

if $f(x_{i+1})$ is approximate to 0, then

$$x_{i+1} = x_i - f'(x_i)/f'(x_i) \tag{2.24}$$

where x_i is the value of ith iteration, $f(x_i)$ is the value of function at x_i and $f'(x_i)$ is the derivative of function at x_i.

A lookup is used to approximate the initial value of the iteration and fast multipliers are used for getting closer to the result (Chen, Wang, Zhang, Hou, 2006). The division operation is formulated with this method as

$$q = x/d = x \times (1/d) \tag{2.25}$$

The reciprocal value of $1/d$ is formulated in Newton-Raphson method as

$$f(q) = 1/q - d \tag{2.26}$$

$$q_{i+1} = q_i \times (2 - q_i \times d) \tag{2.27}$$

$$q_0 = 1/d_0 \tag{2.28}$$

The subtraction of the exponents in biased representation is performed by subtracting the exponents and adding the missed bias. The operation is expressed as

$$E_{B,q} = E_{B,x} - E_{B,d} + B \tag{2.29}$$

where B is bias value.

The second step is normalization of M_q and update of exponents. After division the quotient is in a range of $\left(\frac{1}{2}, 2\right)$, for the $IEEE754$ standard the range is $[1,2)$, a normalization might be required when the result is less than 1. That is a left shift and decrement of exponent. In the third step rounding of Quotient is done. For digit recurrence method the rounding take place with on-the-fly-conversion (Ercegovac and Lang, 1987). The last step is determination of special values. The same situation of Floating-Point multiplication with out any change is applicable to floating point devision. The floating point divider can be seen in Figure 2.9.

2.5 Floating-Point Packed Data

Floating-point operation that applied to the multimedia data are in a form of SIMD type. This type of instructions uses multiple data in packed form. For example, two single precision floating numbers can be packed as shown in Figure 2.10. In this figure $R1$ holds A and C, $R2$ holds B and D, and $R3$ holds E and F.

The multimedia applications perform the same operation on multiple data for example, while processing a 3D scene of a movie, the same lighting transformation is applied to the every pixel of the image or while processing a voice, the same filtering is a applied to the every sample of voice. Generally multimedia data are packed in low precision format that means two or more of them can be stored in one higher precision data. Using this advantage number of loops used for processing multimedia data might be reduced by

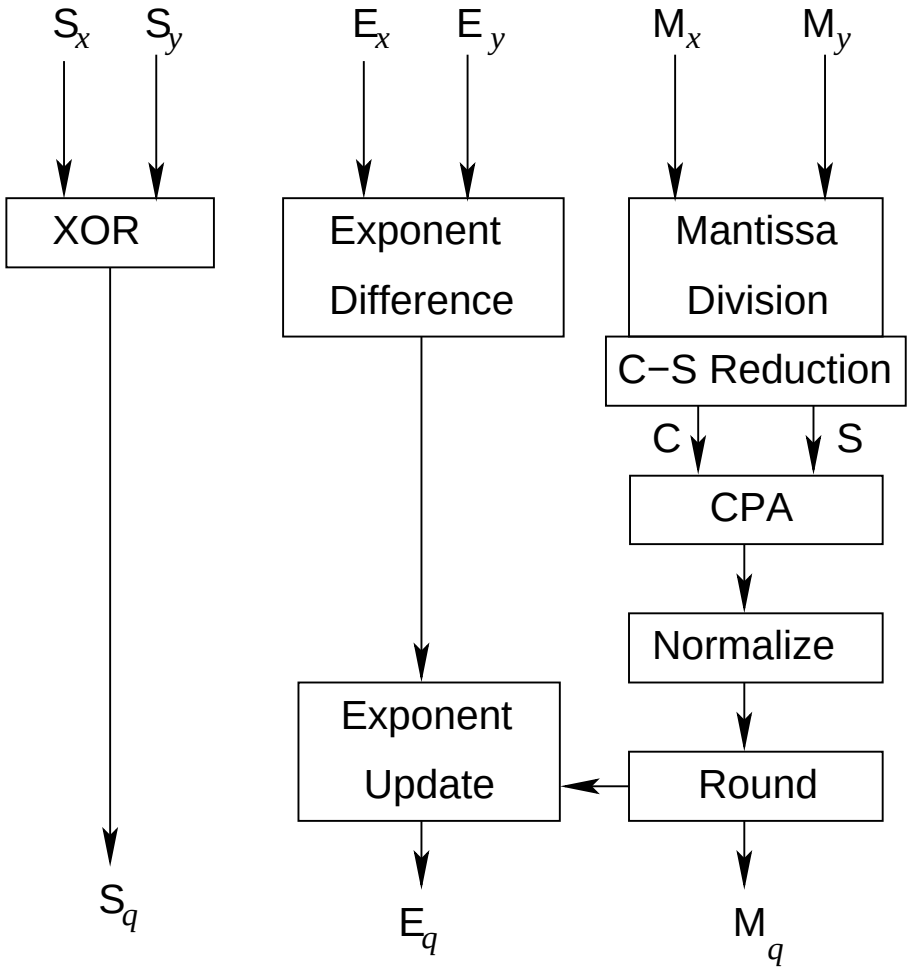
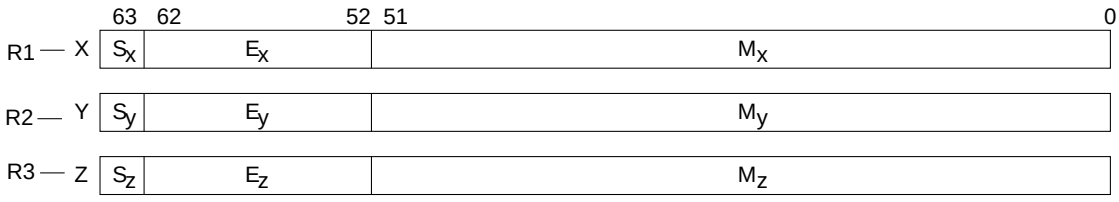
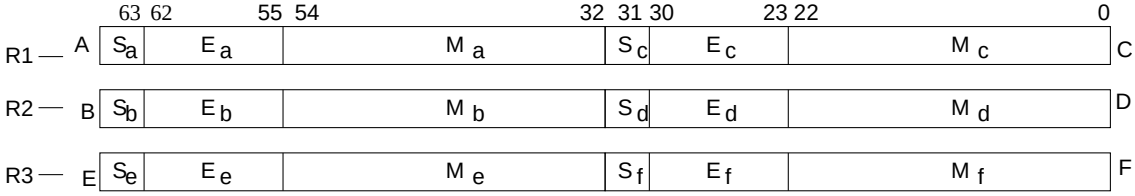


Figure 2.9 Floating-Point Divider.



(a) Double Precision Floating−Point Number



(b) Single Precision Floating−Point Number

Figure 2.10 SIMD Type Data Alignment.

using vector structures. Using these vectors multiple addition, subtraction, multiplication or division can be performed in once.

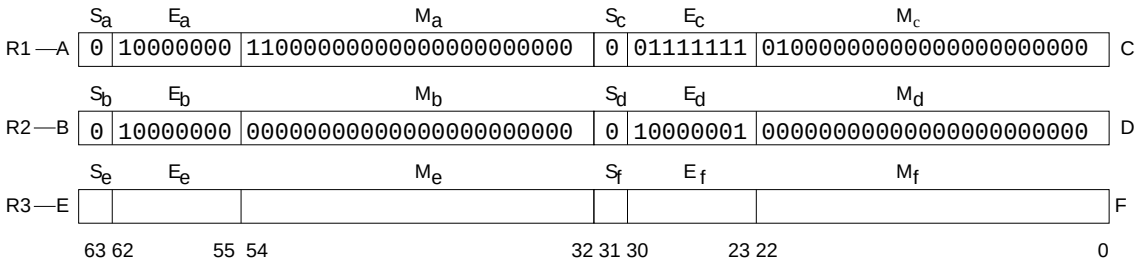


Figure 2.11 SIMD Type Data Alignment Example.

2.5.1 Packed Floating Point Addition and Subtraction

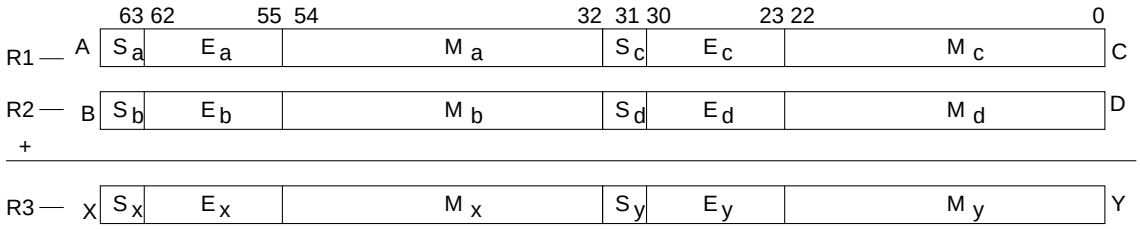


Figure 2.12 SIMD Addition Alignment Example.

Figure 2.12 demonstrates the operation of packed floating point addition operation on single precision operands. The additions of A to B and C to D onto E and F respectively. Formulated as

$$S_x = \begin{cases} S_a & \text{if } E_a > E_c \\ S_c & \text{if } E_a < E_c \end{cases} \qquad S_y = \begin{cases} S_b & \text{if } E_b > E_d \\ S_d & \text{if } E_b < E_d \end{cases} \tag{2.30}$$

$$E_x = max(E_a, E_c), \qquad E_y = max(E_b, E_d) \tag{2.31}$$

$$M_x = 1.M_a + 1.M_c \qquad M_x = 1.M_b + 1.M_d \tag{2.32}$$

Each addition member of the packed is added using standard floating point addition algorithm as shown in Equations (2.9) and (2.10). The mantissas of each addition is aligned in pairs simultaneously. Then the effective operation is performed on aligned mantissas in once. The exponents are also handled in pairs. The greater exponent is selected from

each pair. Both additions are normalized, rounded and each exponent is updated simultaneously. Then each additions are packed with order sign,exponent and mantissa of first addition than second addition like in the Figure 2.10 (Gök and Özbilen, 2008). The computed results and their layout in the resultant register can be seen in Figure 2.13, where the value in part E is 5.5 and in part F is 4.25

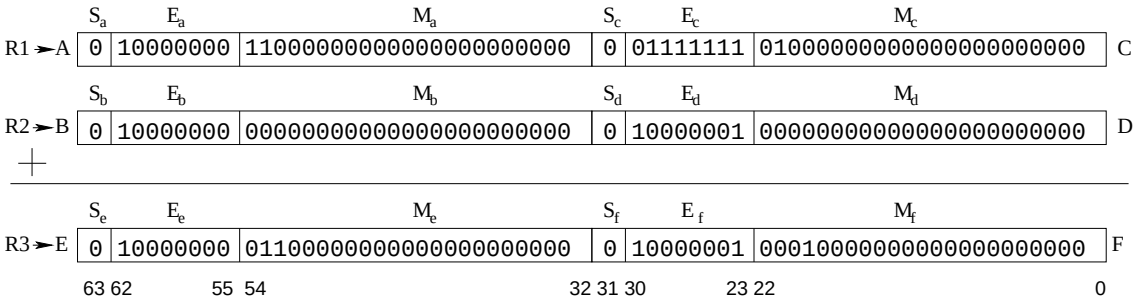


Figure 2.13 SIMD Addition Numerical Example.

2.5.2 Packed Floating Point Multiplication

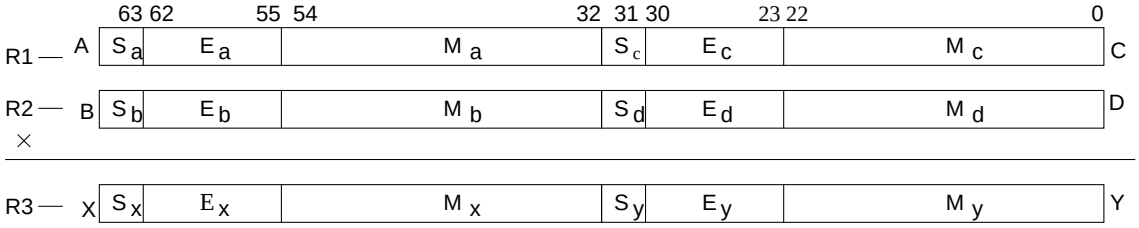


Figure 2.14 SIMD Multiplication Alignment Example.

Figure 2.14 demonstrates the operation of packed floating point multiplication on data packets that contain two single-precision floating point numbers. Each corresponding member of the packets are multiplied independently as

$$S_x = S_a \oplus S_c, \qquad S_y = S_b \oplus S_d \tag{2.33}$$

$$E_x = E_a + E_C - B, \quad E_y = E_b + E_d - B \tag{2.34}$$

$$M_x = 1.M_a \times 1.M_c \quad M_y = 1.M_b \times 1.M_d \tag{2.35}$$

Packed multiplication uses double precision multiplication matrix for multiplication of both mantissas. The reduction of multiplication matrix is done by double precision

matrix. The sum of exponents also handled in the extended exponent adder of double precision multiplier in the same way subword integer addition. Signs are simultaneously. The path-a-way of packed multiplication is same in original floating point multiplication. That is normalization and rounding is done simultaneously. Then the results are packed into one double-precision area as in the floating point addition. The results of the multiplication and their alignment in 64 bit can be seen in Figure 2.15, where the value of part *E* is 7.0 and the value of part *F* is 5.0.

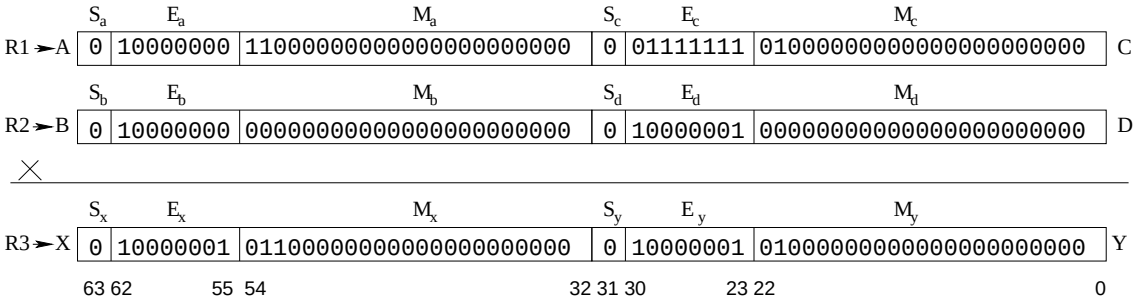


Figure 2.15 SIMD Multiplication Numerical Example.

2.5.3 Packed Floating Point Division and Reciprocal

In modern processors the packed division operation is performed using multiplicative division method. In this method, the reciprocal of packed divisors is multiplied with the packed dividend using packed multiplication operation. In packed reciprocal, the reciprocal of floating point number in location *B* is computed using Newton Rapson method that explained before then result is duplicated to *D* location.

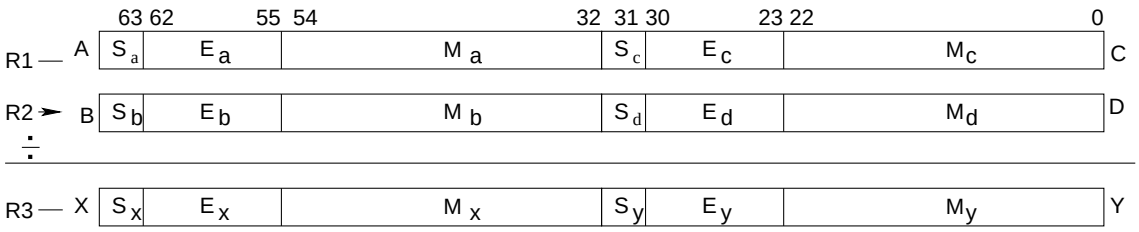


Figure 2.16 SIMD Division Alignment Example.

Figure 2.16 demonstrates the operation of packed floating point division on packets that contains two single-precision floating point numbers. Each corresponding member

of packets are multiplied with reciprocal of divisor independently as

$$S_x = S_a \oplus S_c, \qquad S_y = S_b \oplus S_d \tag{2.36}$$

$$E_x = E_a - E_c + B, \qquad E_y = E_b - E_d + B \tag{2.37}$$

$$M_x = 1.M_a \times (1/1.M_c) \quad M_x = 1.M_b \times (1/1.M_D) \tag{2.38}$$

For example, in Figure 2.18, if the floating numbers in locations A and C on $R1$ are divided to 2.0. The floating point number 2.0 is put in B on $R2$ then packed reciprocal operation is executed in $R2$ register. The results of reciprocal operation can be seen in Figure 2.17.

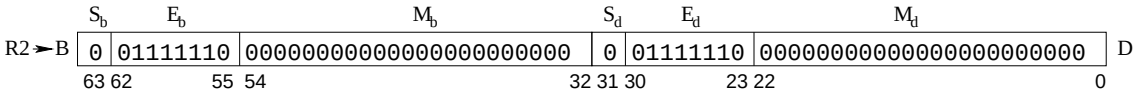


Figure 2.17 SIMD Reciprocal Numerical Example.

Then, the packed multiplication operation is executed between $R1$ and $R2$ for completing division operation. The results of divisions are in locations X and Y on R3, which have values respectively 1.75 and 0.675 can be seen in Figure 2.18.

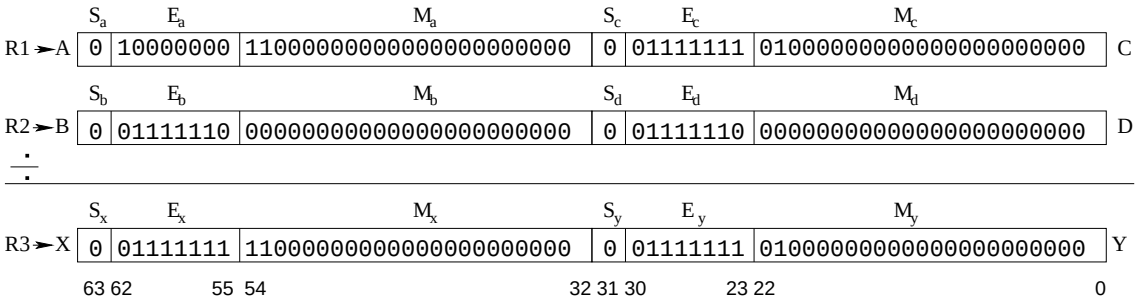


Figure 2.18 SIMD Division Numerical Example.

2.5.4 Packed Floating Point Multiply Add Fused(MAF)

As mentioned before, multiplication and addition operations can be joined and replaced by a MAF circuit. A double precision FPMAF can be modified to work on two packed single precision number. The packed form of FPMAF uses the main functions of

the standard FPMAF. The exponent units are slightly modified to handle both multiplications' exponent addition and update operations. The rounding and normalization units are modified for both single/double precision and multiple data operations. The Multiplication matrix used to multiply two multiplication of packed data. The packed form of MAF can have an additional function *dot product*. With *dot product* operation, two pairs of single precision multiplication can be executed and summed with a third single precision number, which might be previously computed multiplication. Multiplication matrix and adders must be modified to handle this operation. A summery of a packed MAF can perform is listed in Table 2.3 using the inputs in Figure 2.11. As in packed multiplication, all other parts of standard MAF is shared. As an example, the single precision dot product operation and its result is demonstrated in Figure 2.19. Here, the content of single precision floating point numbers in location A and C, and B and D are multiplied in pairs and added to floating point number location in F with value 3.75. The result of dot product operation is location F with value 15.75.

Table 2.3 Operations of Packed MAF

Operation	Description
$A*B+C*D+F$	Dot product
$A*B+C*D$	Sum of product by setting $F=0.0$
$A+C+F$	Triple adder by setting D and B to 1.0
$A*B\|\|C*D$	Dual multiplication by setting $F=0.0$
$A*B+F$	Single MAF by setting D or B to 0.0
$A*B$	Single multiplication by setting D or B and F to 0.0
$A+F$	Single addition by setting D or B to 0.0 and C to 1.0

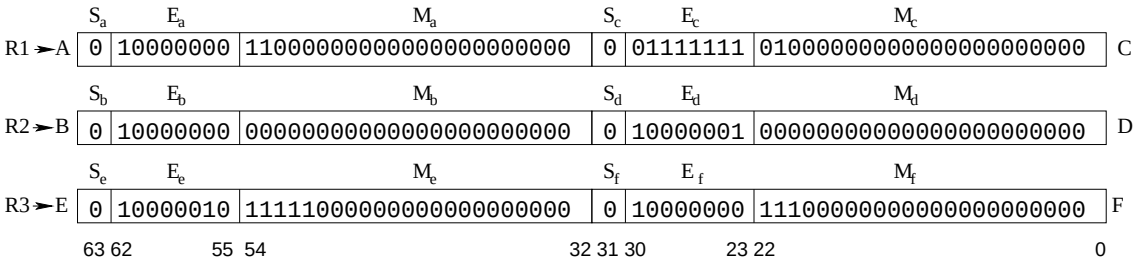


Figure 2.19 Packed Single Precision Floating Point Dot Product Results.

2.6 Floating Point Packed Instruction Extensions

Today, many general purpose processors have multimedia extensions which includes SIMD type instructions. AMD has 3DNow! extension. 3DNow! technology is a set of new instructions providing single-precision floating-point packed data to x86 programs. 3DNow! architecture is an innovative extension of the x86 MMX architecture. It uses the same registers and the same basic instruction formats supporting register-to-register and memory-to-register instructions. 3DNow! technology introduces single-precision floating point format to existing MMX register set, which is compatible with IEEE-754 single-precision format shown in Figure 2.20. 3DNow! instructions support two-packed single-precision floating point operations addition, subtraction, multiplication and reciprocal.

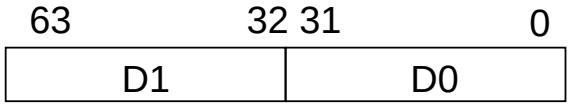


Figure 2.20 3DNow! technology floating-point data type: Two packed IEEE single-precision floating-point doublewords (32 bits $\times$2)(AMD, 2000).

The Intel Corporation introduce SSE extensions with Pentium III processor family. The SSE instructions operate on packed single-precision floating-point values contained in the XMM registers and on packed integers contained in the MMX registers. The SSE SIMD integer instructions are an extension of the MMX technology instruction set. Several additional SSE instructions provide state management, cache control, and memory ordering operations. The SSE instructions are targeted at applications that architecture operate on arrays of single-precision floating-point data elements, including 3-D geometry, 3-D rendering, and video encoding and decoding applications.The packed floating point operations that SSE support are addition, subtraction, multiplication, division and reciprocal with two packed operand. The SSE2 extensions were introduced in the Pentium 4 processors. The SSE2 instructions operate on packed double-precision floating-point values contained in the XMM registers and on packed integers contained in the MMX and the XMM registers. Figure 2.21 shows a summary of the various SIMD extensions, the data types they operated on, and how the data types are packed into MMX and XMM registers(Intel, 2007). With the core architecture, Intel introduces the SSE4 and SSE4.1,

the SSE4.1 has also give support to packed floating point dot product in both double and single precision data type.



Figure 2.21 SIMD Extensions, Register Layouts, and Data Types.(Intel, 2007)

The PowerPC processor from Motorola instruction set is extended by Altivec technology. Altivec is based on SIMD style parallel execution units that operate on 128-bit vectors. The Altivec technology supports 16-way parallelism for 8-bit signed and unsigned integers, 8-way parallelism for 16-bit signed and unsigned integers and 4 way parallelism for 32-bit signed and unsigned integer and IEEE-754 floating point numbers. The Altivec data element can be seen in Figure 2.22. The Altivec ISA (instruction set architecture) includes floating-point arithmetic, rounding and conversion, compare and estimate operation. In this set, it supports packed single precision floating point operations addition, subtraction, multiply-add, multiply-subtract and reciprocal on 4 way packed single-precision floating point numbers. The target application for the AltiVec technology are IP(Internet Protocol) telephony gateways, multi-channel modems, speech processing systems, echo cancelers, image and video processing systems, scientific array processing systems, as well as network infrastructure such as Internet routers and virtual private network servers. (Freescale, 2006)

Quad Word															
Word 0				Word 1				Word 2				Word 3			
Haft−Word 0		Haft−Word 1		Haft−Word 2		Haft−Word 3		Haft−Word 4		Haft−Word 5		Haft−Word 6		Haft−Word 7	
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12	Byte 13	Byte 14	Byte 15

Figure 2.22 Motorola Altivec Vector Register (Motorola, 2000).

2.7 Benchmarking SIMD

A benchmark is a test designed to measure the performance of one particular part of a computer. For example, one benchmark might test your CPU (Central Processing Unit) is at floating point calculations by performing billions of arithmetic operations and timing how long it takes to complete them all.

There are very few benchmarking software especially focused on SIMD architecture some of them are: DARPA, ALPBench, Multibench 1 and 2. DARPA(Defense Advanced Research Projects Agency) is an image understanding benchmark and widely-accepted platform for evaluation of parallel systems (Weems, Riseman, Hanson and Rosenfeld, 1991). MediaBench is a benchmark suite, that introduced in 1997, provides set of full application-level benchmarks for studying video processing characteristics (Lee, Potkonjak and Mangione-Smith, 1997). ALPBench(All Levels of Parallelism for Multimedia) is a suit that includes five complex media applications from various sources: speech recognition, face recognition complex media applications, ray tracing, MPEG-2(Moving Pictures Experts Group) encode/decode.

