# ON THE DESIGN OF FLOATING POINT UNITS FOR INTERVAL ARITHMETIC

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul ŞTIINŢA CALCULATOARELOR
de către

**Ing. Alexandru Amăricăi**



Conducător ştiinţific: prof.univ.dr.ing. Vlăduţiu Mircea
Referenţi ştiinţifici: prof