Below there are some benchmarking suit results taken from the Mediabench suit tools:

JPEG(Joint Photographic Experts Group): This package contains C software to implement JPEG image compression and decompression. Shade analyzer output:

```
#instruction count: 13905129
#alu op's:   8171845
%alu op's:      0.59
#immed op's:   5219031
%immed op's:      0.64

Stores
======
  Total      st08      st16      st32      stxx
========= ========= ========= ========= =========
   709615    139912     54861    514841         1
               0.20      0.08      0.73      0.00
```

```
Alu op's
========
  Total      op08      op16      op32      opxx
========= ========= ========= ========= =========
  2208348    490216    255747   1462385         1
               0.22      0.12      0.66      0.00

#op's used for output:   2208348
%op's used for output:      0.27

Analyzer: /u/gs3/leec/leec/Projects/MediaBench/SPIX/SHADE/src/alu
Version: 1.0 (10/Mar/97)
(shade version: 5.25 V8 SPARC ELF32 (14/Feb/95))
Uname: panther sun4u SunOS 5.5.1 Generic_103640-08
Start: Mon Jun 16 19:31:32 1997
Application:
./cjpeg -dct int -progressive -opt -outfile testout.jpg testimg.ppm
Application Instructions: 13905129
Stop: Mon Jun 16 19:32:07 1997
Instructions: 13905129
Time: 14.580 usr  0.010 sys  35.169 real  41.485%
Speed: 953.059 KIPS
```

MPEG: mpeg2play is a player for MPEG-1 and MPEG-2 video bitstreams. It is based on mpeg2decode by the MPEG Software Simulation Group. Shade analyzer output:

```
#instruction count: 175505114
#alu op's:  78655559
%alu op's:      0.45
#immed op's:  59915131
%immed op's:      0.76

Stores
```

```
======
  Total      st08      st16      st32      stxx
========= ========= ========= ========= =========
 11126484   1544167   1057402   7003691   1521224
               0.14      0.10      0.63      0.14
Alu op's
========
  Total      op08      op16      op32      opxx
========= ========= ========= ========= =========
 16247622   1998403    362264  13886546   1521224
               0.12      0.02      0.85      0.00

#op's used for output:  16247622
%op's used for output:      0.21

Analyzer: /u/gs3/leec/leec/Projects/MediaBench/SPIX/SHADE/src/alu
Version: 1.0 (10/Mar/97)
(shade version: 5.25 V8 SPARC ELF32 (14/Feb/95))
Uname: cheetah sun4u SunOS 5.5.1 Generic_103640-08
Start: Tue Jun 17 02:21:22 1997
Application:
../src/mpeg2dec/mpeg2decode -b mei16v2.m2v -r -f -o0 tmp%d
Application Instructions: 175505114
Stop: Tue Jun 17 02:24:15 1997
Instructions: 175505114
Time: 122.930 usr  0.120 sys  173.355 real  70.982%
Speed: 1426.291 KIPS
```

Testing the performance effects of SIMD instructions on practical, needs special benchmarking suits. To learn the how efficient SIMD instructions work, a program which is suitable for SIMD operations must be written. An ideal program to show SIMD performance must be repetitive in its method. An image or video processing application is a

good candidate, which benchmark suits simulates. An investigation of SIMD instruction set form University of Ballarat, uses a program to compute the approximate value of pi. They use series give in Equation 2.39 for calculating pi.

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}\cdots\approx\frac{\pi}{4} \tag{2.39}$$

This is an inefficient algorithm however, large number of iterations make it ideal candidate. To show the effectiveness of SIMD, the main loop of the program makes 128,000 iteration 1000 times, which gives an accurate pi value with single precision floating numbers. The algorithm is executed five times on :

1. A version that uses the CPU alone in a SISD manner.
2. A version optimized for Altivec on the PowerPC chip.
3. A version optimized for SSE2 on Intel (x86) chip.

In this study, 8 different configurations were used:Pentium 4 with SSE3 at 2.80Ghz on Ubuntu Linux, Pentium 4 with SSE3 at 2.80Ghz on OSX(Dev), Pentium 4 with SSE3 at 1.40Ghz on Ubuntu Linux, Pentium 4 with SSE3 at 1.40Ghz on OSX(Dev), Pentium 4 with SSE2 at 2.00Ghz on Ubuntu Linux, Quad Xeon with SSE3 at 3.10Ghz on Gentoo Linux, Dual PowerPC G5 with Altivec at 2.7Ghz on OSX Version 10.4.3 and PowerPC G5 with Altivec at 1.4Ghz on OSX Version 10.4.

The Figure 2.23 shows the score have while CPU's working with bare instructions, though in Figure 2.23 the CPU's working with SIMD type instructions. These figures shows SIMD type instruction has great impact on performance if they are usable. It is also seen that clock speed hight effective on overall performance.

2.8 Previous Packed Floating Point Designs

2.8.1 Packed Floating Point Multiplication Designs

A recent work in (Akkas and Schulte, 2006) presents a quadruple precision floating point multiplier that supports two dual-precision floating-point multiplications in parallel. The design is shown in Figure 2.24.

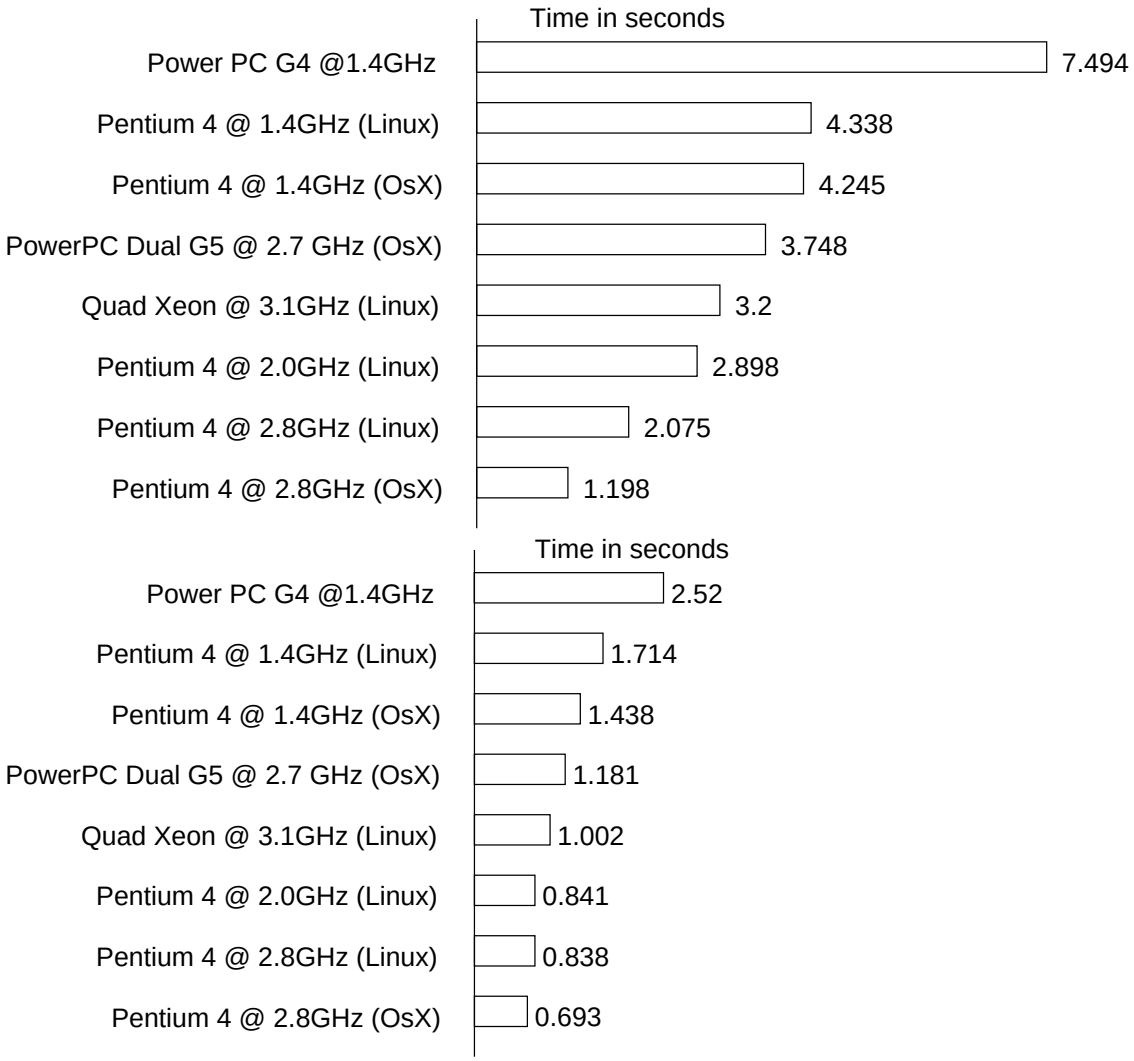


Figure 2.23 benchmark Result of with out SIMD and with SIMD.

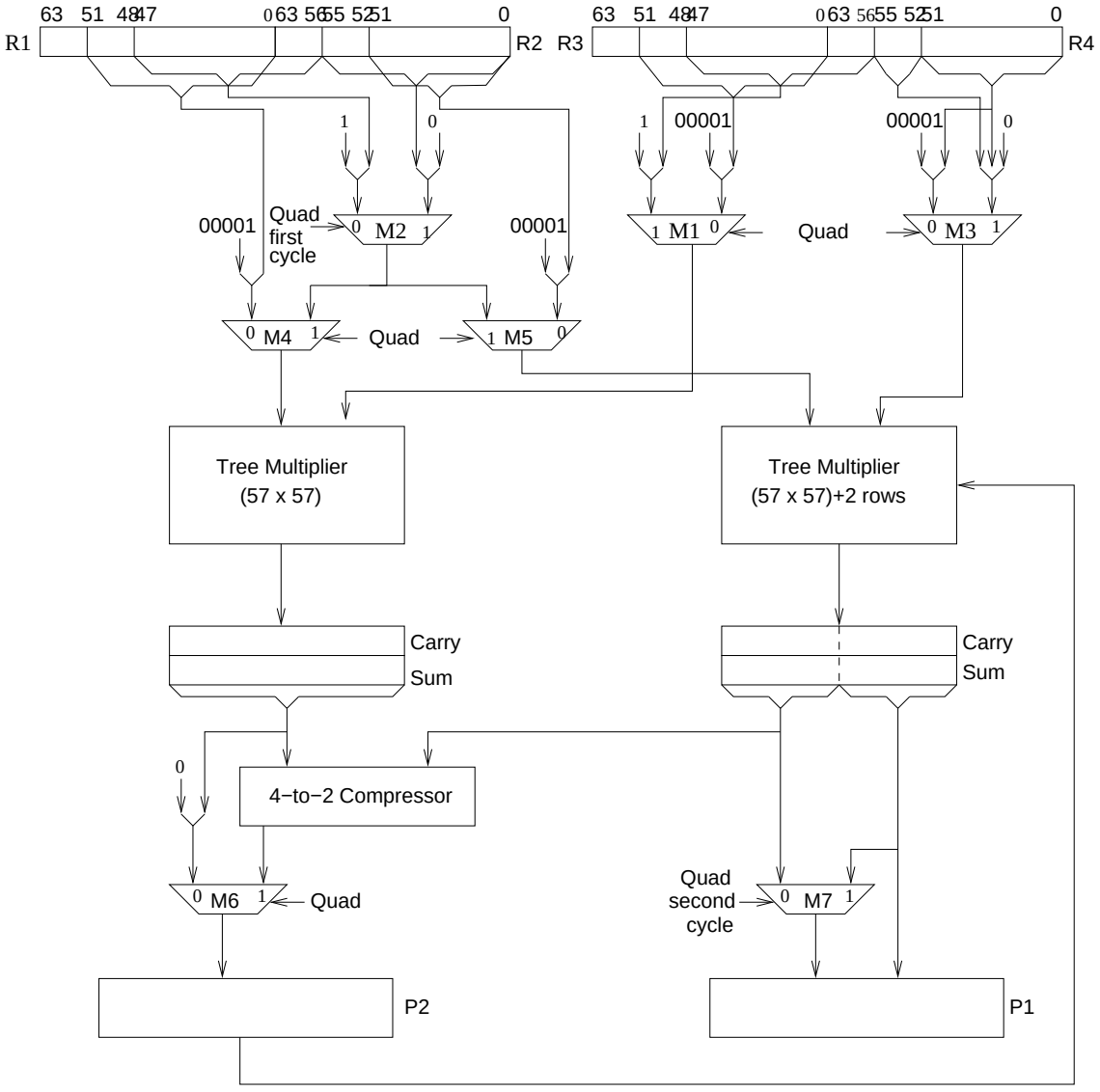


Figure 2.24 Dual Mode Quadruple Precision Multiplier (Akkas and Schulte, 2006).

The same technique is also used for dual-mode double precision floating point multiplier that performs two single precision multiplications in parallel. The divide-and-conquer technique (Beuchat, Tisserand, 2002) is used to multiply mantissas of high precision floating point numbers. This technique uses smaller multiplications and additions to compute high precision multiplication. If two n bits numbers, X and Y can be divided into two parts, such as

$$X = X1 \cdot k + X0 \tag{2.40}$$

$$Y = Y1 \cdot k + Y0 \tag{2.41}$$

where $k = 2^{n/2}$. The product of $X \cdot Y$ is computed as

$$(X1 \cdot k + X0) \cdot (Y1 \cdot k + Y0) = X1 \cdot Y1 \cdot k^2 + (X1 \cdot Y0 + X0 \cdot Y1) \cdot k + X0 \cdot Y0 \tag{2.42}$$

Figure 2.25 shows technique given with Equation 2.42.

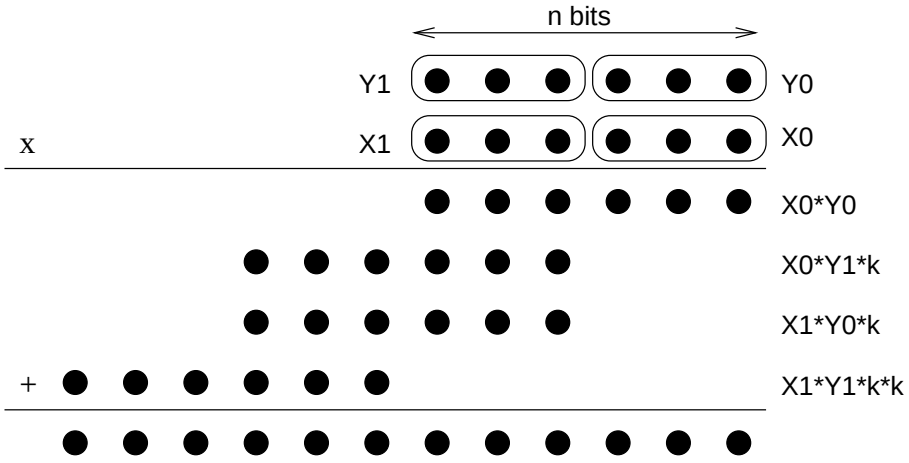


Figure 2.25 The Divide-and-Conquer Technique(Akkas and Schulte, 2006).

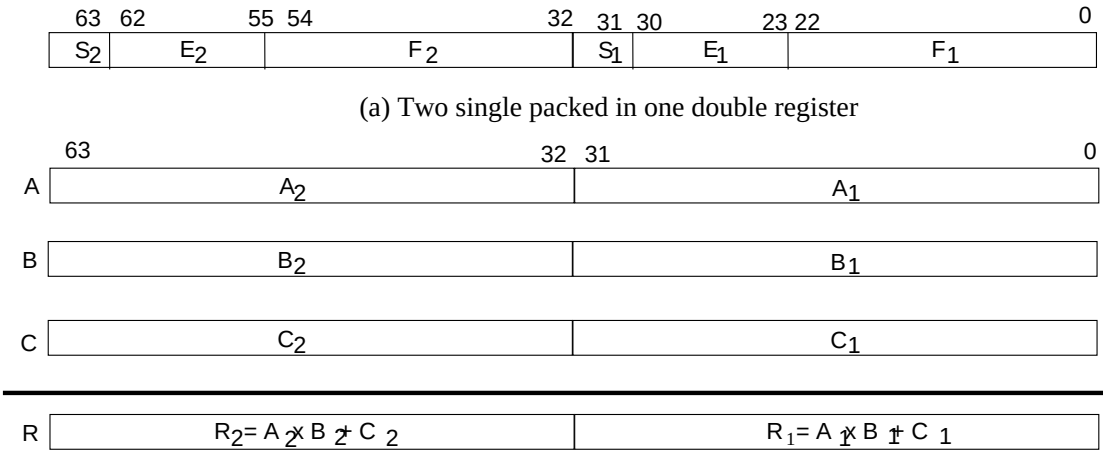
2.8.2 Packed Floating Point Multiplier Add Fused Designs

One of the few multi functional MAF design is presented in (Heikes and Colon-Boneti, 1996). That study describes two floating-point multiply-add units capable of performing IEEE-754 compliant single and double precision floating-point operations. Of course, it is possible to use a larger precision floating-point unit to operate on smaller precision operands, however, this requires the conversion of smaller precision operands

to the larger precision format and then conversion of the result back to smaller precision format. The conversion operations might significantly reduce the performance.

Another MAF design is presented in (Huang, Shen, Dai, and Wang, 2007). That study proposes a new architecture for the MAF unit that supports multiple IEEE precisions multiply-add operation with Single Instruction Multiple Data (SIMD) feature. The proposed MAF unit can perform either one double- precision or two parallel single-precision operations using about 18% more hardware and with 9% increase in delay than a conventional double-precision MAF unit. The simultaneous computation of two single-precision MAF operations is adapted by redesigning several basic modules of double-precision MAF unit. The adaptation are either segmentation by precision mode dependent multiplexers or duplication of hardware. The proposed MAF unit can be fully pipelined and the experimental results show that it is suitable for processors with floating point unit (FPU).

Figure 2.26.a shows the 64-bit double-precision register used to store two single-precision number and Figure 2.26.b shows the generated results when performing two single-precision MAF operations.



(a) Two single packed in one double register

(b) Two single MAF operation result

Figure 2.26 Two Single-Precision Numbers Packed in One Double-Precision Register (Huang, Shen, Dai, and Wang, 2007).

The MAF unit is considered as an exponent and a mantissa unit. From the Table 2.4, it is seen that for exponent processing, the word-length of 13-bit double-precision exponent should be extended to 20-bits for two single-precision computing. But for speed, two

separated single precision exponent is used in this design. Below, the algorithm shows

Table 2.4 Word-lengths in Single/Double Precision MAF

modules	single	double
Multiply Array	24	53
3-2 CSA	48	106
Alignment-Adder-Normalization	74	161
Exponent Processing	10	13

mantissa datapath of the simplified multiple-precision MAF unit. In the algorithm, s_a, e_a, f_a denote sign, exponent and mantissa of the operand A respectively. The same rule is applied for operands B and C. The control signal *double* is used for double- precision operation. The signal $x[m:n]$ denotes the portion of x from n to m. *s.sub*, $s.sub_1$ and $s.sub_2$ in Step 3 denotes the signs of the effective mantissa addition operations for one double and two single-precision operations respectively. The proposed MAF unit derived from algorithm is shown in Figure 2.27.

2.9 Previous Patented Packed Floating Point Designs

2.9.1 Multiple-Precision MAF Algorithm

The algorithm requires *A, B, C* to be normalized numbers (Huang, Shen, Dai, Wang, 2007).

Step 1: Exponent Difference: $\delta[19:10]$

if $double = 1$ **then**

$\delta[12:0] = e_a[12:0] + e_b[12:0] - e_c[12:0] - 967$

else

$\delta[9:0] = e_a[9:0] + e_b[9:0] - e_c[9:0] - 100$

$\delta[19:10] = e_a[19:10] + e_b[19:10] - e_c[19:10] - 100$

end if

Step 2: Mantissa Product: $fprod[105:0]$

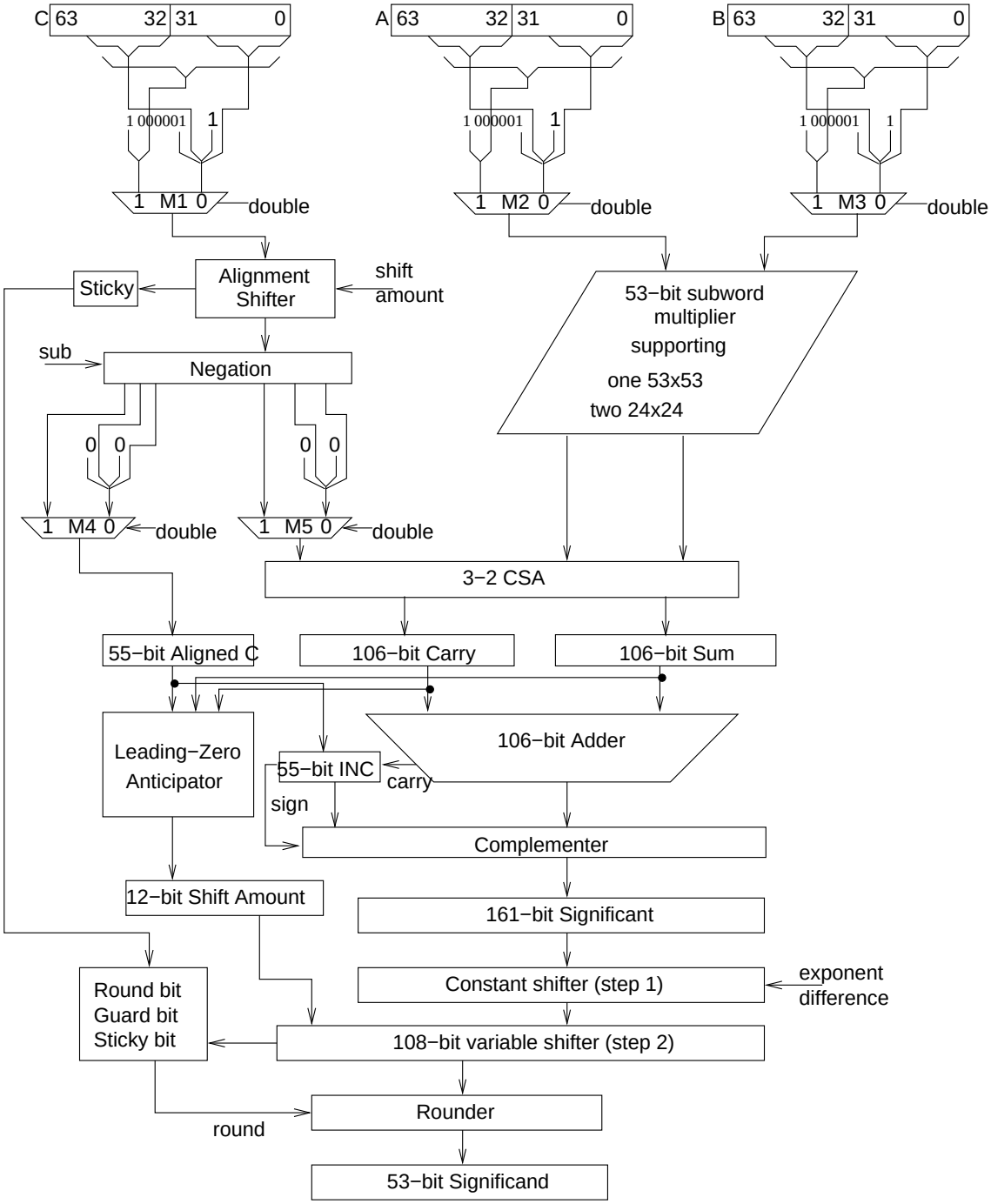


Figure 2.27 General structure of multipleprecision MAF unit (Huang, Shen, Dai, and Wang, 2007).

if $double = 1$ **then**

$fprod[105:0] = f_a[52:0] \times f_b[52:0]$

else

$fprod[47:0] = f_a[23:0] + f_b[23:0]$

$fprod[96:49] = f_a[48:25] + f_b[47:24]$

end if

Step 3: Alignment and negation: $fca[160:0]$

if $double = 1$ **then**

$fca[160:0] = (-1)^{s.sub} \times f_c[52:0] \times 2^{-\delta[12:0]}$

else

$fca[73:0] = (-1)^{s.sub_1} \times f_c[23:0] \times 2^{-\delta[9:0]}$

$fca[148:75] = (-1)^{s.sub_2} \times f_c[47:24] \times 2^{-\delta[9:0]}$

end if

Step 4: Mantissa Addition: $facc[160:0]$

$facc[160:0] = fprod[105:0] + fca[160:0]$

Step 5: Complementation: $faccabs[160:0]$

if $double = 1$ **then**

$faccabs[160:0] = |faccabs[160:0]|$

else

$faccabs[73:0] = |faccabs[73:0]|$

$faccabs[148:75] = |faccabs[148:75]|$

end if

Step 6: Normalization: $faccn[160:0]$

if $double = 1$ **then**

$faccn[160:0] = normshift(faccabs[160:0])$

else

$faccn[73:0] = normshift(faccabs[73:0])$

$faccn[148:75] = normshift(faccabs[148:75])$

end if

Step 7: Rounding: $fres[51:0]$

if $double = 1$ **then**

$fres[51:0] = round(faccn[160:0])$

else

$fres[22:0] = round(faccn[73:0])$

$fres[45:23] = round(faccn[148:75])$

end if

2.9.2 Shared Floating Point and SIMD 3D Multiplier

This is a multiplier that can perform multiplications of scalar floating point values ($X \times Y$) and packed floating points values ($X_1 \times Y_1$ and $X_2 \times Y_2$). The multiplier also can be configured to compute $X \times Y - Z$. The multiplier can be configured to compute two versions of result: With Overflow or With out Overflow exception. The main functional units of design is shown in Figure 2.28.

In Figure 2.28, the multiplexers at the input selects multiplier and multiplicand according to state machine control signal. The selected inputs are routed to booth encoder and adder. The outputs of booth encoders are routed to booth multiplexers for generating partial products. The selected partial products are reduced to carry and save vectors in the adder tree. The pre-rounded results are generated at carry-save adders by adding rounding constant and carry-save vectors. The parallel twice calculation of addition for with or with out overflow condition is for reducing processing time. The outputs of carry-save adders are passed to carry-propagate adders and sticky unit for rounding operation. The normalization units performs corrections and then the rounded result selection unit decides which result will be used.

The multiplier can operate on a maximum of 76-bit operands. It can be configured to perform all AMD 3DNOW! (AMD, 2007) SIMD floating point multiplication. The adder tree can multiply 76 by 76 bit operands or 24-32 bit packed floating point operands. It is implemented as pipelined to increase the instruction throughput.

In the first stage, adder generates the 3X multiple of multiplicand. Booth encoders generate signals to control booth multiplexers for generating signed multiples of multiplicand. In the second stage, partial products are reduced to two using adder tree. The first portion of the multiplier's rounding, which involves addition of rounding constants with

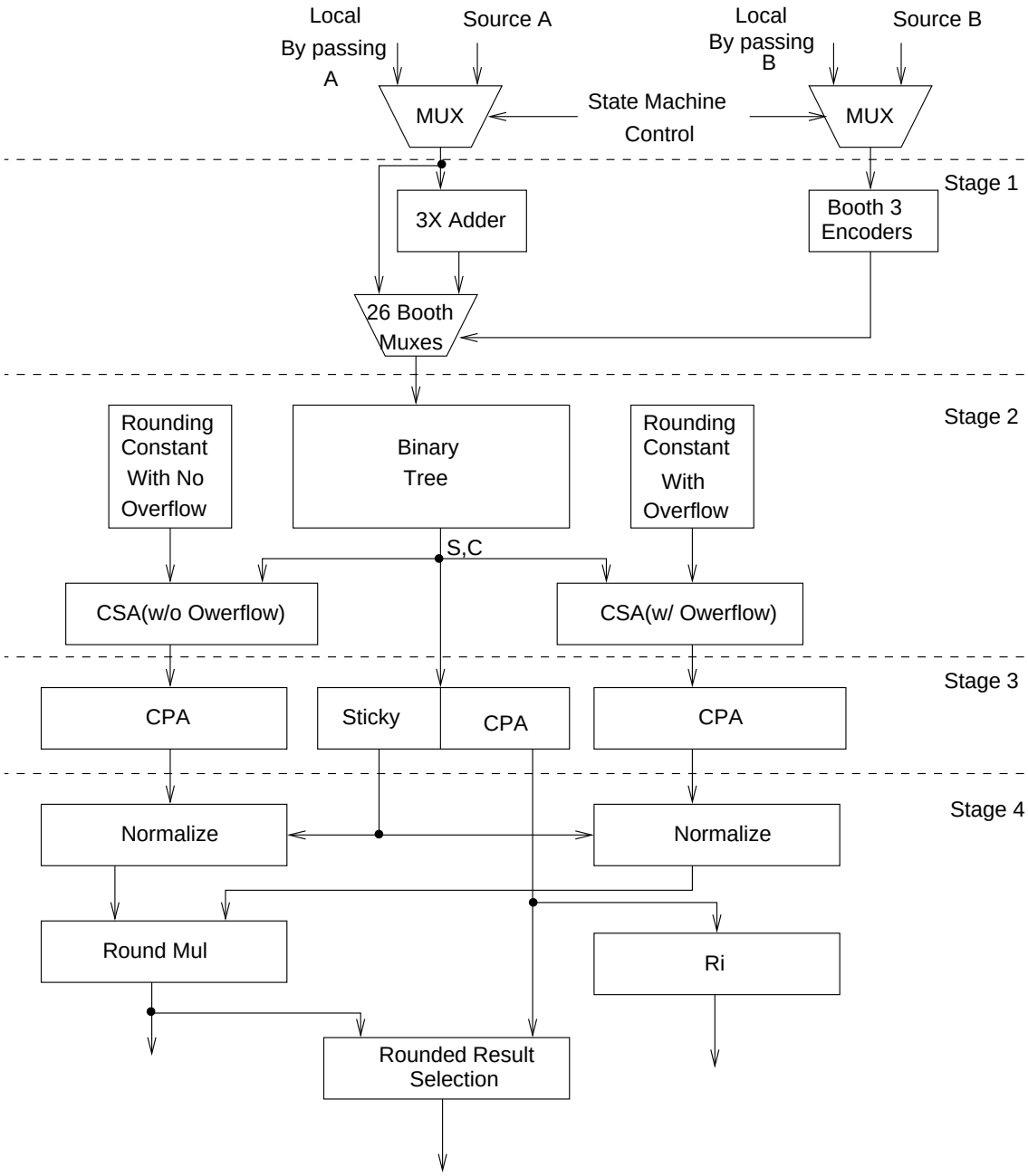


Figure 2.28 Shared Floating Point and SIMD 3D Multiplier(Oberman, 2002).

CSAs, is done in this stage. Because, the result is unknown the addition is performed twice for overflow condition. The carry-save adders are also configured to perform back-multiply and subtract operation which can be used as computation of remainder required for division and square root operation. In the third stage of pipeline, three versions of the carry-assimilated results are computed. The sticky bit is also generated in parallel from carry and save vectors. In the fourth stage, the normalization is done and rounding is completed. The most significant bit of unrounded result determined which rounded result will be used. For division and square root iterations, a result R_i is also computed. R_i is the one's complement of the unrounded multiplication result.

2.10 Method and Apparatus For Performing Multiply-Add Operation on Packed Data

This is a design from Intel Corporation, which performs primarily multiply-add operations on packed data. This design in a part of processor system. The design performs various operations on first and second packed data to generate a third packed data. The main functional blocks of design can be seen from Figure 2.29. The design can perform operations given in Table 2.5, Table 2.6 and Table 2.7. The packed data can be in three form: packed byte, packed word and packed double word. Packed byte is a storage 64-bit or 128-bit long and contains 8 or 16 elements. Packed word is a storage 64-bit or 128-bit long and contains 4 or 8 elements, which each element is 16-bit long. Packed doubleword can be 64-bit or 128-bit long and contains 4 or 8 elements. Each doubleword element is 32-bit long. The design also supports packed single and packed double formats, which can contain floating point elements. Packed single can be 64-bit or 128-bit long and contains 2 or 4 single data element. Each single data element contains 32-bit. Packed Double also can be 64-bit or 128-bit long and contains 1 or 2 double data elements. Each double data element contains 64-bit. The multiply-add and multiply-

subtract instructions can be executed on multiple data elements at the same time by a single multiplication operation on unpacked data. Parallelism may be used to process

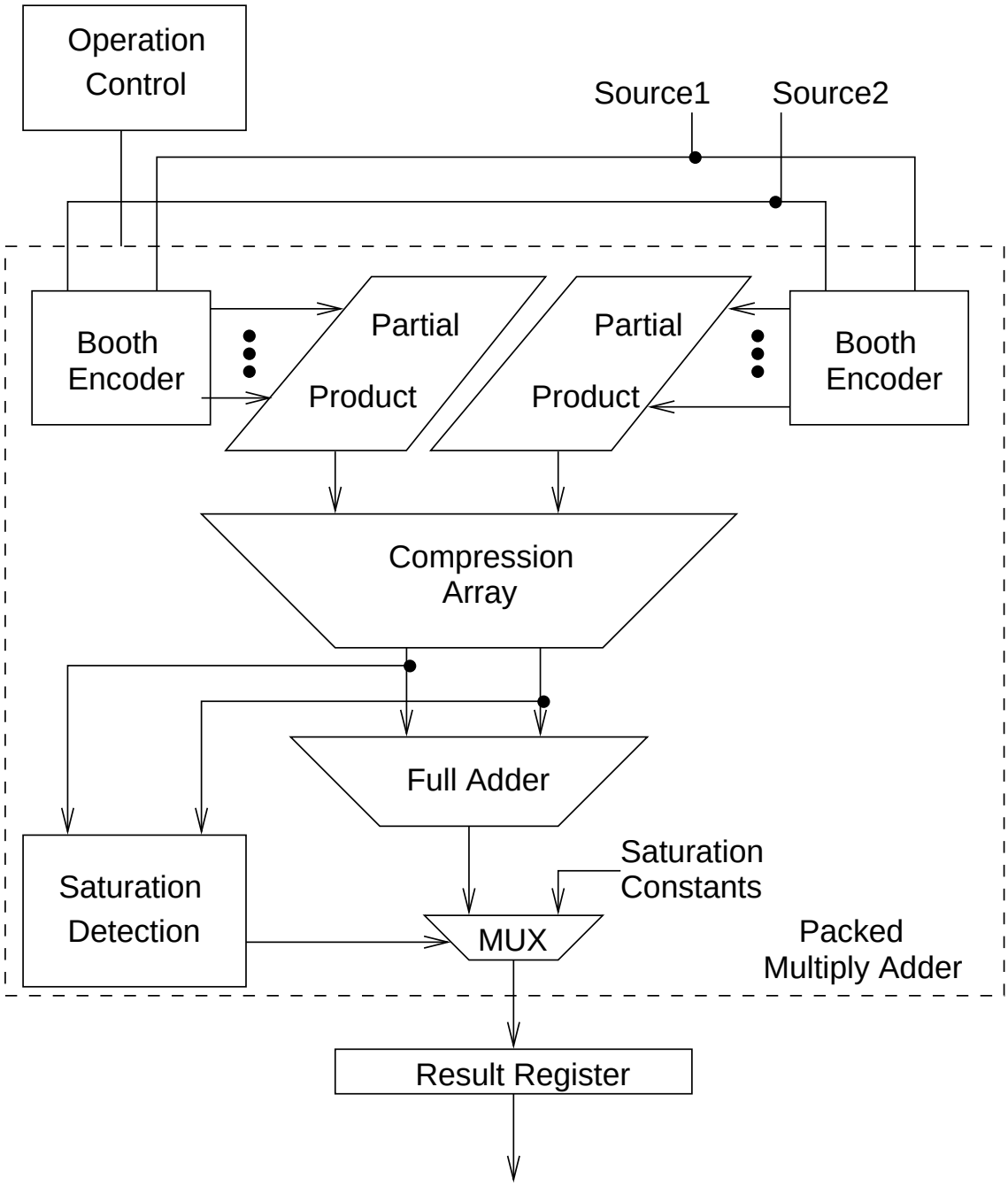


Figure 2.29 Multiply-Add Design for Packed Data (Debes, Macy, Tyler, Peleg, Mittal, Mennemeier, Eitan, Dulong, Kowashi, Witt, 2008).

Table 2.5 Multiply-Accumulate

Multiply-Accumulate Source 1, Source 2	
A_1	Source 1
B_1	Source 2
=	
$A_1 \cdot B_1$+accumulated Value	Result 1

Table 2.6 Packed Multiply-Add

Packed Multiply-Add Source 1, Source 2				
A_1	A_2	A_3	A_4	Source 1
B_1	B_2	B_3	B_4	Source 2
		=		
$A_1 \cdot B_1 + A_2 \cdot B_2$		$A_3 \cdot B_3 + A_4 \cdot B_4$		Result 1

Table 2.7 Packed Multiply-Subtract

Packed Multiply-Subtract Source 1, Source 2				
A_1	A_2	A_3	A_4	Source 1
B_1	B_2	B_3	B_4	Source 2
		=		
$A_1 \cdot B_1 - A_2 \cdot B_2$		$A_3 \cdot B_3 - A_4 \cdot B_4$		Result 1

data at the same time. Figure 2.29 shows the details of packed multiply-add/subtract operation. The operation control unit enables the circuit. The packed multiply-add/subtract circuit contains 16 by 16 multiplier circuits and 32-bit adders. The first 16 by 16 multiplier contains a booth encoder, which has inputs Source1[63:48] and Source2[63:48]. Booth encoder selects partial products depending on its inputs. The second 16 by 16 multiplier also contains a booth encoder, which has inputs Source1[47:32] and Source2[47:32]. This booth encoder also selects partial products depending on its inputs. The booth encoders are used to select a partial products. For example, partial product of zero, if Source1[47:45] are 000 or 111; Source2[47:32], if Source1[47:45] are 001 or 010; 2 times Source2[47:32], if Source1[47:45] are 011; negative 2 times Source2[47:32], if Source1[47:45] are 100; or negative 1 times Source2[47:32], if Source1[47:45] are 101

or 110. Like this, Source1[45:43], Source1[43:41], Source1[41:39], etc can be used to select respective partial products.

Partial products are routed to compression array. Here partial products are aligned in according to Source1. The compression array may be implemented as a Wallace tree structure of carry-save adders or a sign-digit adder structure. The results are then routed to adder. Depending on operation, compression array and adders do addition or subtraction. The results routed to result register for formating output.

2.11 Multiplier Structure Supporting Different Precision Multiplication Operations

This is a multiplier design that can perform operation on both integer and floating point operands. The multiplier is design as sub-tree form, so it can be configure as single-tree structure for non-SIMD or partitioned into 2 or 4 for SIMD operation. The design can be seen from Figure 2.30. The figure also shows various ways of multiplier partition structure.

When multiplier is configured for 4 partitions, 4 multiplications executed simultaneously on independent data. When multiplier is configured for 2 partitions, two 32-bit similar structures Tree AB in Figure 2.30 is multiplied. When multiplier is not partitioned then combined 64-bit structure TreeABCD in Figure 2.30 is multiplied. Various partitioning tree structures can be formed in order to support different multiplier structures.

The data flow can be summarized as: First, partial products are generated by Wallace tree structure for each bit in the multiplier, then partial products are summed with carry-save adders (CSA). In binary number system, the multiplier can be either one or zero, that means the product is either $1 \times multiplicant$ or $0 \times multiplicant$. The number of partial product that is going to be added is related with non-zero bits in the multiplier. Booth encoding is used to reduce the number of partial products. Booth encoding uses two side by side bits as well as MSB(Most Significant Bit) of the previous two bits to determine the partial product.

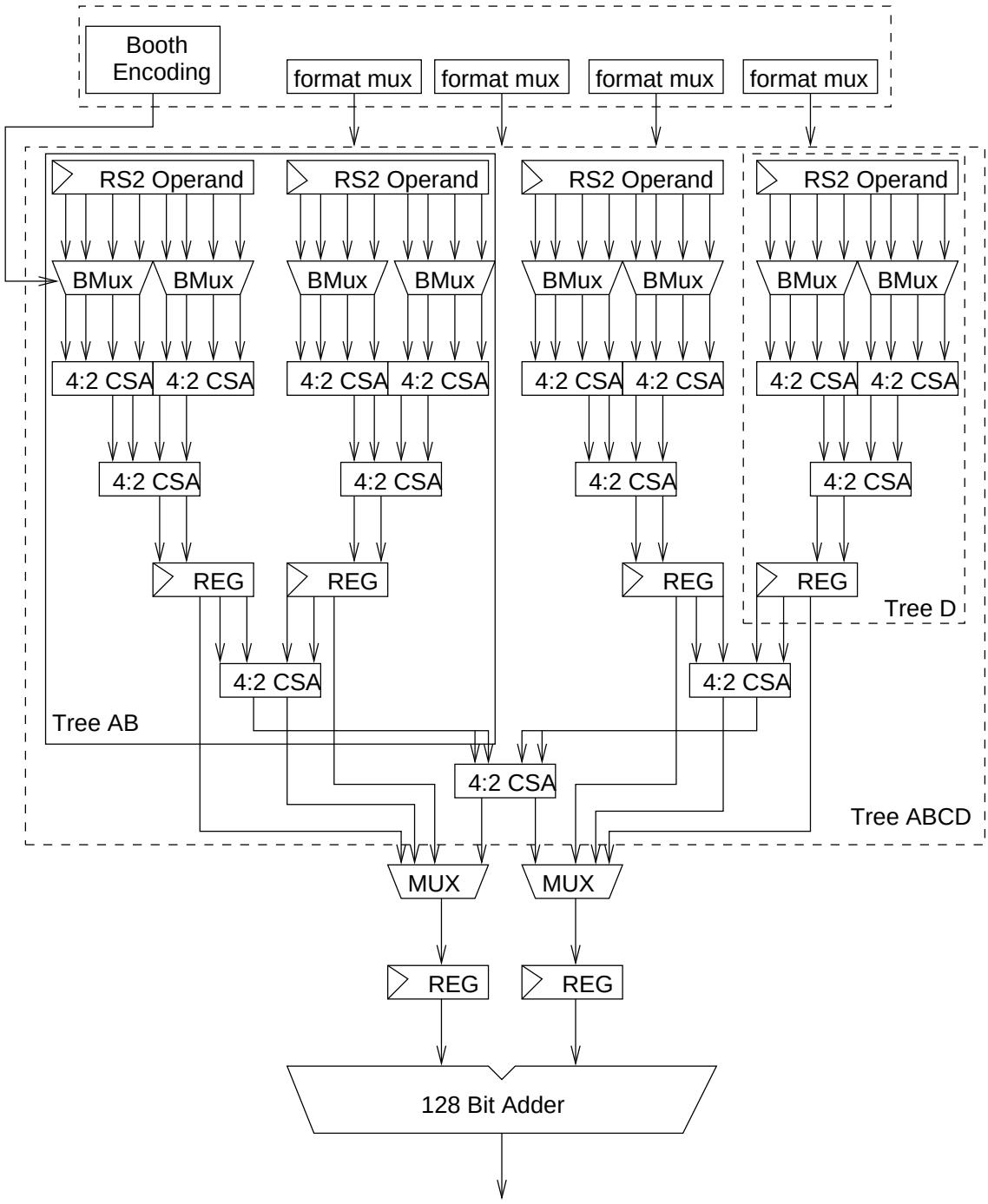


Figure 2.30 Multiplier Structure Supporting Different Precision Multiplication Operations(Jagodik, Brooks, Olson, 2008).

2.12 Method and Apparatus for Calculating:Reciprocals and Reciprocal Square Roots

The design is a part of microprocessor design from AMD Inc. The design gives processor capability of evaluating reciprocal and reciprocal square root of operand. The processor has a multiplier that can be used to perform iteration operations needed. The design uses two path one, assumes that overflow has occurred, other assumes that no-overflow has occurred. The intermediate results are stored for next iteration. The general form of design is shown in Figure 2.31.

The design utilities division operation over reciprocal and multiplication operation. The operation is formulated as $A \times B^{-1}$ where A is dividend and B is divisor. The reciprocal of divisor is realized using a version of Newton and Raphson iteration. The iteration Equation used for calculation of reciprocal of B is

$$X_1 = X_0 \times (2 - X_0 \times B) \tag{2.43}$$

The iteration needs an initial estimation X_0, which can be determined from a ROM(Read Only Memory). Once X_0 is determined, it is multiplied by B. After multiplication, the term $(2 - X_0 \times B)$ is formed, by inverting the term $(X_0 \times B)$. One's complement is used to speedup the calculation. The corresponding sign and exponent bits are also computed along the mantissa computation. The approximations for $(2 - X_0 \times B)$ are performed in parallel by each path. Using double path may be save time in normalization by without needing normalization bits. After this step, the result is passed back to multiplier to complete the iteration by multiplying with X_0. If the desired accuracy is reached, the results are output. If desired accuracy is not reached, the iteration is repeated. The results of the multiplication are once passed down the paths in parallel. The accuracy is depended on initial guess X_0.

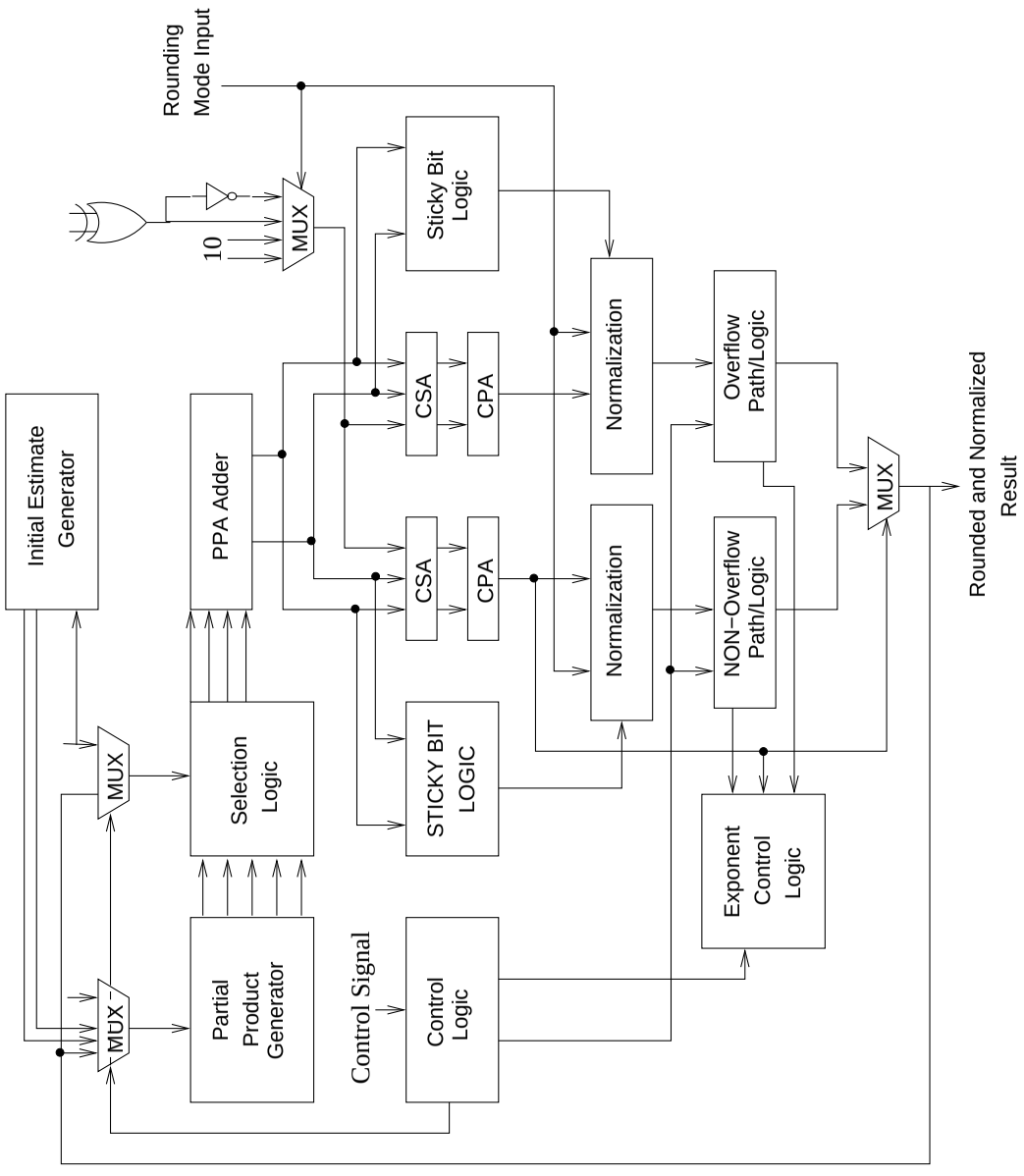


Figure 2.31: Reciprocal and Reciprocal Square Root Apparatus (Oberman, Juffa, Weber, 2000).

3. THE PROPOSED FLOATING POINT UNITS

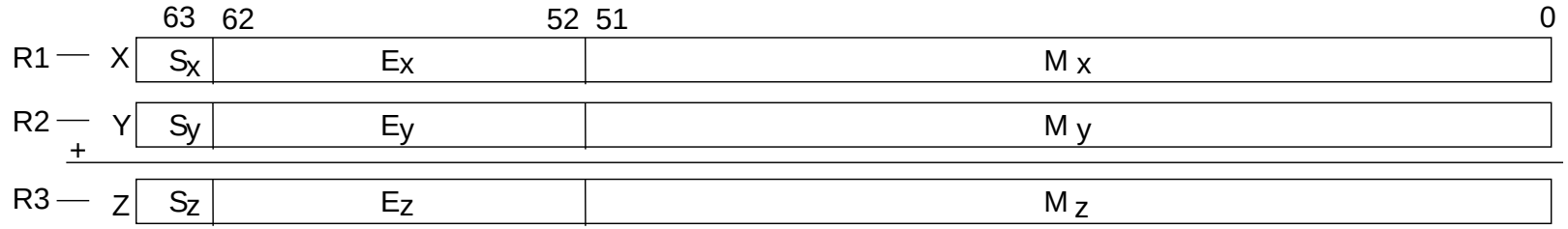
This section presents the floating point designs for multimedia processing. The following designs are discussed in detail: Multi-Precision Floating Point Adder, Double/Single Floating Point Multiplier, Multi-Functional Double Precision Floating Point MAF, Multi-Functional Quad Precision Floating Point MAF and Multi-Precision Floating Point Reciprocal Unit.

3.1 The Multi-Precision Floating-Point Adder

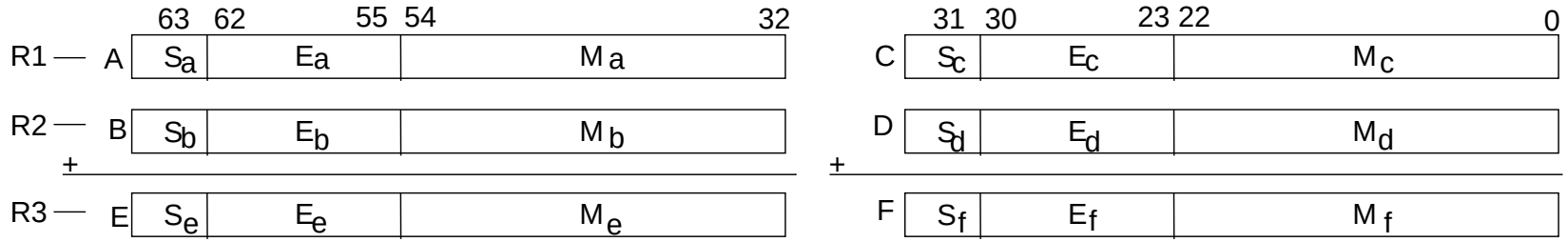
The proposed multi-precision adder can operate on double, single and half precision numbers. In single precision addition mode two simulteneous floating point additions are performed. In half-precision addition mode four simultanous floating point additions are performed.

The input operands for the multi-precision adder are packed based on the operation mode. Figure 3.1 presents the alignments of double, single, and and half precision floating-point numbers and their sums in three 64-bit registers $R1$, $R2$, $R3$. The registers are used for demonstration purpose they are not a part of the actual implementation. In Figure 3.1.a, three double precision floating-point numbers X, Y and their sums are shown. In Figure 3.1.b, four single-precision floating-point numbers A, B, C, D and their sums, E, and F are shown. In Figure 3.1.c, eight half-precision floating-point numbers K, L, M, N, P, R, S, T, and their sums I, O, Q, and V are shown in NVIDIA half-precision format (Nvidia, 2007). The half-precision format described by NVIDIA is not included in the IEEE-754 standard, however, it is widely used in graphics processing applications.

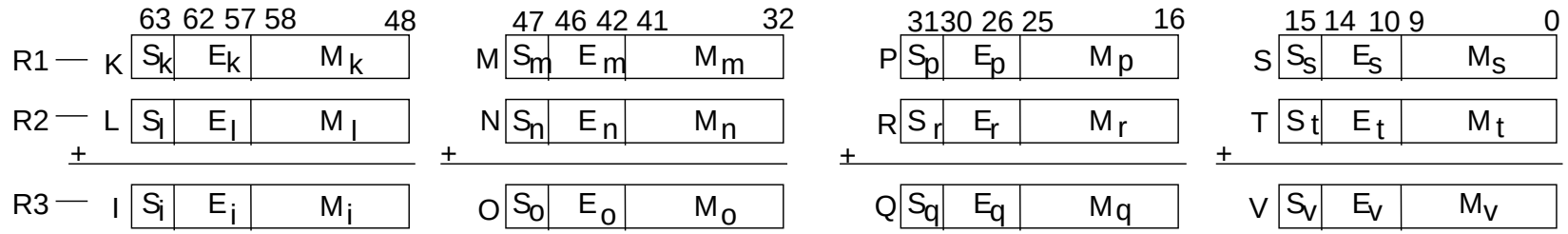
Figure 3.2 presents the block diagram for the proposed multi-precision floating point adder. The design of this adder is based on a modified version of the single-path floating point adder presented in (Ercegovac and Lang, 2004). The mode of operation is selected by using a control signal, M. When $M = 01$ (Mode 1), a double-precision floating-point addition is performed. When $M = 10$ (Mode 2), two parallel single-precision floating-point additions are performed. When $M = 11$ (Mode 3), four parallel half-precision floating-point additions are performed. EOP represents the effective operation. To reduce



(a) Double Precision Floating−Point Number



(b) Single Precision Floating−Point Numbers



(c) Half Precision Floating−Point Numbers

Figure 3.1: The Alignments of Double, Single, and Half Precision Floating-Point Numbers.

the complexity of the figure, the inputs of the units in Figure 3.2 are plainly designated as $R1$ and $R2$. In the actual implementation only the parts of the vectors that are used in the unit are connected. The location of these parts can be observed from Figure 3.1. The functionality of the main units and data flow are explained as follows:

The *exponent subtracter* unit computes the differences of the operands' exponents in all modes. These differences are used to align the operands. The signs of the differences are used in the *Swap* unit for decision of small operand. The *Swap* unit changes the places of the mantissa if the sign of the difference is negative. By this way only the mantissa with the smaller exponent is right-shifted. Based on the operation mode, the *swap* unit operates on different operands. The *Compare* unit compares the magnitudes of the operands when the difference or differences between the exponents are zeros. Then informs the *swap* unit for smaller operand. The *Bit Invert* unit inverts the mantissa (or mantissas) with the smallest exponent so that the result (or results) is always positive. The addition of 1 ulp required for twos complement conversion is performed in the mantissa adder. The *Mantissa Generator* unit prepares the mantissa bits for operation in all modes. The mantissas are converted into two's complement format and they are also shifted for alignment. The *mantissa adder* is a two's complement adder that can perform an addition on 53-bit operands or two parallel additions on 24-bit operands, or four parallel additions on 10-bit operands. The signs of the results are generated in the *Mantissa Adder*. The *Leading One Detector (LOD)* units compute the number of right-shifts to normalize the result when the EOP is a subtraction. LOD 1 operates in all modes, LOD 2 operates in Modes 2 and 3, and LOD 3 operates only in Mode 3. LOD 3 operates on two half-precision operands in Mode 3. The *Normalize* units are normalizing shifters. The mantissas are either left shifted with the amount determined in LOD units or right shifted by one digit when addition overflow occurs. The *Flag* units determine the rounding flags with respect to the rounding mode that is selected. Since all IEEE-754 rounding modes are supported a flag for each rounding mode is generated. The *Rounding* units perform the addition of 1 ulp when it is necessary to perform rounding. These cases are indicated by the flags generated by the *Flag Units*. The overflow due to the addition in rounding units is also checked here and adjustment shift is performed when necessary. The *Exponent Update* units update the exponent strings which are prepared in exponent generator unit. The *Sign* unit generates

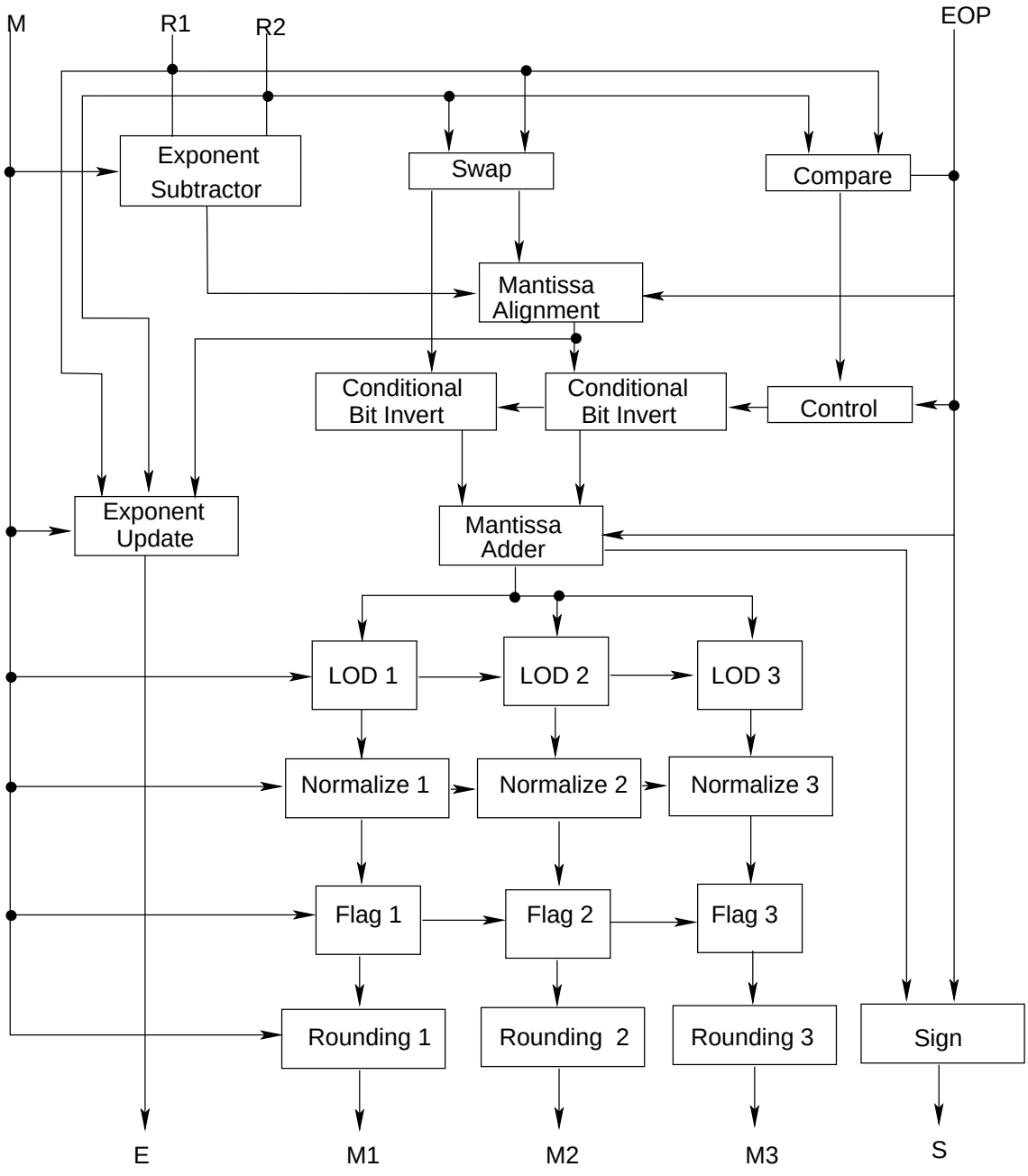


Figure 3.2 The Block Diagram of Multi-Precision Floating-Point Adder.

the sign of the result or results based on the signs of the operands with greater magnitude. The sign, exponent and mantissa of the result (or results) are represented as S, E, M, respectively.

3.2 The Single/Double Precision Floating-Point Multiplier Design

This section presents a new floating-point multiplier which can perform a double-precision floating-point multiplication or two simultaneous single precision floating-point multiplications. Since in single precision floating-point multiplication two results are generated in parallel, the multiplier's performance is almost doubled compared to a conventional floating-point multiplier. Figure 3.3.a shows the alignments of two double precision floating point numbers X, Y and their product Z that are placed in three 64-bit registers. Figure 3.3.b shows the alignments of four single precision floating-point numbers A,B,C and D and the product of A and B, E, and the product of C and D, F that are placed in three 64-bit registers.

The multiplication of X and Y is performed as

$$E_z = E_x + E_y \tag{3.1}$$

$$M_z = M_x \times M_y \tag{3.2}$$

$$S_z = S_x \otimes S_y \tag{3.3}$$

The multiplication of A and B, and the multiplication of C and D are performed as

$$E_e = E_a + E_b, \tag{3.4}$$

$$E_f = E_c + E_d \tag{3.5}$$

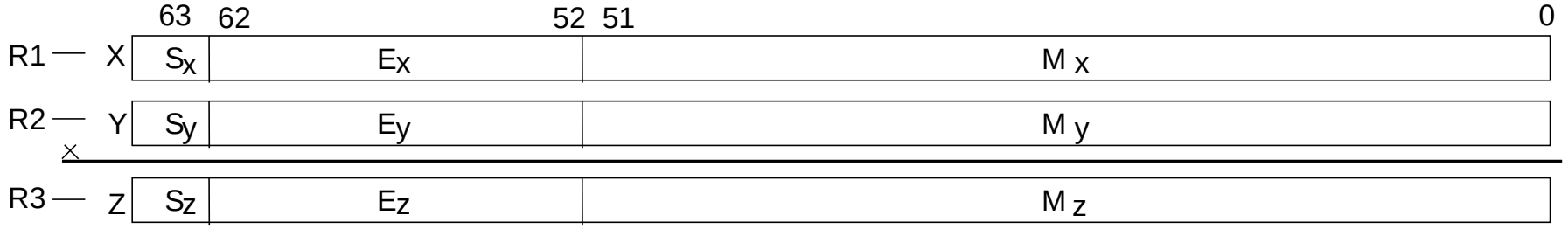
$$M_e = M_a \times M_b, \tag{3.6}$$

$$M_f = M_c \times M_d \tag{3.7}$$

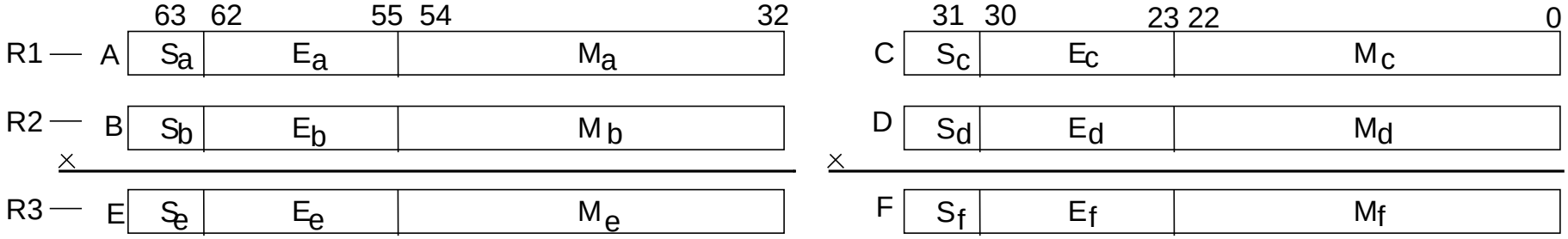
$$S_e = S_a \otimes S_b, \tag{3.8}$$

$$S_f = S_c \otimes S_d \tag{3.9}$$

The proposed design performs these two floating-point multiplications in parallel. In (Gok, Krithivasan and Schulte, 2004) a design method for the multiplication of two unsigned integer operands is presented. Figure 3.4 presents the adaptation of that technique



(a) Double Precision Floating−Point Number



(b) Single Precision Floating−Point Numbers

Figure 3.3: The Alignments for Double and Single Precision Numbers.

to implement the proposed method. In this figure, the matrices generated for the two single precision floating-point multiplications are placed in the matrix generated for a double precision floating-point multiplication. All the bits are generated in double precision mode, the shaded area Z is not generated when single precision multiplication is performed, the non-shaded regions designate the generated bits.

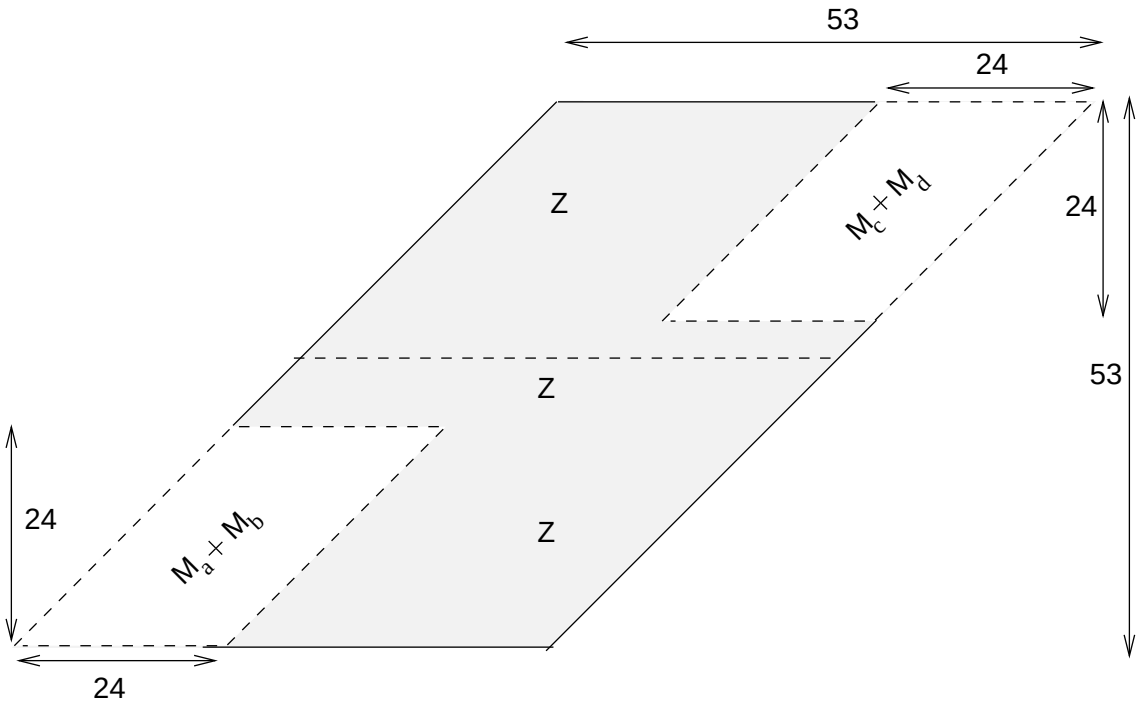


Figure 3.4 The Multiplication Matrix for Single and Double Precision Mantissas.

The partial products within the regions Z are generated using the following equations

$$\widehat{b}_j = s \cdot b_j \text{ and } p_{ij} = a_i \cdot \widehat{b}_j \tag{3.10}$$

and the rest of the partial products are generated with

$$p_{ij} = a_i \cdot b_j \tag{3.11}$$

s is used as control signal. When $s = 0$, only the bits in no shaded regions are generated otherwise all bits are generated. The i and j are respective matrix indexes.

High-speed multipliers reduce the partial product matrix to two vectors using a reduction method. Then, these two vectors are added to produce the result with carry-propagate adder. The reduction method and the type of the carry-propagate adder are not important for the proposed design, since it only modifies the generation of the partial products. This

also means that the reduction algorithm and the carry-propagate adder is not modified for the implementation of the proposed method.

The standard floating-point multiplier, which is mentioned in section 2.3 implements Equation 3.1 to Equation 3.3. Figure 3.5 presents the proposed single/dual floating-point multiplier which is designed by slightly modifying the standard floating-point multiplier. The modifications can be used on every type of double precision floating-point multiplier.

The data flow and the functionality of each unit in the proposed design are explained as follows: The *Control Signal* determines the mode of execution; when $s = 0$ double precision floating-point multiplication is performed, otherwise two single precision multiplications are performed. A 11-bit adder is used for double precision exponent addition and two 8-bit adders are used for single precision exponent additions. The *Exponent Updaters* remove extra bias values form the exponent sums. The *Mantissa Modifier* selects the appropriate mantissas to be send to the mantissa multiplier. The *Mantissa Multiplier* generates carry-save vectors. The *Add*, *Normalize* and *Round* unit generates normalized and rounded result or results. The signs of the products are obtained by XOR gates.

3.3 The Multi-Functional Double-Precision FPMAF Design

The Multi-functional double-precision $FPMAF$ design supports three modes named as double-precision multiplication (*DPM*), single-precision multiplication (*SPM*) and dot-product (*DOP*).

1. In *DPM* mode, the design works as a double-precision $FPMAF$ unit. It computes $XD \cdot YD + ZD$,where XD, YD and ZD are double-precision floating-point operands.

2. In *SPM* mode, the design works as a single-precision floating-point multiplier and computes $AS \cdot BS$ and $CS \cdot DS$ in parallel, where AS, BS, CS and DS are single-precision floating-point operands. This mode has two advantages: first, the latency for performing two single-precision multiplications is approximately the same as the latency for performing one double-precision multiplication. The second advantage is that there is no need to convert operands from single to double-precisions back and forth.

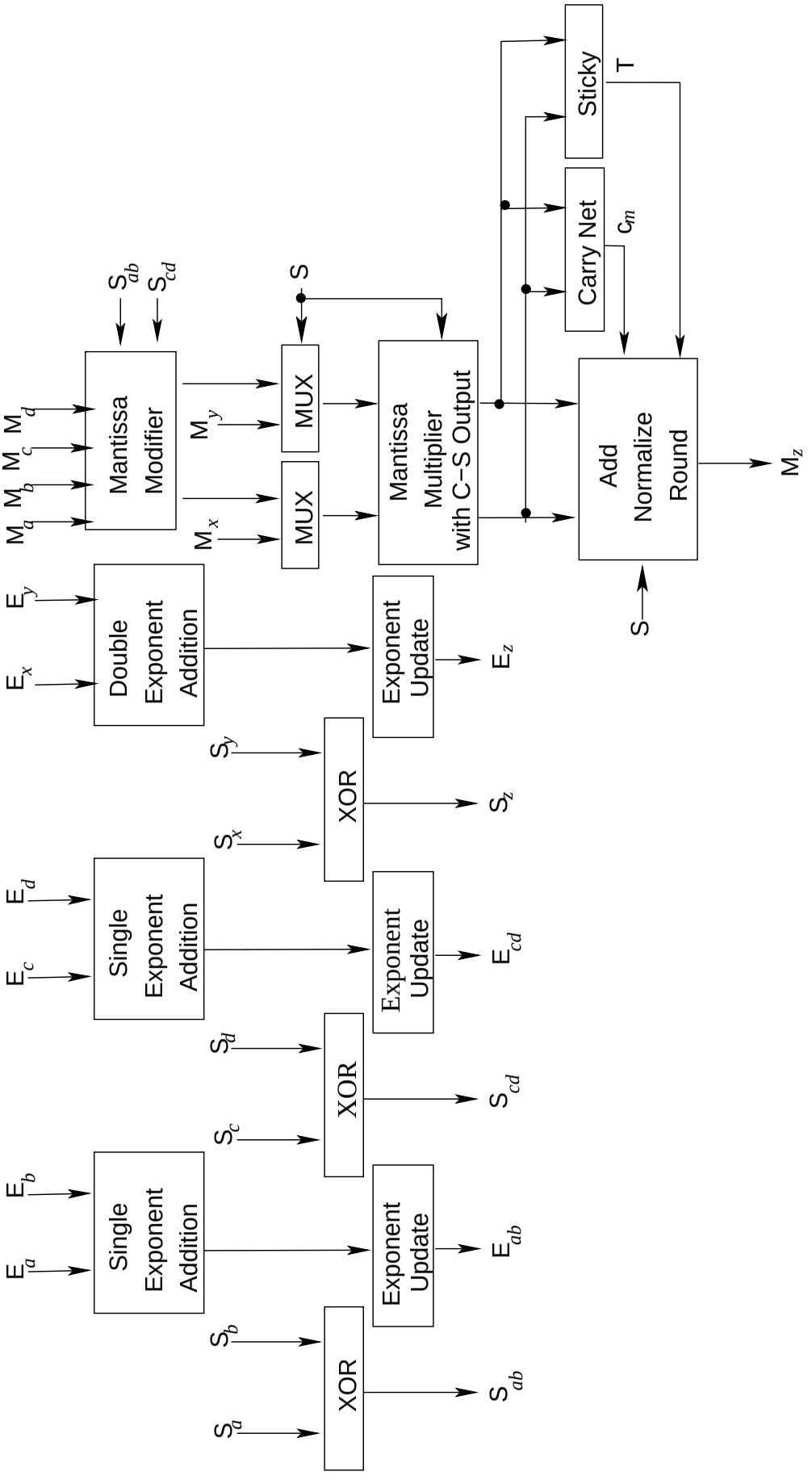


Figure 3.5: The Block Diagram for the Proposed Floating Point Multiplier.

3. In *DOP* mode, the design works as a dot-product unit, and performs two single-precision floating-point multiplications in parallel and then adds the products of these multiplications with a single-precision operand. This operation can be expressed as $AS \cdot BS + CS \cdot DS + US$. By setting appropriate operands to 0 and 1,a two-operand or a three-operand single-precision floating-point addition, or a single-precision floating-point multiply-add can be performed.

3.3.1 The Mantissas Preparation step

Figure 3.6 shows the alignments of the three double-precision and five single-precision IEEE-754 floating-point operands in 64-bit registers, R1, R2, and R3. These registers are used for demonstration purpose, they are not a part of the actual design. The double-precision format is used in *DPM* mode, and the single-precision format is used in *SPM* and *DOP* modes. Based on the execution mode, the initial mantissas are modified before they are input to the mantissa multiplier. The modified mantissas (named as $M1$ and $M2$) are differently generated for each mode.

In DPM mode, the inputs for mantissa multipliers are produced as

$$DPM(M1) = 1 \ \& \ R1_{51:0} \tag{3.12}$$
$$DPM(M2) = 1 \ \& \ R2_{51:0}$$

where

'1's are the concatenated hidden bits described by IEEE-754 standard(IEEE, 1985).

'&' represents the concatenation operator

$R1_{51:0} = M_x$

$R1_{51:0} = M_y$

Figure 3.7 shows the 53 by 53 mantissa multiplication matrix generated for *DPM* mode. All the partial product bits in this matrix contribute the generation of the product.

In SPM mode, two versions of $M1$ and one version of $M2$ are produced. The first version of $M1$ is designated as $M1_{UH}$. The least-significant 26 bits of $M2$ and $M1_{UH}$ are used to generate the upper half of the 53 by 53 multiplication matrix. These vectors are

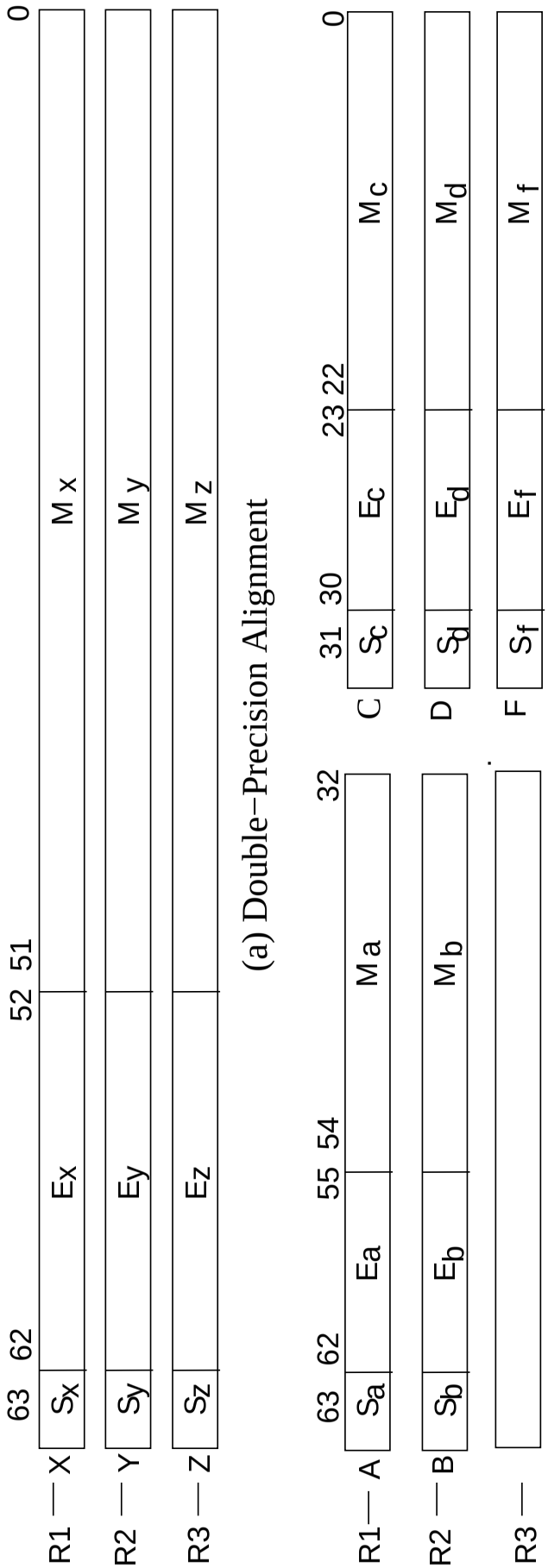


Figure 3.6: The Alignments of Double and Single Precision Floating-Point Operands in 64-bit Registers.

produced as

$$SPM(M1_{52:0})_{UH} = \{0\}^{29} \,\&\, 1 \,\&R1_{22:0} \tag{3.13}$$
$$SPM(M2_{25:0}) = 001 \,\&R2_{22:0}$$

where

$\{0\}^{29}$ represents 29 instances of 0,

$R1_{22:0} = M_c$,

$R2_{22:0} = M_d$.

The second version of $M1$ is designated as $M1_{LH}$. The most significant 27 bits of $M2$ and $M1_{LH}$ are used to generate the lower half of the 53 by 53 matrix. These vectors are produced as

$$SPM(M1_{52:0})_{LH} = 1 \,\&R1_{54:32}\,\{0\}^{29} \tag{3.14}$$
$$SPM(M2_{52:26}) = 1 \,\&\, R2_{54:32} \,\&\, 000$$

where

$R1_{54:32} = M_a$

$R2_{54:32} = M_b$.

Figure 3.7.b shows the multiplication matrix generated for *SPM* mode. In this figure, the partial product bits located inside the regions designated by Z are set to zeros. The unshaded regions contain the matrices generated for the multiplications; $(1 \,\&\, M_a) \cdot (1 \,\&\, M_b)$, and $(1 \,\&\, M_c) \cdot (1 \,\&\, M_d)$.

The main idea for *DOP* implementation is performing the addition of the products by only using the adders in the partial product reduction tree. The application of this idea requires a little more complex modifications than the modifications for the previous modes. In *DOP* mode, the upper half of the matrix is generated using

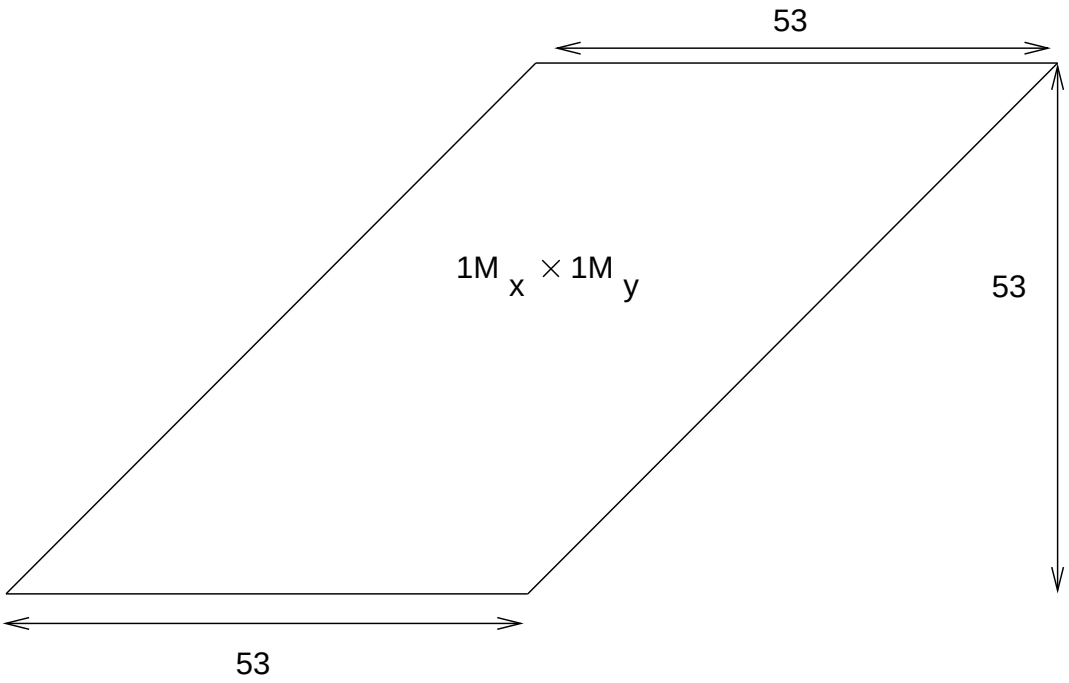
$$\begin{aligned} DOP(M1_{52:0})_{UH} &= \{R1_{31} \oplus R2_{31}\}^{d} \,\&\, 1 \,\&\, R1_{22:0} \,\&\, \{0\}^{29-d} \\ DOP(M2_{25:0}) &= 001 \,\&\, R2_{22:0} \end{aligned} \tag{3.15}$$

where

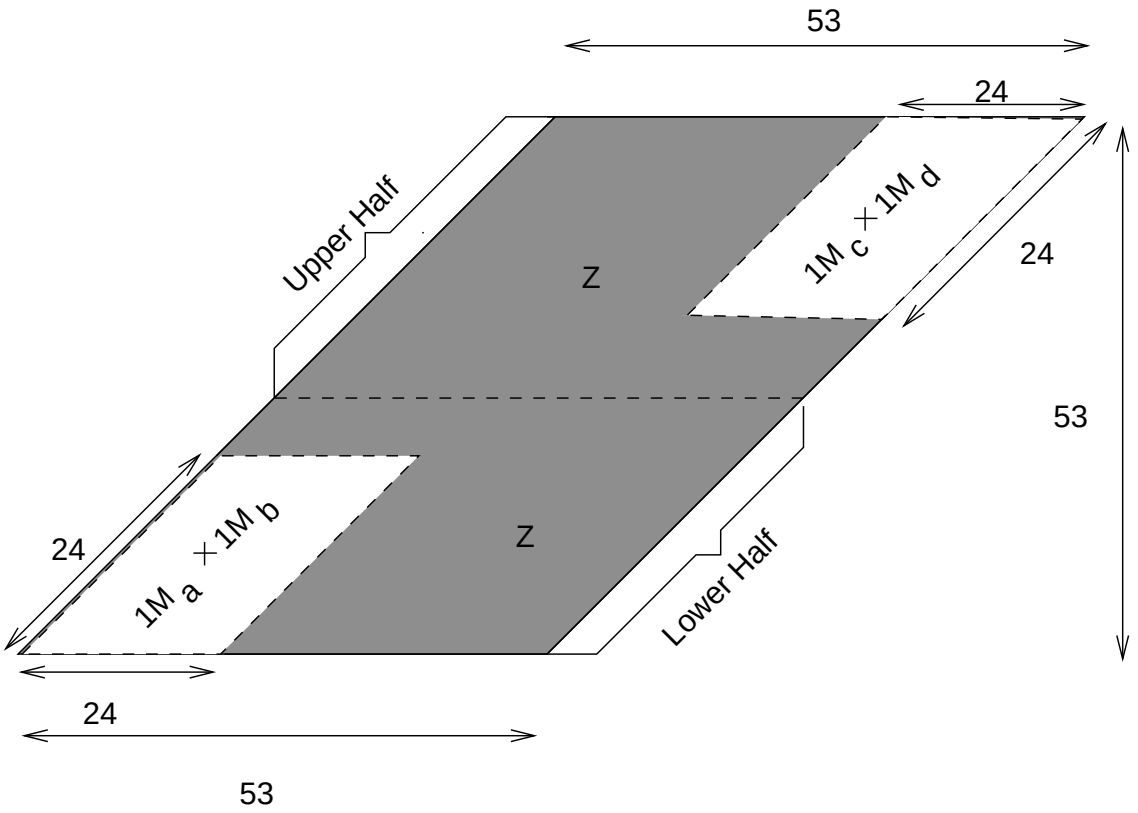
$d = \mid E_{ab} - E_{cd} \mid$

$E_{ab} = E_a + E_b - 127$

$E_{cd} = E_c + E_d - 127$.



(a) DPM Mode



(b) SPM Mode

Figure 3.7 The Partial Product Matrices Generated for (*DPM*) and (*SPM*).

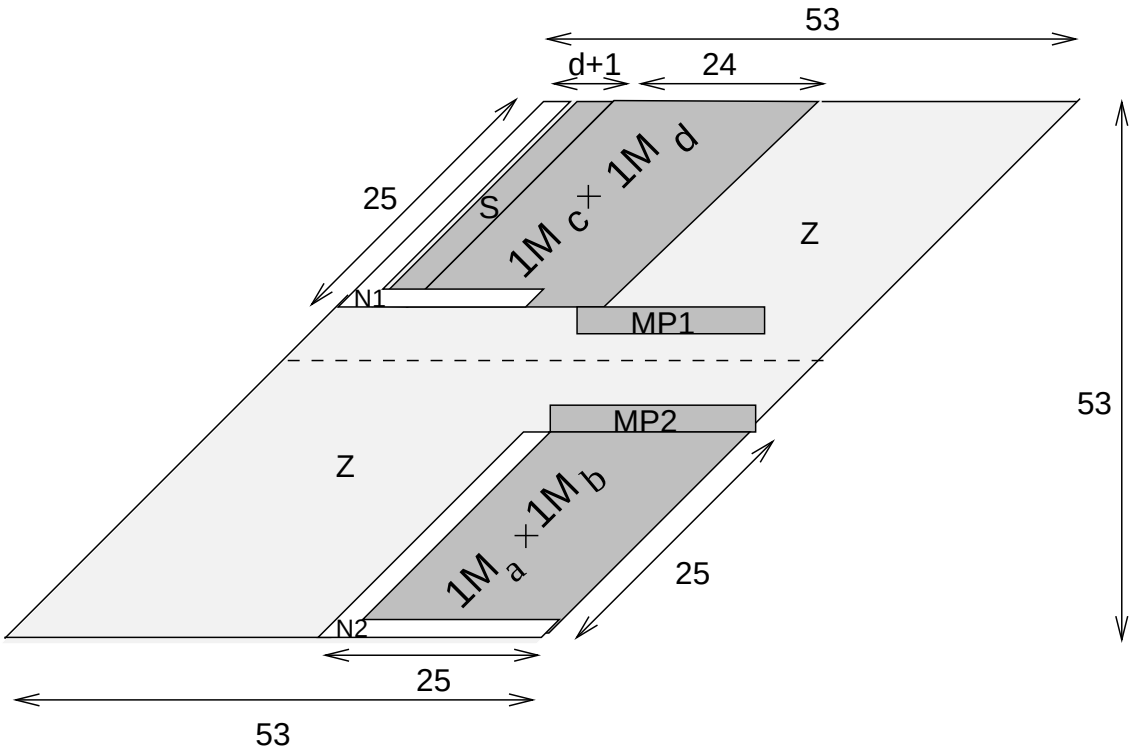


Figure 3.8 The Matrix Generated for (*DOP*) Mode.

Without loss of generality, in Equation 3.16, it is assumed that $E_{cd} \leq E_{ab}$. The lower half of the multiplication matrix is generated using

$$\begin{aligned} DOP(M1_{52:0})_{LH} &= \{0\}^{29} \,\&\, 1 \,\&\, R1_{54:32} \\ DOP(M2_{52:26}) &= 01 \,\&\, R2_{54:32} \,\&\, 00 \end{aligned} \tag{3.16}$$

Figure 3.8 presents the multiplication matrix generated for *DOP* mode. In addition to the mantissa modifications described by Equation 3.16 and Equation 3.17 following adjustments are made.

The operands are extended by one bit and converted into two's complement format when their sign bits are different. By this way, the addition of the partial products can be performed without considering the signs of the operands (i.e. no need to consider the effective operation). To prevent a performance decrease due to two's complement conversion, the mantissa with the negative sign is selected as the multiplicand, then it's bits are inverted and a copy of the positive mantissa (the multiplier) is inserted into the

multiplication matrix. These operations can be expressed as

$$(\overline{MN}+1)\cdot MP = (\overline{MN}\cdot MP)+MP \tag{3.17}$$

where MN and MP represent the negative and positive mantissas, respectively. In Figure 3.8, $MP1$ and $MP2$ vectors are injected into the matrix to perform the addition of the positive mantissas. $MP1$ and the upper 25 by 25 matrix are shifted together.

The two's complement multiplication algorithm presented in (Baugh and Wooley, 1973) is used to prevent the sign extension of the partial products. This algorithm requires $2n-2$ bits to be complemented. The complemented bits are located inside the dark gray shaded areas, $N1$ and $N2$ in Figure 3.8. The bits in $N1$ and $N2$ regions are not shifted.

The 25 by 25 matrix with the smaller exponent is moved in the upper half, and right shifted by d columns. The region S is filled by zeros, if the sign of the operands are the same, otherwise, it is filled by ones. So, the addition of the bits in S does not effect the result.

3.3.2 The Implementation Details for Multi-Functional Double-Precision *FPMAF* Design

The proposed design is implemented by mainly using the hardware of the standard double-precision floating-point multiplier. Naturally, some extra hardware is used to support additional operation modes, however, this extra hardware is significantly less than the hardware required to design a different unit for each mode. The block diagram for the proposed multi-functional *FPMAF* design is shown in Figure 3.10. Although some of the units in the design can be combined, this approach is not preferred for double-precision implementation to keep the organization simple. The design is divided into four pipeline stages. Except the first stage, the stages are similar to the basic double-precision *FPMAF* design. The function of each block and the data flow between stages are explained as follows:

The mantissa bits are modified in the first stage. The control signals T_1 and T_0 are used to select the operation mode which is given in Table 3.1. The function of the each unit in this stage is explained as follows:

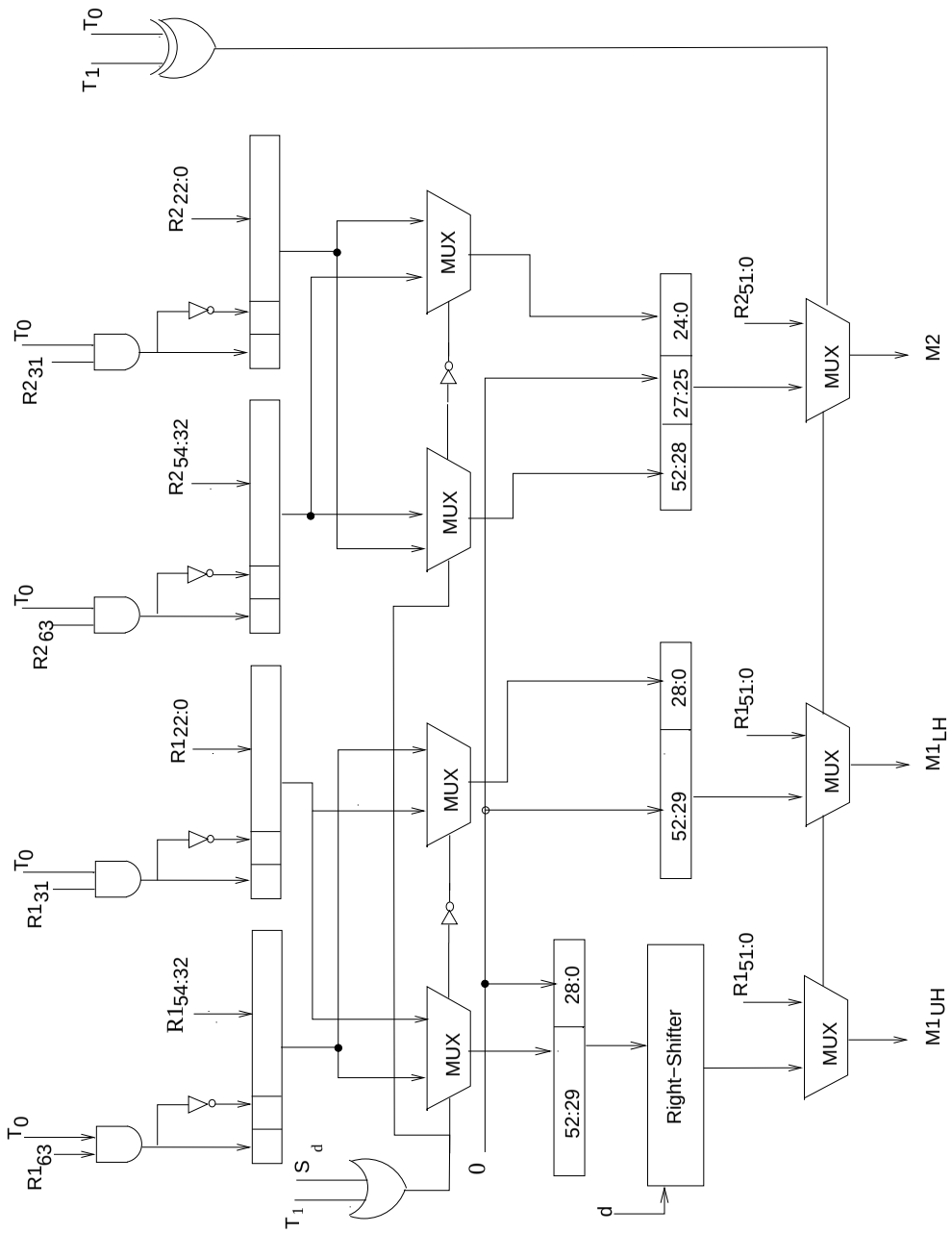


Figure 3.9: The Mantissa Modifier Unit in the Double Precision *FPMAF*.

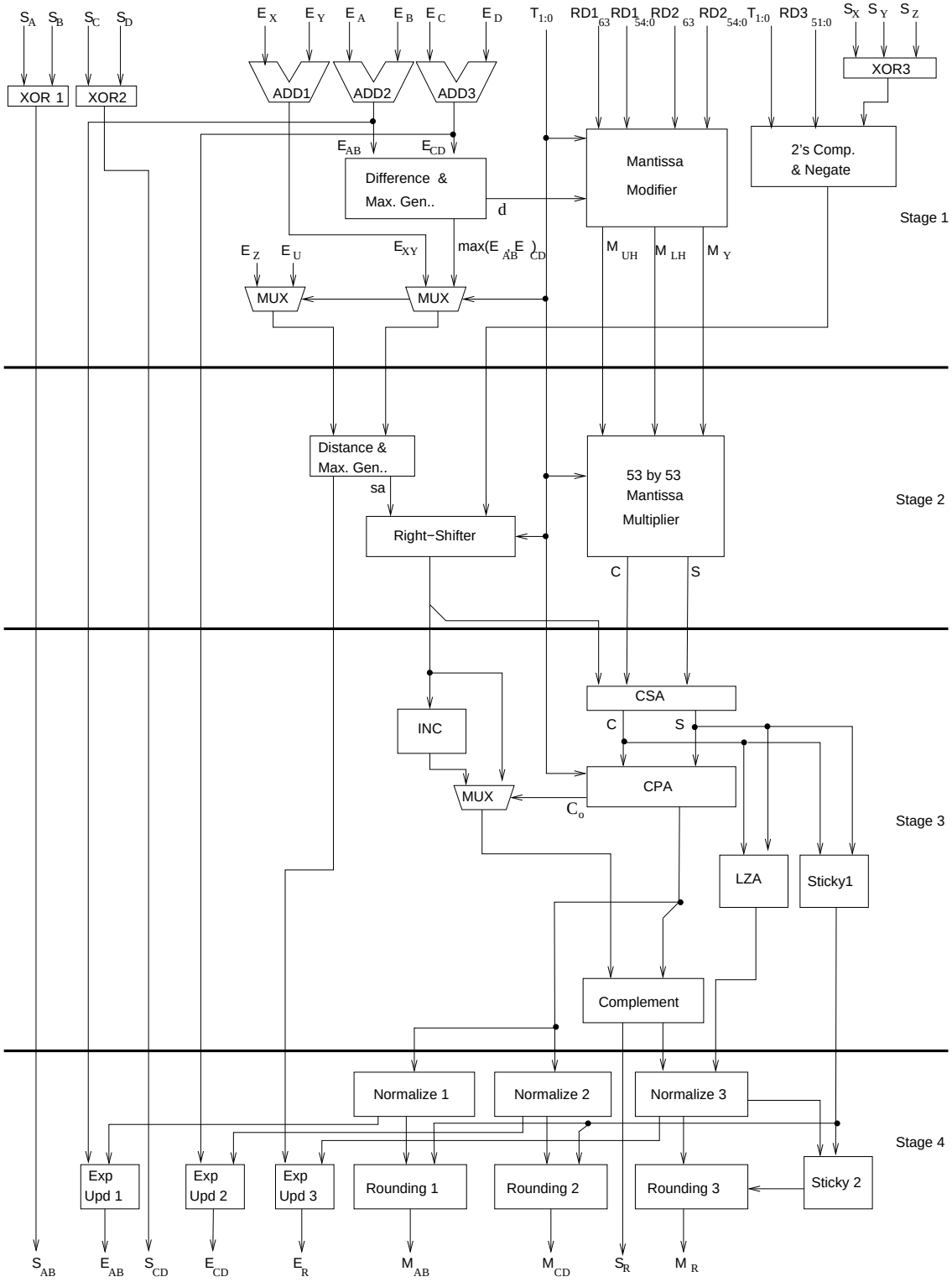


Figure 3.10 The Block Diagram for Multi-Functional Double Precision *FPMAF* Design.

Table 3.1 The Execution Modes

T_1T_0	Operation
00	DPM
10	SPM
01	DOP
11	NAN

The *XOR1* and *XOR2* gates compare the signs of the operands in *SPM*. The *XOR3* gate compares the signs of the operands in *DPM* mode, the output of this gate is sent to *2's Comp. & Negation Unit*. There is no need to compare the signs of the operands in *DOP* mode, since the operands are in two's complement format in this mode.

The 11-bit adder (*ADD1*) computes $E_{xy} = E_x + E_y - 1023$ where $R1_{62:52} = E_x$ and $R2_{62:52} = E_y$. The first 8-bit adder (*ADD*2) computes $E_{ab} = E_a + E_b - 127$, where $R1_{62:55} = E_a$ and $R2_{62:55} = E_b$. The second 8-bit adder (*ADD*3) computes $E_{cd} = E_c + E_d - 127$, where $R1_{30:23} = E_c$ and $R2_{30:23} = E_d$.

The *Difference and Maximum Generator Unit* computes $d = \mid E_{ab} - E_{cd} \mid$ and $max(E_{ab}, E_{cd})$. d is sent to the *Mantissa Modifier Unit.* Two 2-input multiplexers select the correct inputs to the *Distance and Maximum Generator Unit* (located in the second stage.)

The *Mantissa Modifier Unit* shown in Figure 3.9 generates the modified mantissas using Equations (3.14)-(3.17) for all modes. This unit consists of a 32-bit right-shifter (that can shift up to 29 digits) and several multiplexers and glue logic. The inputs to the *Mantissa Modifier Unit* are $R1_{63}$, $R1_{54:0}$, $R2_{63}$, and $R2_{54:0}$. Based on the multiplication mode, these vectors contain the mantissas and sign bits as follows, $M_x = R1_{51:0}$, $M_y = R2_{51:0}$, or $M_a = R1_{54:32}$, $M_b = R2_{54:32}$, $M_c = R1_{22:0}$, and $M_d = R2_{22:0}$, and the sign bits $S_x = R1_{63}$, $S_y = R2_{63}$, or $S_a = R1_{63}$, $S_b = R2_{63}$, $S_c = R1_{31}$, and $S_d = R2_{31}$.

The *2's Comp. & Negation Unit* negates the addend M_z or M_u based on the multiplication mode and the sign comparison of the operands. In *DPM* mode, if S_z is different than $S_x \oplus S_y$, M_z is negated. In this case, the correct sign of the result is determined later by comparing the signs of the operands and the sign of the output of the *CPA*. In *DOP* mode, M_u is converted into two's complement format.

The functions of the units located in this stage are explained as follows:

The modified mantissas are multiplied by the *Mantissa Multiplier*. The generation of partial products in the multiplier is slightly modified to implement the insertion of $MP1$ and $MP2$ vectors and to perform the inversion of the bits in regions $N1$ and $N2$ in *DOP* mode. The rest of the multiplier hardware is not modified. The *Mantissa Multiplier* generates sum and carry vectors.

The *Distance Computation and Maximum Generation Unit* computes $| E_z - E_{xy} + 56 |$ or $| E_u - max(E_{ab}, E_{cd} + 28) |$. Since the biases are subtracted during the computation of E_{xy} and $max(E_{ab}, E_{cd})$, the constants used to calculate the *sa* are 56 and 28. The selected difference, *sa*, is the shift-amount sent to the *Right-Shifter Unit* when the multiplier operates in *DPM* or *DOP* mode. This unit also computes $max(E_z, E_{xy})$ or $max(E_u, E_{ab}, E_{cd})$ based on the multiplication mode.

The *Right-Shifter Unit* can perform up to 161 digit right-shift. This unit right shifts either $('1'\&M_z)$ by $(sa+55)$ digits in *DPM* mode or $('1'\&M_u)$ by $(sa+85)$ digits in *DOP* mode.

The functions of the units located in the third stage are explained as follows:

The aligned mantissa (M_z or M_u) is split into two parts, low and high, the low part consists of least-significant 106 bits and the high part consists of most-significant 55 bits. The low part is added with sum and carry vectors in the 106-bit *CSA* adder and the high-part is incremented by the *INC* unit. The incremented value of the high-part is selected, if the 106-bit *CPA* generates a carry-out.

The *CPA* generates a sum or two sums based on the multiplication mode. In *DPM* mode, a 106-bit sum is generated; in *SPM* mode two 48-bit sums are generated; in *DOP* mode a 50-bit sum is generated.

The last stage performs the normalization, exponent update, and rounding as follows:

The *Complement Unit* generates the complement of a negative result and updates the sign of the result (S_r) in *DPM* and *DOP* modes. The *LZA* computes the shift-amount required to normalize the sum generated by the *CPA*. The *LZA* unit is designed by using the method presented by (Schmookler and Mikan, 1996). Note that this unit determines the shift-amount exactly because there is no carry input to the *CPA*. The *Sticky1 Unit* is designed by adapting the method presented in (Yu and Zyner, 1995). This unit computes preliminary sticky-bit using the carry and save vectors.

The *Normalize 1* and *Normalize 2* units generate the normalized products in *SPM* mode. These units can perform a 1-digit right-shift. The *Normalize 3* unit performs the normalization for *DPM* and *DOP* modes. This unit is capable of performing up to 108 digit left-shift. The *Sticky2 Unit* generates the sticky-bits based on the preliminary sticky-bits and shifted-out bits. The *Exp Upd 1* and *Exp Upd 2* increments their inputs by one if a normalization right-shift is performed. *Exp Upd 3* decrement the exponent down to 53, this unit is only used in *DPM* and *DOP* modes.The signals S_r, E_r, and M_r represent the sign, exponent and mantissa of the result in *DPM* and *DOP* modes, respectively.

3.4 Multi-Functional Quadruple-Precision *FPMAF*

This section presents a multi-functional quadruple-precision *FPMAF* designed by extending the techniques presented in the previous sections. The Quadruple-Precision *FPMAF* design execute parallel double-precision and single-precision multiplications, and dot product operations. (Gok and Ozbilen, 2008) Also, the number of single-precision operands that can be operated on is increased from two to four. Brief descriptions for the supported modes of operations are given as follows:

1. In *QPM* mode, the design works as a quadruple-precision *FPMAF* unit. It computes $X \cdot Y + Z$, where X, Y and Z are quadruple-precision floating-point numbers.

2. In *DPM* mode, the design works as a double-precision floating-point multiplier and computes $K \cdot L$ and $R \cdot T$, where K, L, R, and T are double-precision floating-point numbers.

3. In *SPM* mode, the design works as a single-precision floating-point multiplier and computes $A \cdot B$, $C \cdot D$, $E \cdot F$, and $G \cdot H$ in parallel, where all operands are single-precision floating-point numbers.

4. In *DDOP* mode, the design works as a double-precision dot-product unit, and performs two double-precision floating-point multiplications in parallel and then adds the products of these multiplications with a double-precision operand, UD. This operation can be expressed as

$$K \cdot L + R \cdot T + U \tag{3.18}$$

5. In *SDOP* mode, the design works as a single-precision dot-product unit, and performs four double-precision floating-point multiplications in parallel and then adds the products of these multiplications with a single-precision operand, *NS*. This operation can be expressed as

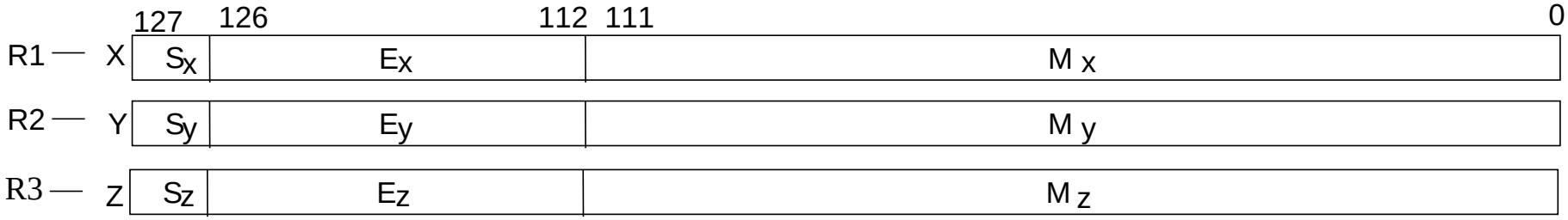
$$A \cdot B + C \cdot D + E \cdot F + G \cdot H + N \tag{3.19}$$

3.4.1 The Preparation of Mantissas

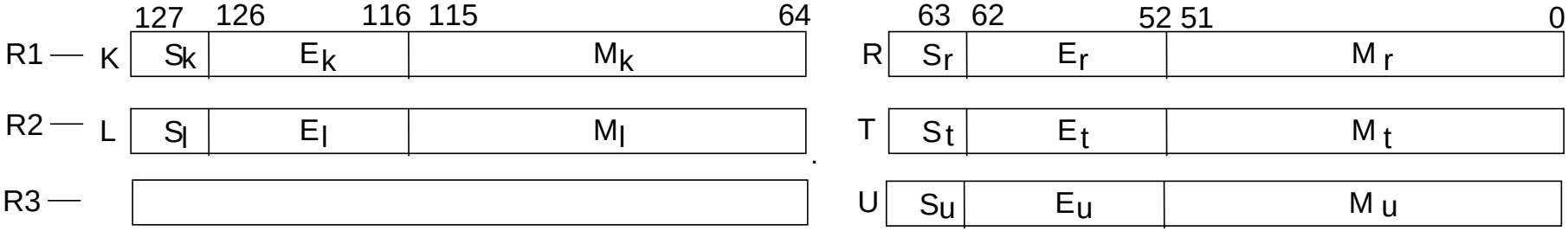
Figure 3.11 shows the alignments of the three quadruple-precision, five double-precision, and nine single-precision floating-point operands in 128-bit registers, $R1$, $R2$, and $R3$ register. The proposed design method modifies the operands based on the execution mode.

Table 3.2 shows the logic Equations used to generate modified mantissas for all modes in quadruple-precision *FPMAF*. Without loss of generality, the Equations in this table are derived based on the following assumptions for the exponents:

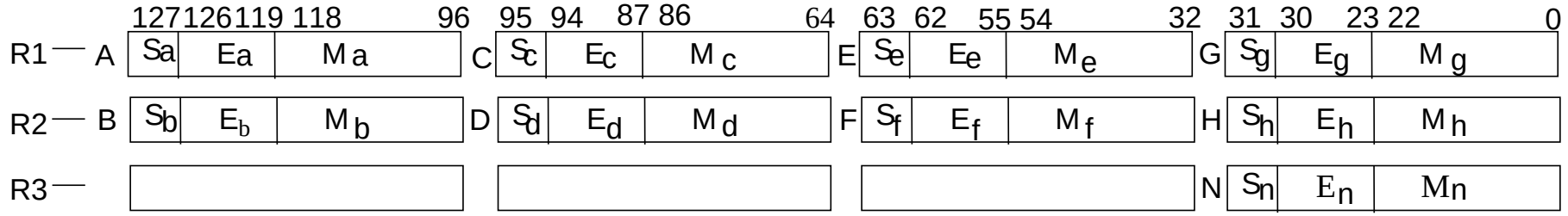
$$E_{rt} \leq E_{kl}, E_{ab} \leq E_{cd},$$
$$E_{ef} \leq E_{gh}, E_{cd} \leq E_{gh}$$



(a) Quadruple−Precision Alignment



(b) Double−Precision Alignment



(c) Single−Precision Alignment

Figure 3.11: The Alignments of Quadruple, Double and Single Precision Floating Point Operands in 128-bit Registers.

Table 3.2 The Logic Equations for The Generation of The Modified Mantissas for All Modes.

QPM	$M1 = 1$ & $R1_{111:0}$	$M2 = 1$ & $R2_{111:0}$
DPM	$M1_{UH} = \{0\}^{60}$ & 1 & $R1_{51:0}$	$M2 = 0001$ & $R2_{51:0}$ & 1 & $R2_{115:64}$ & 000
	$M1_{LH} = 1$ & $R1_{115:64}$ & 0^{60}	
DDOP	$M1_{UH} = \{R1_{63} \oplus R2_{63}\}^{d4}$ & 1 & $R1_{51:0}$ & $\{0\}^{60-d4}$	$M2 = 0001$ & $R2_{51:0}$ & 1 & $R2_{115:64}$ & 000
	$M1_{LH} = \{0\}^{60}$ & 1 & $R1_{115:64}$	
SPM	$M1_1 = \{0\}^{89}$ & 1 & $R1_{22:0}$	$M2 = 001$ & $R2_{22:0}$ & $\{0\}^{7}$ & 1 & $R2_{54:32}$ & $\{0\}^{6}$ & 1 & $R2_{86:64}$ & 1 & $R2_{118:96}$ & 00
	$M1_2 = \{0\}^{60}$ & 1 & $R1_{54:32}$ & $\{0\}^{29}$	
	$M1_3 = \{0\}^{29}$ & 1 & $R1_{86:64}$ & $\{0\}^{60}$	
	$M1_4 = 1$ & $R1_{118:96}$ & $\{0\}^{89}$	
SDOP	$M1_1 = \{R1_{31} \oplus R2_{31}\}^{d1+d3}$ & 1 & $R1_{22:0}$ & $\{0\}^{89-(d1+d3)}$	$M2 = 001$ & $R2_{22:0}$ & $\{0\}^{7}$ & 1 & $R2_{54:32}$ & $\{0\}^{6}$ & 1 & $R2_{86:64}$ & 1 & $R2_{118:96}$ & 00
	$M1_2 = \{0\}^{29}$ & $\{R1_{63} \oplus R2_{63}\}^{d3}$ & 1 & $R1_{54:32}$ & $\{0\}^{60-d3}$	
	$M1_3 = \{0\}^{60}$ & $\{R1_{95} \oplus R2_{95}\}^{d2}$ &1 & $R1_{86:64}$ & $\{0\}^{29-d2}$	
	$M1_4 = \{0\}^{89}$ & 1 & $R1_{118:96}$	

The modifications of the mantissas in *QPM*, *DPM* and *DDOP* modes in the quadruple-precision *FPMAF* are similar to the the modifications of the mantissas in *DPM*, *SPM*, and *DOP* modes in the proposed double-precision *FPMAF*. In *QPM* mode, one version of $M1$ and $M2$ are produced. In *DPM* and *DDOP* modes, two versions of $M1$ and one version of $M2$ are produced. Two versions of $M1$ ($M1_{UH}$ and $M1_{LH}$) are used for the generation of upper and lower half of the 113 by 113 matrix similar to the previous implementation. In *SPM* and *SDOP* modes four version of $M1$ and one version of $M2$ are generated. In these modes, 113 by 113 matrix is divided into four regions. These regions are generated by the multiplications; $M1_1 \cdot M2$, $M1_2 \cdot M2$, $M1_3 \cdot M2$, and $M1_4 \cdot M2$. The implementations for *SPM* and *SDOP* modes will be explained in detail, since they are slightly different than the implementations described before.

Figure 3.12 shows the 113 by 113 multiplication matrix generated for *SPM* mode in the quadruple-precision implementation. In this figure, the shaded regions,that are labeled with 'Z', are set to zeros and the four unshaded regions contain 24 by 24 sub-matrices generated for the following multiplications:

$$(1 \,\&\, M_a) \cdot (1 \,\&\, M_b), (1 \,\&\, M_c) \cdot (1 \,\&\, M_d) \tag{3.20}$$

$$(1 \,\&\, M_e) \cdot (1 \,\&\, M_f), (1 \,\&\, M_g) \cdot (1 \,\&\, M_h) \tag{3.21}$$

Figure 3.13 presents the 113 by 113 matrix multiplication matrix generated for *SDOP* mode. In this figure, four 25 by 25 matrices are placed into the 113 by 113 matrix based on the assumptions for the exponents given above. In *SDOP* mode, the matrices are aligned according to the difference between their exponents. To do that, four 25 by 25 matrices are grouped in two pairs. One of the pairs consists of the matrices generated by the multiplications:

$$(1 \,\&\, M_a) \cdot (1 \,\&\, M_b) \text{ and } (1 \,\&\, M_c) \cdot (1 \,\&\, M_c) \tag{3.22}$$

and the other pair consists of the matrices generated by the multiplications:

$$(1 \,\&\, M_e) \cdot (1 \,\&\, M_f) \text{ and } (1 \,\&\, M_g) \cdot (1 \,\&\, M_h) \tag{3.23}$$

The distances used for the alignment of matrices are computed as follows:

$$d1 = \begin{cases} \mid E_{ab} - E_{cd} \mid & \text{, if } max(E_{ab}, E_{cd}) \leq max(E_{ef}, E_{gh}) \\ \mid E_{gh} - E_{ef} \mid & \text{, otherwise} \end{cases} \tag{3.24}$$

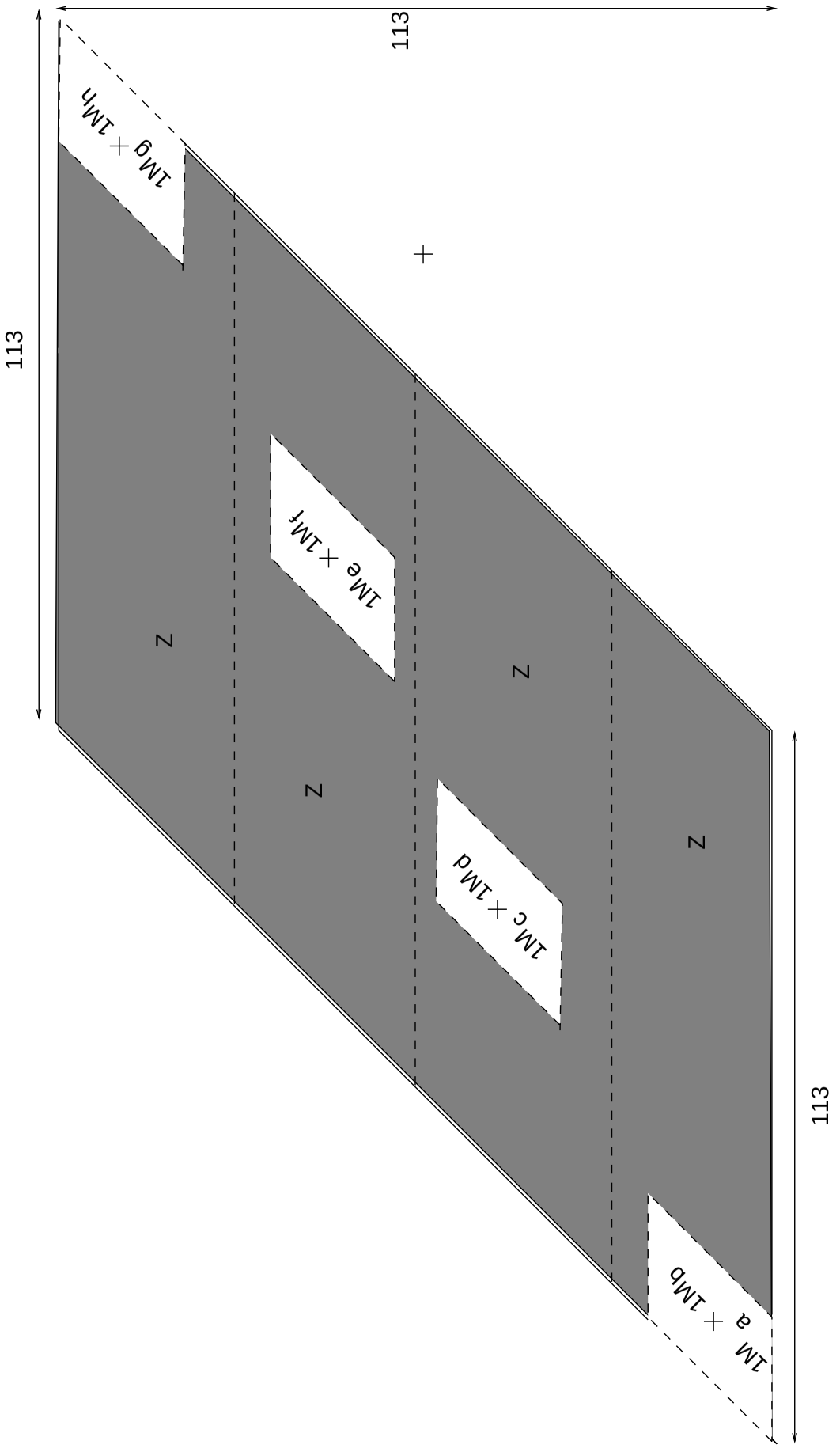


Figure 3.12: The Partial Product Matrices Generated for *SPM* Mode in the Quadruple Precision *FPMAF*.

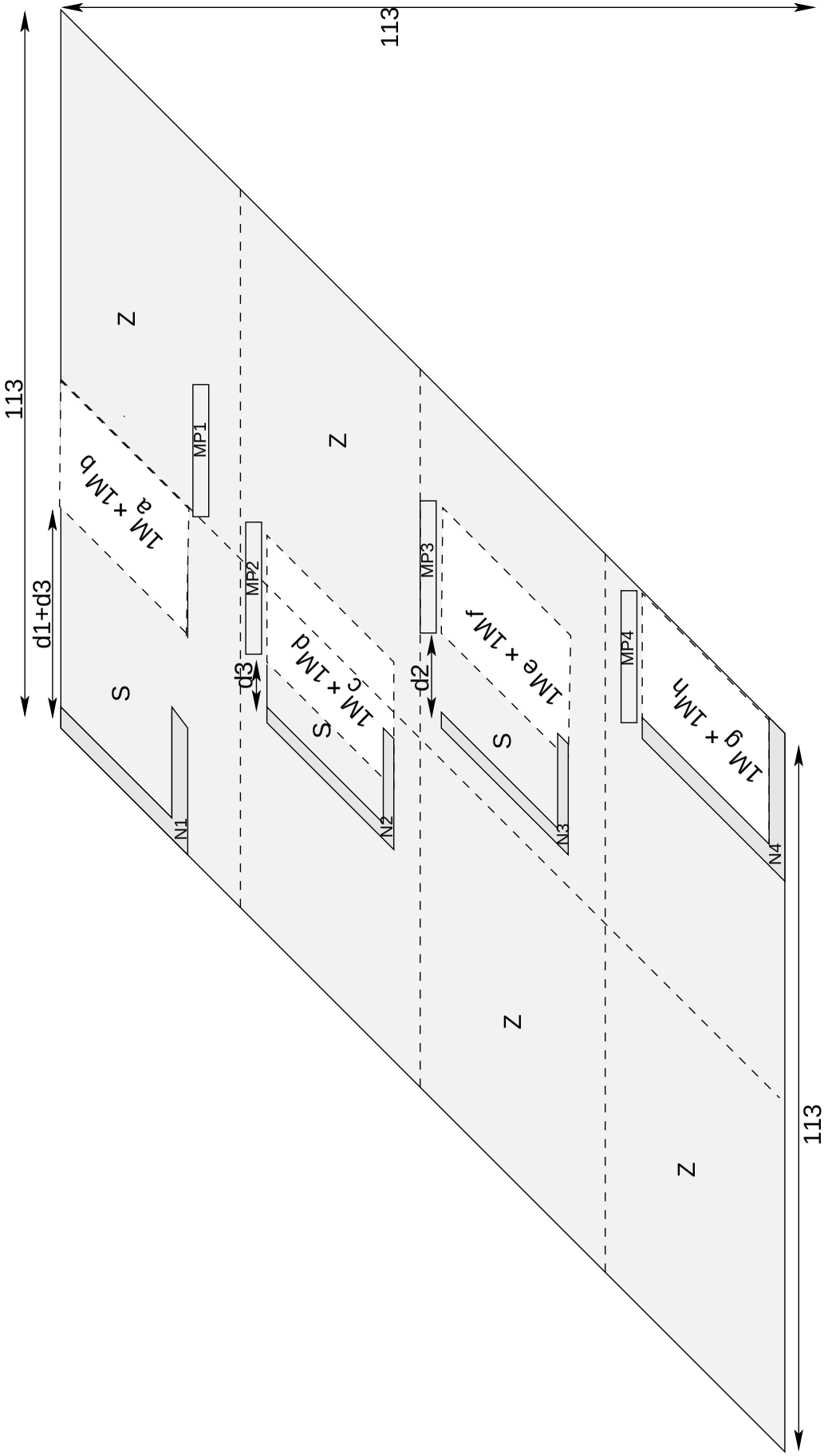


Figure 3.13: The Matrix Generated for Single Precision Dot Product (*SDOP*) Mode in the Quadruple Precision *FPMAF*.

$$d2 = \begin{cases} \mid E_{ef} - E_{gh} \mid & \text{, if } max(E_{ab}, E_{cd}) \leq max(E_{ef}, E_{gh}) \\ \mid E_{ab} - E_{cd} \mid & \text{, otherwise} \end{cases} \tag{3.25}$$

$$d3 = \mid max(E_{ab}, E_{cd}) - max(E_{ef}, E_{gh}) \mid \tag{3.26}$$

The pair that contains the matrix with the maximum exponent is placed into the lower half of the 113 by 113 matrix, in which the matrix with the maximum exponent is located at the bottom and the other one is placed above it and right shifted by $d2$ columns. The other pair is moved into the upper half of the 113 by 113 matrix, in which the matrix that has the minimum exponent is located at the top and right shifted by $(d1 + d3)$ columns. The second matrix in this pair is located under the top matrix and right shifted by $d3$ digits. Similar to double-precision implementation, the additional adjustments such as conversion of the operands in two's complement format when the signs are different, and the application of two's complement word correction algorithm are also used in this implementation. The vectors $MP1$ to $MP4$ represents the positive multiplicands inserted into the multiplication matrix.

3.4.2 The Implementation Details for The Multi-Functional Quadruple-Precision *FPMAF* Design

The block diagram for the proposed quadruple-precision *FPMAF* design is shown in Figure 3.14. This design is quite similar to the proposed double-precision *FPMAF* design, except the sizes of the components are increased and some of the units are modified to be used in different precisions. The design is divided into four pipeline stages. The function of each block and the data flow between stages are explained as follows:

The first stage is mainly dedicated to the preparation of mantissa vectors. The control signals $T_{2:0}$ are used to select the operation mode given in Table 3.3

The function of the each unit in this stage is explained as follows: The *Sign Generator Unit* consists of XOR gates that compares the signs of the operands for all modes. This

Table 3.3 Quadruple Precision Execution Modes

T_1T_0	Operation
00	DPM
10	SPM
01	DOP
11	QPM

unit generates the following signals

$$S_{kl} = S_k \oplus S_l, S_{rt} = S_r \oplus S_t \tag{3.27}$$

$$S_{ab} = S_a \oplus S_b, S_{cd} = S_c \oplus S_d \tag{3.28}$$

$$S_{ef} = S_e \oplus S_f, S_{gh} = S_g \oplus S_h \tag{3.29}$$

$$S_1 = S_x \oplus S_y \oplus S_z \tag{3.30}$$

There is no need to compare the signs of the operands in *SDOP* and *DDOP* modes because the operands are in two's complement format in those modes. In *QPM* mode, *2's Comp. & Negate* Unit computes the negative of its input, when S_1 signal is set to one, and in the other modes, it generates the two's complement representation of the addend based on its sign.

The *Exponent Adder Unit* consists of two 17-bit adders. For space reasons, in Figure 3.14, 15-bit, 11-bit, and 8-bit exponents are grouped and represented as E_Q, E_D, and E_S, respectively. The 17-bit adders operates on three different size exponents as follows: In *QPM* mode, one 17-bit adder computes

$$E_{xy} = E_x + E_y - 16383 \tag{3.31}$$

In *DPM* mode, two 17-bit adders in parallel compute

$$E_{kl} = E_k + E_l - 1023 \tag{3.32}$$

$$E_{rt} = E_r + E_t - 1023 \tag{3.33}$$

In *SPM* mode, one 17-bit adder computes

$$E_{ab} = E_a + E_b - 127 \tag{3.34}$$

$$E_{cd} = E_c + E_d - 127 \tag{3.35}$$

and the other one computes

$$E_{ef} = E_e + E_f - 127 \tag{3.36}$$

$$E_{gh} = E_g + E_h - 127 \tag{3.37}$$

The *Difference and Maximum Generator Unit* consists of one 11-bit subtracter, three 8-bit subtracters and several multiplexers. In *DPM* mode, this unit computes

$$d4 = \mid E_{rt} - E_{kl} \mid \text{ and } max(E_{rt}, E_{kl}) \tag{3.38}$$

In *SDOP* mode, the unit computes $d1$, $d2$, and $d3$. These values and the signs of the differences before the absolute value conversions ($sd1$, $sd2$, $sd3$) are sent to the *Mantissa Modifier Unit1* The *Mantissa Modifier Unit* is split into two parts to balance the delay between Stage 1 and Stage 2. The *Mantissa Modifier Unit1* and *Mantissa Modifier Unit2* generate the modified mantissas using Equations presented in Table 3.2 for all modes. The *Mantissa Modifier Unit1* unit consists of multiplexers and *Mantissa Modifier Unit2*(located in Stage 2) consists of three 113-bit right shifters that can shift up to 89, 60 and 29 digits, respectively.The functions of the units in the second stage are explained as follows:

The modified mantissas are multiplied by the *Mantissa Multiplier*. The generation of partial products in the multiplier is slightly modified to implement the insertion of $MP1$ to $MP4$ in *SDOP* mode or $MP1$ and $MP2$ in *DDOP* mode ($MP1$ and $MP2$ are generated differently in *SDOP* and *DDOP* modes) and to perform the inversion of the bits in regions, $N1$, $N2$, $N3$, and $N4$ ($N1$ and $N2$ regions are different in *SDOP* and *DDOP* modes). The rest of the hardware that handles the partial product reduction is not modified. The *Mantissa Multiplier* generates sum and carry vectors.

The *Distance and Maximum Generation Unit* computes

$$E_z - E_{xy} + 116 \tag{3.39}$$

$$\text{or } E_u - max(E_{kl}, E_{rt}) + 57 \tag{3.40}$$

$$\text{or } E_n - max(E_{ab}, E_{cd}, E_{ef}, E_{gh}) + 28 \tag{3.41}$$

This difference, $'sa'$, is sent to the *Right-Shifter Unit* when the multiplier operates in *QPM*,

DDOP, and *SDOP* modes. Based on the multiplication mode, this unit also generates

$$max(E_z, E_{xy}) \tag{3.42}$$

$$\text{or } max(E_u, E_{kl}, E_{rt}) \tag{3.43}$$

$$\text{or } max(E_n, E_{ab}, E_{cd}, E_{ef}, E_{gh}) \tag{3.44}$$

The *Right-Shifter Unit* can perform up to 200 digit right shift. This unit right shifts $(1\ \&\ M_z)$ by $(sa+116)$ digits in *QPM* mode, or $(1\ \&\ M_u)$ by $(sa+172)$ digits in *DDOP* mode, or $(1\ \&\ M_n)$ by $(sa+200)$ digits in *SDOP* mode. The functions of the units in the third stage are explained as follows:

The *CSA* adds the sum and carry output of the *Mantissa Multiplier* and the aligned mantissa (M_z or M_u or M_n) and generates carry and save vectors. The high part of the aligned mantissa is sent to the *INC* unit, the low part of the aligned addend is sent to the 226-bit *CPA*. The incremented high-part is selected, if the carry-out bit of the *CPA* is 1.

The 226-bit *CPA* generates different sums based on the multiplication mode. In *QPM* mode, a 226-bit sum is generated; in *DPM* mode two 106-bit sums are generated; in *SPM* mode four 48-bit sums are generated; in *DDOP* mode a 108-bit sum is generated; in *SDOP* mode a 51-bit sum is generated.

The *Sticky1 Unit* is designed by adapting the method presented in (Yu and Zyner, 1995). This unit computes preliminary sticky-bit/s for all modes. The *LZA* computes the shift-amount required to normalize the sum generated by the *CPA*. This unit is designed based on the method presented in (Schmookler and Mikan, 1996).

The last stage performs the normalization, exponent update, and rounding as follows:

The *Complement Unit* generates the complement of a negative result and updates the sign of the result (S_r). This unit is used in *QPM*, *DDOP*, and *SDOP* modes. The *Normalize 1* unit generates the normalized products in *DPM* and *SPM* modes. This unit consists of two 53-bits right shifters. 53-bit right shifters are modified to operate on 24-bit operands too. The *Normalize 2* unit performs the normalization for *QPM*, *DDOP*, and *SDOP* modes. This unit is capable of performing up to 239 digit left-shift. The *Sticky2 Unit* generates the sticky-bit/s based on the preliminary sticky-bit and shifted-out bits. All rounding units can increment the normalized products by 1 *ulp* based on the rounding mode. The *Exp Upd 1* unit consists of two 17-bit incrementers. This unit increments four 8-bit operands in *SPM* mode or two 11-bit operands in *DPM* mode. The *Exp Upd 2* in-

crements the 15-bit operand up to 113, this unit is only used in *QPM*, *DDOP*, and *SDOP* modes.

S_r, E_r, and M_r represent the sign, exponent and mantissa of the result in *QPM*, *DDOP* and *SDOP* modes, respectively.

3.5 Multi-Precision Floating-Point Reciprocal Unit

3.5.1 Derivation of Initial Values

Let n-bit mantissa M is represented as

$$1.m_1m_2m_3\cdots m_{n-1}\,(m_i \in \{0,1\}, i = 1\cdots n) \tag{3.45}$$

when M is divided in to two parts M_1 and M_2 as

$$M_1 = 1.m_1m_2m_3\cdots m_m \text{ and } M_2 = 0.m_{m+1}m_{m+2}m_{m+3}\cdots m_{n-1} \tag{3.46}$$

The first-order Taylor expansion of M^p of number, M is between M_1 and $M_1 + 2^{-m}$ and is expressed as (Takagi, 1997)

$$\left(M_1 - 2^{-m-1}\right)^{p-1} \times \left(M_1 + 2^{-m-1} + p \cdot \left(M_2 - 2^{-m-1}\right)\right) \tag{3.47}$$

The Equation 3.47 can be expressed as

$$C \times M' \tag{3.48}$$

where

$$C = \left(M_1 - 2^{-m-1}\right)^{p-1} \tag{3.49}$$

$$\text{and } M' = M_1 + 2^{-m-1} + p \cdot \left(M_2 - 2^{-m-1}\right)$$

C can be read from a lookup-table which is addressed by M_1, without leading one. The look-up table contains the 2^m of C values of M for special values of p, where it is -2^0 for reciprocal of M. The size of the required ROM for the look-up table is about $2^m \times 2 \cdot m$ bits.The initial approximation of floating point number M^{-1} is computed by multiplication of term C with modified operand M. The modified form of M by only complementing M_2 part bitwise. The last term can be ignorable.

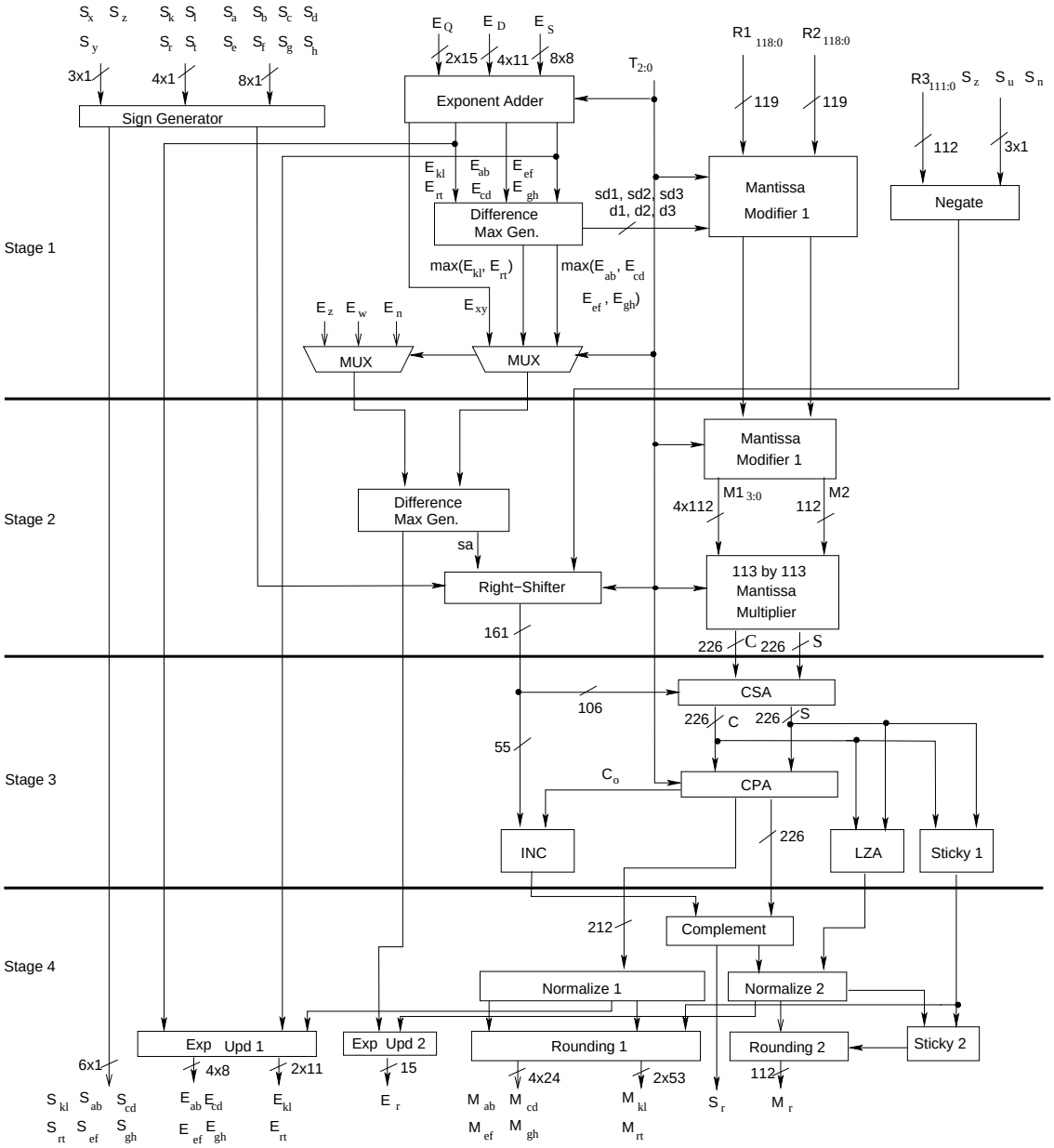


Figure 3.14 The Block Diagram for the Proposed Quadruple Precision *FPMAF* Design.

3.5.2 Newton-Raphson Iteration

The Newton-Raphson iteration was discussed in Previous Work. The iteration general formula is rewritten here (Ercegovac, 2004):

$$x_{i+1} = x_i - f(x)/f'(x_i) \tag{3.50}$$

An initial look-up table is used to obtain an approximate value of the root. The derivation of algorithm using Newton-Raphson method for computing reciprocal as follows:

$$x = 1/X \tag{3.51}$$

$$f(x) = 1/X - x \tag{3.52}$$

$$f'(x) = 1/x^2 \tag{3.53}$$

When Equations 3.53 are put into Equation 3.50, the iteration Equation yields to

$$x_{i+1} = x_i - (2 - X \centerdot x_i) \tag{3.54}$$

The Equation 3.54 can be implemented in hardware. The implementation requires two multiplications and one subtraction operation. The block diagram of this implementation can be seen in Figure 3.15. The circuit can be implemented as pipelined. Basic multiplicative reciprocal unit is show in Figure 3.15. The mantissa modify unit process the most significant part of the M and generates M according to Equation 3.50. Also, the initial approximation, C is obtained from the look-up table.

In the first cycle, the first multiplexer selects modified M value, the second multiplexer selects the output of the first multiplexer. The third multiplexer selects the output of the lookup table and the forth selects also the output of third multiplexer. In the second cycle, the multiplier generates a result in carrysave format. In the third cycle the carry-save vectors are summed by a fast carry-propagate adder. At the end of the third cycle the initial value, x_i is obtained. In the fourth cycle, the first and second multiplexers select the initial value generated in the previous cycle, the third and fourth multiplexer select M. In the fifth cycle, these values are multiplied and in the sixth cycle, the vectors generated by the multiplication are added. In the seventh cycle, the twos complement of the result is selected and the stored initial value in first iteration of the Newton-Raphson is selected. In the seventh and eighth cycle, these values are multiplied and vectors are summed for final

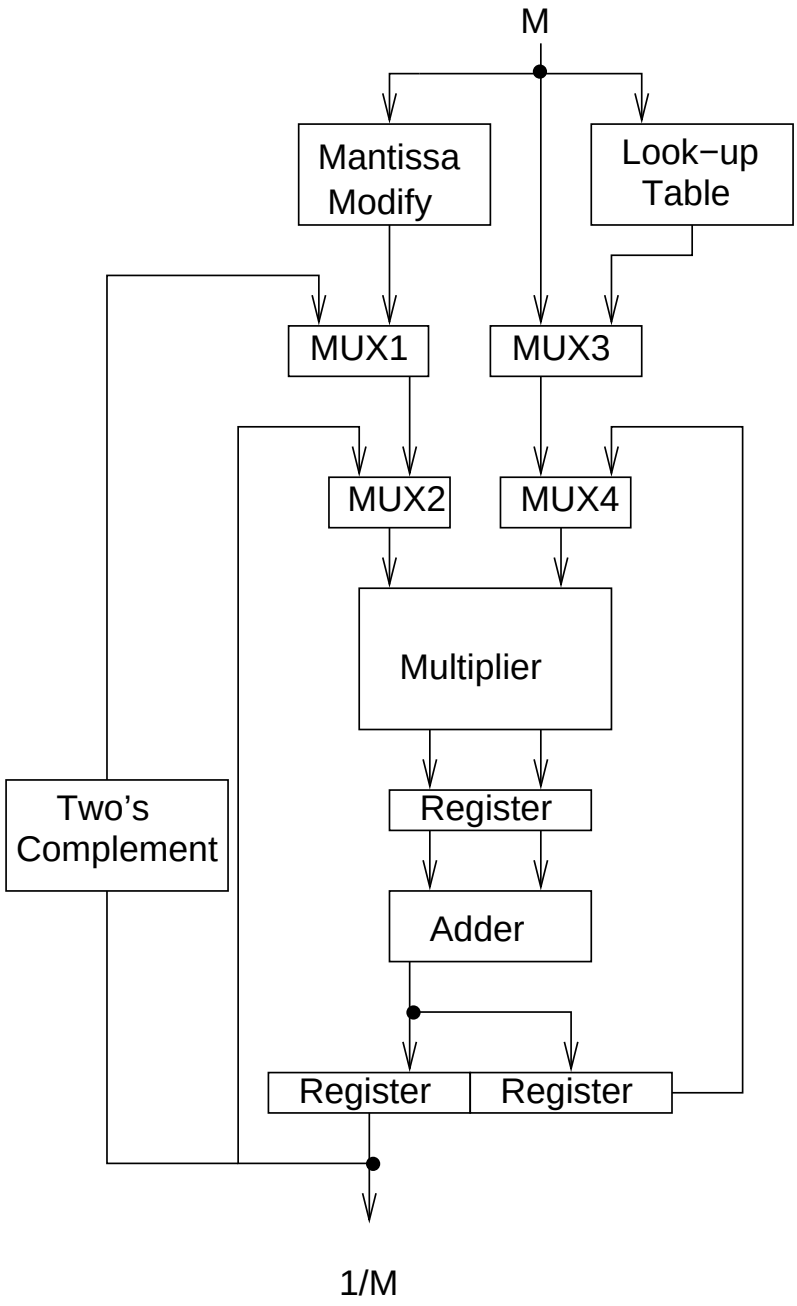
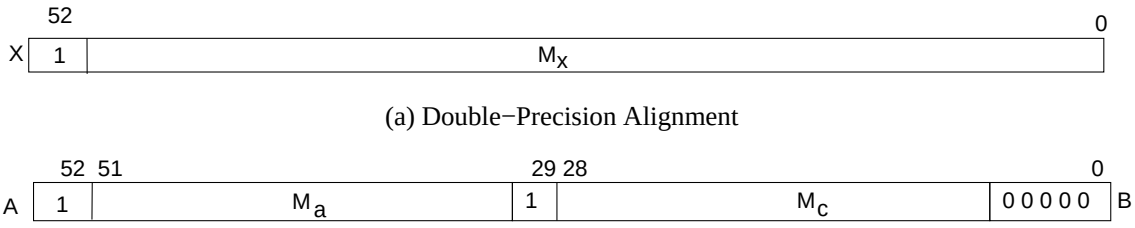


Figure 3.15 Simple Reciprocal Unit that uses Newton-Raphson Method.

result of iteration calculation. In the ninth cycle, the final result routed to normalization to suit IEEE mantissa format.

Rounding is not handled here because this circuit can be coupled with a floating point multiplier for realizing floating point division operation. Rounding can be handled after multiplication by multiplication circuitry. This also minimizes the rounding error.

A packed multiplier design which performs the mantissa multiplications for Newton-Raphson method was discussed in Double/Single Precision Multiplier is rearranged here. Figure 3.16.a shows the alignment of one double precision floating-point mantissa and Figure 3.16.b shows the alignments of two single precision mantissas (Gok, Schulte, and Krithivasan, 2004).



(a) Double−Precision Alignment

(b) Single−Precision Alignment

Figure 3.16 Alignment of Double Precision and Single Precision Mantissas

Figure 3.17 presents the adaptation of the techniques given in (Gok, Schulte, and Krithivasan, 2004) to implement the proposed design. In this figure, the matrices generated for two single precision mantissa multiplications are placed in the matrix generated for a double precision mantissa multiplication. All the bits are generated in double precision multiplication; the shaded areas labeled with Z1, Z2 and Z3 are not generated in single precision multiplication. The un-shaded areas are generated for single precision multiplication. The partial products within the regions Z1, Z2, Z3 are generated using Equations:

$$\hat{b_j} = s \cdot b_j \tag{3.55}$$

$$p_{ij} = a_i \cdot \hat{b_j} \tag{3.56}$$

The rest of the partial products are produced with

$$p_{ij} = a_i \cdot b_j \tag{3.57}$$

The signal *s* is used as control. When $s = 1$ only bits with un-shaded regions are generated. When $s = 0$, all bits are generated. The *i* and *j* are indexes for appropriate partial product in multiplication the matrix (Gok and Ozbilen, 2009).

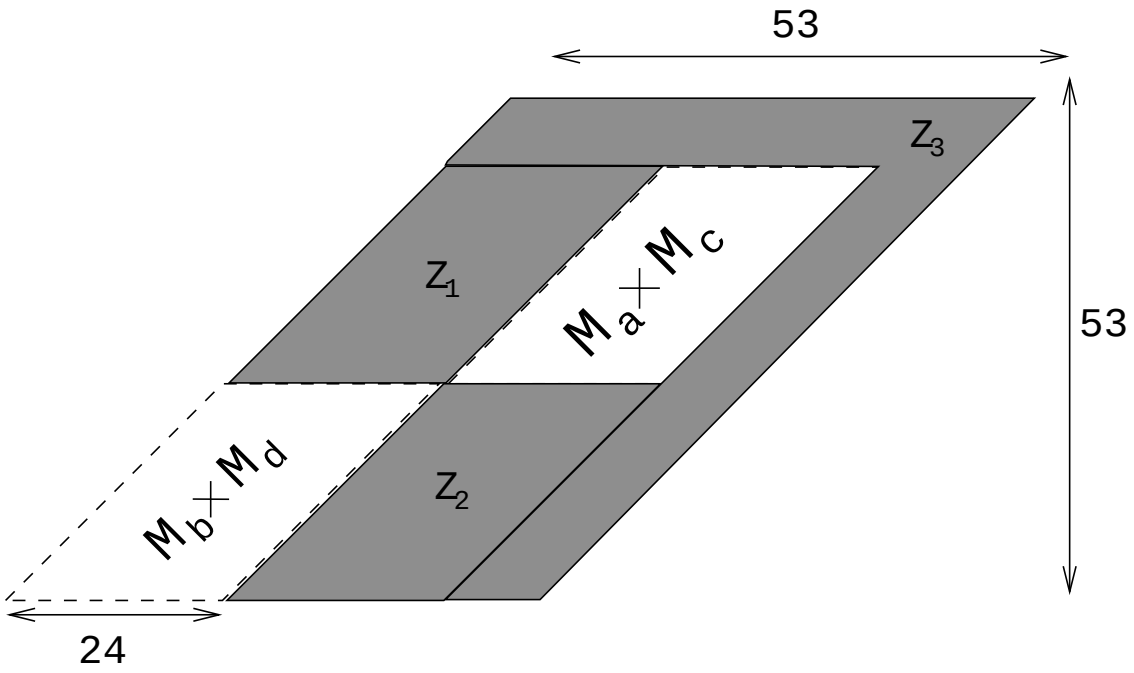


Figure 3.17 Multiplication Matrix for Single and Double Precision Mantissas

3.5.3 The Implementation Details for Double/Single Precision Floating Reciprocal Unit

This unit uses the previous reciprocal computation methods and generates reciprocals in different precisions as follows:

1. In double precision mode the unit generates a double-precision reciprocal.
2. In first single-precision mode, the reciprocal unit generates a single-precision reciprocal and a copy of generated.
3. In the second single-precision mode, the reciprocal unit generates two different reciprocals in parallel.

The input format of modified design is shown in Figure 3.18. Figure 3.18.a shows the input and output format in double precision mode. Figure 3.18.b shows the same

input and output in single precision mode. An input, S signal selects operating mode. The

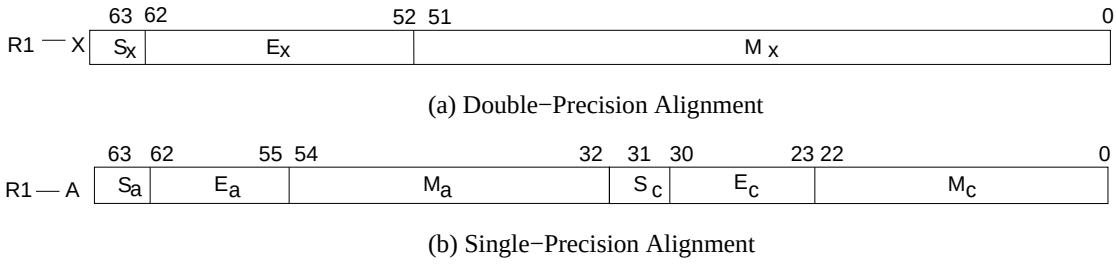


Figure 3.18 Alignment of Double and Single Precision Floating Point Numbers

block diagram for the proposed design is shown in Figure 3.19. The explanations of the main units are as follows:

Exponent Unit generates the exponents of one double precision or two single precision results. In single precision mode exponents are obtained with Equation 3.58. Two circuit compute either exponent in parallel. In double precision mode, the circuits are cascade connected.

$$E_z = \text{"1111111"} - \bar{E_x} \tag{3.58}$$

Mantissa Modifier generates modified mantissas based on the operation mode in order to prepare the inputs ready for the packed multiplier like in Figure 3.16.

Lookup Table contains look-up tables needed for initial approximation required for Newton-Raphson method. These are *C* values of Equation 3.49. They are pre-computed values generated by computer software such Maple, MatLab, etc.

Operand Modifier modifies the operands required for initial value calculation. The value evaluated here is X of Equation 3.48. It is evaluated by inverting the digits starting from 10th digit for this design. The modification of operand(s) depends on the selected operation mode.

State Counter drives the multiplexers to select correct inputs to the packed multiplier during the computation of Newton-Raphson iteration. The computation of Equation 3.54 requires three multiplications. Depending on selected operation mode the inputs of multiplexers are in double precision or packed single precision format as shown in Figure 3.16. In the second cycle of circuit multiplexers are arranged for multiplication of look-up value(s) and modified mantissas as in the Equation 3.54. In the third cycle, multiplexers are arranged for multiplication of computed initial approximation value(s) and the input

mantissa(s) in the Equation 3.54. And, in the fourth cycle, multiplexers are arranged for multiplication of stored initial value(s) and computed value(s) of inside parenthesis of the Equation 3.54.

Packed Multiplier is 53 by 53 multiplier slightly modified to handle two single and one double precision number as described. The input format of multiplier is shown in Figure 3.18. Multiplication output depends on selected operation mode.

Packed Product Generator processes the output of packed multiplier and generates output used in next stages of iteration. The output of this unit is stored in a register. The format of output is truncated one 53-bit double mantissa or two 24-bit single mantissas depending on selected mode. The mantissas arranged as in Figure 3.16.

I.A.Store unit stores the Initial Approximation value(s) computed in the second cycle of circuit. These are xi values in Equation 3.48, which are needed in fourth cycle.

Inverter inverts the stored multiplication result for the third stage of stage controller to compute the expression in the parenthesis of Equation 3.54. The inversion is done depending of selected mode.

Single Normalizer(s) normalize the result in single-precision mode. *Double Normalizer* normalizes the result in double precision mode. The normalization is one left shift if required.

Exponent Updater updates the exponents depending on the normalization results. Two decrementers are separately used to update 8-bit exponents in single mode or in double mode these decrementers are connected cascade to update 11-bit exponent.

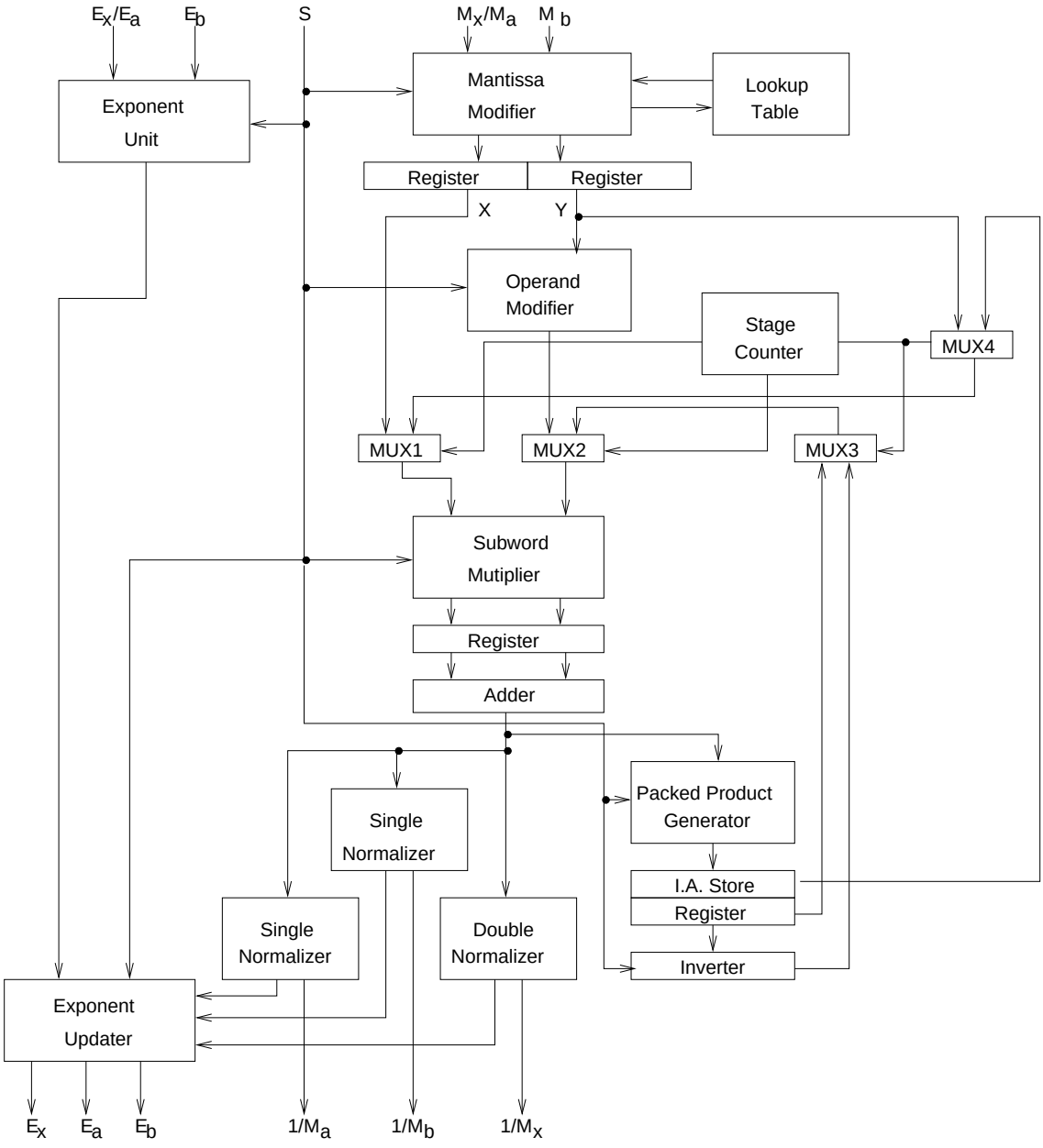


Figure 3.19 The proposed Single/Double Precision Reciprocal Unit

4. RESULTS

This chapter presents synthesis results for the proposed and reference designs, that detailed implementation descriptions given in Chapter 3.

All designs are modeled with VHDL(Very High Speed Integrated Circuit Hardware Description Language). Syntheses are done using TSMC(Taiwan Semiconductor Manufacturing Company) 0.18 micron standard ASIC(Application Specific Integrated Circuit library and Leonardo Spectrum program. The syntheses are tuned for delay optimizations with maximum effort.

4.1 The Results for Multi-Precision Floating-Point Adder Design

This section presents the syntheses results obtained for the proposed multi-precision floating-point adder and single-path floating-point adders. In addition to the double-precision floating-point adders, single-precision floating adders are also designed. The second multi-precision design performs a single-precision floating-point addition or a two half-precision floating-point adders in parallel.

The area and delay estimates are presented in Table 4.1. In this table, the unit for area is the number of gates and the unit for delay is nanoseconds (ns).

Table 4.1 Area and Delay Estimates for Multi-Precision Floating Point Adder

Adder Design	Area(Gates)	Delay(ns)
Double-Precision	4868	14.65
Multi-Precision 1	8195	17.33
Single-Precision	2056	9.33
Multi-Precision 2	2854	9.51

According to the given estimates the first multi-precision design has approximately 68% more area and has less than 3 nanoseconds more delay than the reference double precision design and the second multi-precision design has approximately 38 % times more gates and has less than half nanoseconds more delay than the reference single precision floating-point adder. The delay differences between the proposed designs and the refer-

ence designs are expected to decrease if the designs are pipelined. A question that can be raised is why not use one double-precision, two single-precision and four half-precision floating-point adders instead of the multi-precision one floating-point adder, that capable of handling all mentioned formats. The proposed unit is expected to use approximately 20% less gates than the total gates required to design all separate units (Assuming a half-precision floating-point adder can be designed by using approximately 500 gates). Also, the dedicated bus requirement for all the units can be a serious design problem since the wire delay gets significant as the transistor sizes decreases. The additional components used to provide single/double precision can be seen in Table 4.2.

Table 4.2 Additional Components in Multi-Precision Adder Design

Unit Name	Wide	Number
Adder/Subtractor	8-bit	6
Decoder/Encoder	3-bit	3
Left Shifter	24-bit	1
Left Shifter	10-bit	2

The proposed design eliminates the type conversion requirement and generates multiple results in parallel. The presented design is especially expected to increase the performance for 2D and 3D applications since these applications performs intensive floating-point additions on low-precision floating-point operands.

4.2 The Results for Single/Double Precision Floating-Point Multiplier Design

In this section we present the synthesis results for the proposed single/double precision floating point multiplier and the standard dual precision floating-point multiplier. Both circuits are optimized for delay. The values in Table 4.3 are in nanoseconds for time and in number of gate for area.

The single/double precision multiplier has approximately 9.49% more area and has about 34% more critical delay. The floating-point multipliers used in modern processors are usually pipelined designs. If the proposed method is applied to a pipelined multiplier

Table 4.3 Area and Delay Estimates for Single/Double-Precision Multiplier Design

Adder Design	Area(Gates)	Delay(ns)
Double-Precision	25175	4.10
Multi-Precision	27566	5.49

the area increase is expected to fall down below 5% and also critical delay increase will be dissolved in pipeline stages.

One of the important aspects of the presented design method is that it can be applicable to all kinds of floating-point multipliers. The presented design is compared with a standard floating point multiplier via synthesis. The synthesis results showed that proposed design is 10% larger than conventional multiplier and critical path increment is only one or two gate delay. Since modern floating-point multiplier designs have significantly larger area than the standard floating-point multiplier, the percentage of the extra hardware will be less for those units. The additional components used to provide single/double precision can be seen in Table 4.4. The methods presented in this design is used on the design of floating-point multiplier-adder circuits.

Table 4.4 Additional Components in Single/Double-Precision Multiplier Design

Unit Name	Wide	Number
Adder/Subtractor	8-bit	2
Incrementer	8-bit	2
Left Shifter	24-bit	1

4.3 The Results for Multi-functional Double-precision FPMAF design

The major additional components used to convert the basic double-precision to the Multi-functional double-precision *FPMAF* are placed in the following stages:

The first stage: Two 8-bit adders, one 11-bit adder, and one 8-bit subtracter (in the *Difference and Maximum Generator*). One 53-bit right-shifter that can shift up to 29 digits (in the *Mantissa Modifier*). The fourth stage: Two 8-bit incrementers (*Exp Upd 1*

and *Exp Upd 2*), and two 24-bit incrementers (*Rounding 1* and *Rounding 2*). Two 48-bit 1-digit right-shifters.

The *Right-Shifter* in Stage 2, the *Mantissa Multiplier*, *LZA*, and *Sticky1* in Stage 3 are also slightly modified to handle multiple-precision operands, but the amount of extra hardware for these modifications are negligible. The proposed double precision design can be optimized by combining *Normalize 1* and *Normalize 2*, *Rounding 1* and *Rounding 2*, *Exp Upd 1* and *Exp Upd 2* units. However, the hardware gain by this optimization is not significant.

The proposed multi-functional *FPMAF* design is compared with the standard double-precision *FPMAF* by syntheses. All circuits are modeled using structural VHDL code. The adders, subtracters and incrementers in these designs are implemented by using parallel-prefix adders. The correctness of the proposed designs are verified with extensive simulation. Syntheses are done using TSMC 0.18 micron standard ASIC library and Leonardo Spectrum program. Both syntheses are tuned for delay optimizations with maximum effort. Table 4.5 presents area estimates for conventional and the proposed designs. In this table, the number of gates for each pipeline stage is presented. The proposed double-precision *FPMAF* design have approximately 8% more area than the standard double-precision design.

Table 4.5 Area Estimates for Double-Precision *FPMAF* Design

Pipeline Stage	Basic MAF	Multi-Functional
Mantissa Prepare	-	2805
Multiplication	23771	24184
Add	6450	6570
Round	5428	4950
Total Area	35649	38509

Table 4.6 presents delay estimates for conventional and the proposed design in nanoseconds. The critical delay for the proposed double-precision *FPMAF* design is approximately 2.2% more than the critical delay for the standard double precision design. The delay of the extra pipeline stage is less than the delay for the stage with longest delay.

Table 4.6 Delay Estimates for Double-Precision *FPMAF* Design

Pipeline Stage	Basic MAF	Multi-Functional
Mantissa Prepare	-	3.36
Multiplication	3.42	3.34
Add	3.53	3.61
Round	2.98	2.27

The previous double-precision designs presented in (Jessani and Putrino, 1998), (Huang, Shen, Dai and Wang, 2007) and the proposed double-precision designs are structurally very similar. The dual- precision design in (Huang, Shen, Dai and Wang, 2007) and the proposed design in this study are synthesized using 0.18 TSMC standard library. The extra hardware required to provide multi-precision execution functionality for the proposed designs is less than 9% where as for Huang, Shen, Dai and Wang(2007) design it is 18%. Note that the unit of area estimate for the proposed designs is the number of gates while for Huang *et al.*'s design it is micrometer squares Huang, Shen, Dai and Wang(2007). Even though synthesis tools, mantissa multiplier designs, and adder types are different the estimated clock delays for the proposed and Huang *et al.*'s designs are very close. The delay estimate for Jessani & Putrino's design in (Jessani and Putrino, 1998) could be also very close to those two estimates, if it was synthesized with the same ASIC library. So it can be assumed that the clock delays for all designs are equal. On the other hand, the latencies for the designs in (Jessani and Putrino, 1998), (Huang, Shen, Dai and Wang, 2007), and the proposed design are 3, 3, and 4, respectively.

Table 4.7 Additional Components in Multi-Functional Double-Precision *FPMAF* Design

Unit Name	Wide	Number
Adder/Subtractor	8-bit	3
Incrementer	8-bit	2
Incrementer	24-bit	2
Left Shifter	48-bit	1
Right Shifter	53-bit	1
Right Shifter	108-bit	1

The design is implemented by extending the hardware of conventional *FPMAF* units. The additional components used to provide multifunctionality can be seen in Table 4.7. However, the presented design methods can be tailored to provide same functions in other high-performance *FPMAF* designs. The extra hardware used to modify the standard designs is not significant compared to the overall hardware. In fact, most part of it is fitted into an additional pipeline stage. The proposed designs are expected to increase performance results for the applications that perform lots of independent floating-point multiplications. However, for the applications that may be data dependent, the extra pipeline stage may reduce the performance compared to standard *FPMAF* designs.

4.4 The Results for Multi-Functional Quadruple-Precision *FPMAF*

The additional components used to convert the basic quadruple-precision to the Multi-functional quadruple-precision *FPMAF* are placed in the following stages:

The first stage: Two 17-bit adders, four 8-bit subtracters (in the *Exponent Adder* and *Difference and Maximum Generator*). The second stage: Three 103-bit right shifters (in the *Mantissa Modifier 2*). The fourth stage: Two 17-bit incrementers (in the *Exp Upd 1*) and, and two 53-bit incrementers (in the *Rounding 1*). Two 106-bit 1-digit right-shifters.

The multi-functional *FPMAF* design is compared with the standard quadruple-precision *FPMAF* by syntheses. All circuits are modeled using structural VHDL code. The adders, subtracters and incrementers in these designs are implemented by using parallel-prefix adders. The correctness of the proposed designs are verified with extensive simulation. Table 4.8 presents area estimates for conventional and the proposed designs. In this table, the number of gates for each pipeline stage is presented. The quadruple-precision *FPMAF* design have approximately 12.5% more area than the standard quadruple-precision design. The percentage increase in area is more than the one for double-precision design, since the number of the supported modes is increased in the quadruple-precision design. Table 4.9 presents delay estimates for conventional and the proposed design in nanoseconds. The critical delay for the proposed quadruple-precision *FPMAF* design is approximately 5% more than the critical delay for the standard quadruple-precision design. The delay of the extra pipeline stage is less than the delay for the stage with longest delay.

Table 4.8 Area Estimates for Quadruple-Precision *FPMAF* Design

Pipeline Stage	Basic MAF	Multi-Functional
Mantissa Prepare	-	3494
Multiplication	106224	119684
Add	13518	13940
Round	11663	10720
Total Area	131405	147838

Table 4.9 Delay Estimates for Quadruple-Precision *FPMAF* Design

Pipeline Stage	Basic MAF	Multi-Functional
Mantissa Prepare	-	4.63
Multiplication	4.43	4.71
Add	4.51	4.74
Round	4.26	4.65

The design is implemented by extending the hardware of conventional *FPMAF* units. However, the presented design methods can be tailored to provide same functions in other high-performance *FPMAF* designs. The extra hardware used to modify the standard designs is not significant compared to the overall hardware. The additional components used to provide multifunctionality can be seen in Table 4.10. The single-precision operation modes supported in all the design can be especially useful in 3D multimedia applications which do not require high-precision floating-point operands. The proposed design also support dot product with low-precision operands. The presented dot-product mode reduces the rounding error, since only one rounding is performed in each pass. The proposed design are expected to increase performance results for the applications that perform lots of independent floating-point multiplications. Another advantage of the proposed design over the previous designs is that the proposed design can support more than two precisions where as the previous designs can support only two different precisions. The proposed quadruple-precision multiplier can perform double and single precision operations.

Table 4.10 Additional Components in Multi-Functional Quadrable-Precision *FPMAF* Design

Unit Name	Wide	Number
Adder/Subtractor	17-bit	4
Incrementer	17-bit	2
Incrementer	53-bit	2
Left Shifter	106-bit	1
Right Shifter	113-bit	3
Right Shifter	168-bit	1

4.5 The Multi-Precision Floating-Point Reciprocal Unit

The synthesis results for the proposed single/double precision floating point reciprocal unit is presented. The design in (Kucukkabak and Akkas, 2004) was used as reference standard double precision floating-point reciprocal unit with some estimation. The estimations include design of unsigned radix-2 multiplier, carry propagate-adders and a controlling logic for the multiplexers. The clock delays and area estimates (in terms of number of gates) for both designs are given in Table 4.11. The values in Table 4.11 are in nanoseconds for time and in number of gate for area.

Table 4.11 The Comparison of the Standard Double Precision and Proposed Floating-Point Reciprocal Design

Design	Numb. of Gates	Latency
Reference Double Precision	31979	3.86
Single/Double Precision	33997	3.94

The single/double precision reciprocal unit has approximately 6% more area and has about 3% more critical delay. The most critical delay occurs in the multiplier. Because of the multiplier we used is slightly modified a negligible difference occurs in delay. The additional circuits cause also negligible grows in design. The floating-point reciprocal units used in modern processors are usually pipelined designs. The design performs two single-precision reciprocal with about same latency which is dissolved in pipeline stages.

The presented reciprocal unit is designed for multimedia applications and operates on SIMD type data input. The accuracy of the results are 20 bits for each iteration. Compared to the previous reference designs less than 1% area increase and delay increase is reported based on synthesis results. However the functionality of the reciprocal unit is improved to support three operation modes. The mode that generates two different reciprocals simultaneously is expected to double the performance of single precision division operations. The extra hardware used to modify the standard designs is significant compared to the overall hardware. The additional components used to provide multi-precision can be seen in Table 4.12. The proposed unit can be expanded to support reciprocal-square-root operation with additional circuit and modifications.

Table 4.12 Additional Components in Multi-Precision Reciprocal Design

Unit Name	Wide	Number
Adder/Subtractor	8-bit	1
Incrementer	8-bit	1
Left Shifter	24-bit	1
Right Shifter	168-bit	1

5. CONCLUSIONS

This dissertation presents novel floating-point hardware designs for multimedia applications. The main goal of the dissertation is to add functionality and accelerate the basic arithmetic operations used in multimedia applications. Though multimedia applications require too much computational power, this computation is usually repetitive for multimedia data. The SIMD extensions are developed to operate the same operation on the pieces of a packed data in parallel. SIMD instruction set extensions are very popular among major processor manifacturars. For example, SSE, SSE', SSE3, SSE4 from Intel Corp., 3DNow form AMD are well know examples. instructions sets. The proposed designs presented in this thesis offers efficient implementations of the main SIMD instructions offered in those popular multimedia instruction set extensions. More precisiley the implementation for the following instructions are presented: Packed floating-point add, packed floating-point multiply, packed floating- point multiply-add, dot product, packed reciprocal operations.

The proposed multi-precision adder can be used in addition or subtraction of two single precision or four half precision operands. When a matrix data has to be added or subtracted, the proposed design can decrease the delay for the calculation about 70%. The proposed floating point adder has about 40% more area with nearly same delay with additional precision capabilities.

The proposed multi-functional MAF design can decrease the delay for the matrix multiplication with its dot product function. It decreases the delay for parallel low precision floating-point multiplications. The proposed design has about 2% more area and same delay with basic double or quad precision multiplier with additional functions like dot product and simulteneous multiplication of two or four single precision number.

Similar gains are achieved by the multi-precision reciprocal design. The proposed design has about 6% more area than reference design. It has about 3% more delay but has capability of taking reciprocal of two single precision floating number beside double precision. When this design is coupled with a multi-functioncal MAF design, the design can perform division or divide and sum operation or a divide and subtract operation.

The major general purpose processor manifactures and graphical processing unit manifacturers are adding new futures to their designs to overcome multimedia load because the

demand on digital world increases day by day. Every single new future requires greater computational power. The purposed designs give support with more computation with same delay. This designs can be implemented directly into microprocessor as an extension or implemented as separate co-processor on a daughter board. When implemented as add-on, it can be used by either graphical processing unit or central processing unit. With some modification, they can be fit on an FPGA(Field Programmable Gate Array) and used for extra calculating power for microcontrollers or analog digital processing units.

Although there exists an abundance of multimedia applications, most of the operations required to execute them are uniform. For example some image manipulation operations, some 3D transformation like rotation, sizing, translation operation or some audio manipulation like amplification, equalization or echo addition/cancellation operations require similar type of operations. All those applications may benefit from the designs developed in this dissertation.

BIBLIOGRAPHY

Akkas, A., Schulte, M.J., 2006. Dual-mode floating-point multiplier architectures with parallel operations. Journal of Systems Architecture, 549-562.

AltiVec Technology Programming Environments Manual, Motorola, Online (2006) http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPEM.pdf?WT_TYPE=ReferenceManuals&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation

AMD-3DNow!, Technology manual, Online (2000). http://www.amd.com

AMD, 2007. ATI FireGL Technical Specifications. Online. http://ati.amd.com/products/workstation/techspecs2.html

Ansi/ieee standard 754, 1985. IEEE standard for binary floating-point arithmetic.

Arfken, G., 1985. Mathematical Methods for Physicists, 3rd ed, Academic Press, Orlando, pp.13-18.

Baugh, C.R., Wooley, B.A., 1973. A Two's Complement Parallel Array Multiplication Algorithm, Computers, IEEE Transactions, C-22(12):1045-1047.

Baugh, C.R., Wooley, B.A., 1973. A two's complement parallel array multiplication algorithm, IEEE Transactions on Computers, C-22(12):1045-1047.

Beaumont-Smith, A., Lim, C.C., 2001. Parallel prefix adder design, Computer Arithmetic, Proceedings. 15th IEEE Symposium on, 218-225.

Beuchat, J.L., Tisserand, A., September 2002. Small multiplier-based multiplication and divison operators for Vertex-II devices. in Proceedings of 12th International Conference on Field-Programble Logic and Applications, 513-522.

Booth, A., 1951. A Signed Binary Multiplication Technique, Quarterly J. Mechanics of Applied Math., 4:236-240.

Buford, J.F.K., 1994. Multimedia Systems, Addison-Wesley Pub. Co.

Charles, P., 25 Jul 2007, 3D Programming for Windows, Microsoft Press, 448p

Chen S., Wang D., Zhang T., Hou C., 2006. Design and Implementation of a 64/32-bit Floating-point Division, Reciprocal, Square root, and Inverse Square root Unit. Solid-State and Integrated Circuit Technology, ICSICT'06. 8th International Conference on, Shanghai, 1976-1979.

Chirca, K., Schulte, M., Glossner, J., Horan W., Mamidi, B., Balzola, P., Vassiliadis, S., 2004. A static low-power, high-performance 32-bit carry skip adder. Digital System Design, DSD Euromicro Symposium on, 615-619.

Cole, P., Oct/Nov 2005. OpenGL ES SC - open standard embedded graphics API for safety critical applications. DASC 2005, 2:8.

Dadda, L., 1965. Some Schemes for Parallel Multipliers, Alta Frequenza, 34:349-356

Debes, E., Macy, W.W., Tyler, J.J., Peleg, A.D., Mittal, M., Mennemeier, L.M., Eitan, B., Dulong, C., Kowashi, E., Witt, W., 2008. Method and Apparatus for Performing Multiply-Add-Operations on Packed Data. Intel Corporation, Patent Number 7.395.298 B2.

Diefendorff, K., Dubey, P.K., Hochsprung, R., Scale, H., Mar/Apr 2000. AltiVec extension to PowerPC accelerates media processing. Micro, IEEE, 20(2):85-95.

Ercegovac, M.D., Lang, T., 2004. Digital Arithmetic, Morgan Kauffmann.

Ercegovac M.D., Lang, T., 1987. On-the-fly conversion of redundant into conventional representations. IEEE Transactions on Computers, 895-897.

Even, G., Mueller, S., Seidel, P., 1997. A dual mode ieee multiplier. Proceedings of the 2nd Annual IEEE Int. Conf. on Innovative Systems in Silicon, Austin, TX, USA, 282-289.

Even, G., Seidel, P.M., 2000. A comparison of three rounding algorithms for ieee floating-point multiplication. IEEE Transactions Computers, 49:638650.

Fossum, T., Grundmann, R.W., Hag, M.S., 1991. Pipelined Floating Point Adder For Digital Computer. Digital Equipment Corporation, Patent Number 4.994.996.

Fu-Chiung, C., Unger, S.H., Theobald, M., Jul 2000. Self-timed carry-lookahead adders, Computers, IEEE Transactions on, 49(7):659-672.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Yao Z., Volkov, V., Jul/Aug 2008. Parallel Computing Experiences with CUDA. Micro, IEEE, 28(4):13-27.

Gok, M., Ozbilen, M.M., 2008. Multi-functional floating-point MAF designs with dot product support Journal of Microelectonics, 39:30-43

Gok, M., Ozbilen, M.M., 2009a. Evaluation of Sticky-Bit Generation Methods for Floating-Point Multipliers. Journal of Signal Processing Systems, 56:51

Gok, M., 2007. A novel IEEE rounding algorithm for high-speed floating-point multipliers. Integration, the VLSI Journal, 40:549-560.

Gok, M., Schulte, M.J., Krithivasan, S., 2004. Designs for subword-parallel multiplications and dot product operations. in: WASP'04, Third Workshop On Application Specific Processors, Stockholm, Sweden, 27-31.

Gok, M., Ozbilen, M. M., 2009b. A Single or Double Precision Floating-Point Multiplier Design for Multimedia Applications. Istanbul University Journal Of Electrical and Electronics Engineering, 9:827-831

Gok, M., Ozbilen, M. M., 2009c. A Single or Double Precision Floating-Point Reciprocal Unit for Multimedia Applications. In Review

Gurkayna, F.K., Leblebicit, Y., Chaouati, L., McGuinness, P.J., 2000 Higher radix Kogge-Stone parallel prefix adder architectures, Circuits and Systems Proceedings. ISCAS 2000 Geneva, 5:609-612.

Harris, D., Sutherland, I., 9-12 Nov 2003. Logical effort of carry propagate adders. Conference Record of the 37th Asilomar Conference on, 1:873-878.

Heikes, C., Colon-Bonet, G., Feb 1996. A Dual Floating Point Coprocessor with an FMAC Architecture. ISSCC Dig. Tech. Papers, 354-355

Hillman, D., 1997, Multimedia Technology and Applications, Delmar Pub., 274p

Hokenek, E., Montoye, R., Cook, P., 1990. Second-generation risc floating point with multiply-add fused. IEEE Journal of Solid-State Circuits, 25(10):1207-1203.

Huang, L., Shen, L., Dai, K., Wang, Z., 2007. A new architecture for multiple-precision floating-point multiply-add fused unit design. Proceedings of the 18th IEEE Symposium on Computer Arithmetic, IEEE Computer Society, Washington, DC, USA, 69-76.

Intel 64 and IA-32 architectures software developer's manual, Online (2007). http://www.intel.com/design/processor/manuals/253667.pdf

Intel SSE4 programming reference, Online (2007). http://softwarecommunity.intel.com

Jagodik, P.J., Brooks, J.S., Olson, C., 2008. Multiplier Structure Supporting Different Precision Multiplication Operations. Sun Microsystems Inc., Patent Number 7.433.912 B1

Jessani, R.M., Putrino, M., 1998. Comparison of single and dual pass multiply add fused floating-point units. IEEE Trans. Comput., 47(9):927-937.

Koren, I., 2002, Computer Arithmetic Algorithms. A.K. Peters Ltd., Canada, 281p

Kucukkabak, U., Akkas, A., 2004. Design and implementation of reciprocal unit using table look-up and Newton-Raphson iteration. Digital System Design 2004 Euromicro Symposium on, 249-253

Lee, C., Potkonjak, M., Mangione-Smith, W.H., 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, IEEE Computer Society, 330-335

Lempel, O., Peleg, A., Weiser, U., 23-26 Feb 1997. Intel's MMXTM technology-a new instruction set extension. Compcon '97. Proceedings, IEEE, 255-259.

Lindholm, E., Nickolls, J., Oberman, S., Montrym, J., Mar/Apr 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. Micro, IEEE, 28(2):39-55.

Macedonia, M., Oct 2003. The GPU enters computing's mainstream. Computer, IEEE, 36(10):106-108.

Microprocessor Standards Committee, 2006. DRAFT Standard for Floating-Point Arithmetic P754, IEEE.

Min C., Swartzlander, E.E., 2000. Modified carry skip adder for reducing first block delay. Circuits and Systems, Proceedings of the 43rd IEEE Midwest Symposium on, 1:346-348.

Nvidia, 2007. GeForce Family. Online. `http://www.nvidia.com/object/geforce_family.html`

Oberman, S., Favor, G., Weber, F., Mar/Apr 1999. AMD 3DNow! technology: architecture and implementations. Micro, IEEE , 19(2):37-48.

Oberman, S.F., Juffa, N., Weber, F., 2000. Method and Apparatus For Calculating Reciprocals and Reciprocal Square Roots. Advanced Micro Devices Inc., Patent Number 6.115.773

Oberman, S.F., 2002. Shared FP and SIMD 3D Multiplier. Advanced Micro Devices Inc., Patent Number 6.490.607 B1.

Singhal, R., Agu 2004. Intel Pentium 4 Processor on 90nm Technology. Hot Chips, 16

O'Connell, F.P., White, S.W., 2000. Power3: The next generation of PowerPC processors., IBM Journal of Research and Development 44(6):873-884.

Ozbilen, M.M, Gok, M., 2008. A Multi-Precision Floating-Point Adder. 4th International Conference on Ph.D. Research in Electrical and Electronics Engineering, Prime 2008, 117-120

Quach, N., Takagi, N., Flynn, M., 2004. Systematic ieee rounding on high-speed floating-point multipliers, IEEE Transactions VLSI Systems, 12:511519

Takagi, N. 1997. Generating a power of an operand by a table look-up and a multiplication. In Proceedings of 13th Sym. on Computer Arithmetic, Asilomar, 126-131

Schmookler, M.S., Mikan, D.G., 1996. Two state leading zero/one anticipator (LZA). Patent Number 5.493.520

Varghese G., Sanjeev J.,; Chao T., Smits, K., Satish D., Siers, S., Ves N.,; Tanveer K., Sanjib S., Puneet S., Nov. 2007. Penryn: 45-nm next generation Intel core 2 processor. Solid-State Circuits Conference, IEEE Asian, 14-17.

Wallace, C.S., 1964. A Suggestion for a Fast Multiplier, IEEE Transections on Electronic Computers, EC-13:14-17

Wang, Z., Jullien, G.A., Miller, W.C., Wang, J., May 1993 New concepts for the design of carry lookahead adders, Circuits and Systems, ISCAS '93, 3:1837-1840.

Weems, C., Riseman, E., Hanson, A., Rosenfeld, A., 1991. The DARPA image understanding benchmark for parallel computers. Journal of Parallel and Distributed Computing, 11:1-24.

Yang, X., Lee, R.B., 2004. PLX FP: An efficient floating-point instruction set for 3D graphics. in: ICME'04, IEEE International Conference on Multimedia and Expo, Taipei, 1:137-140.

Yang, C.L., Sano, B., Lebeck, A.R., 2000. Exploiting parallelism in geometry processing with general purpose processors and floating-point simd instructions. IEEE Trans. Comput., 49(9):934-946.

Yu, R.K., Zyner, G.B., 1995. 167 mhz radix-4 floating point multiplier. in: ARITH'95: Proceedings of the 12th Symposium on Computer Arithmetic, IEEE Computer Society, Washington, 149.

Yu-Ting P., Yu-Kumg C., Jan. 2004 The fastest carry lookahead adder, Electronic Design, Test and Applications, DELTA 2004. Second IEEE International Workshop on, 434-436

CURRICULUM VITAE

Metin Mete Özbilen was born in Tarsus in 1974. He completed his elementary education at Kayseri Ahmet Paşa Primary School in 1984. He went to high school at Kayseri Nuh Mehmet Küçükçalık Anatolia High School. He graduated from Gaziantep University department of Electrical and Electronics Engineering in 1996. He worked as electrical and electronics engineer in a company in Gaziantep from 1996 to 1998. He worked as an information technology instructor in Gaziantep Vocational High School from 1999 to 2001. He taught Database Management, Computer Hardware, Microprocessors and Operating Systems courses. He graduated from Çukurova University, department of Electrical and Electronics Engineering with degree M.Sc. in 2002. Since 2001, he has been working as a research assistant in Mersin University. He is married and father of a son and a daughter. His interest areas are computer architecture, digital design, microprocessors and operating systems and system programming.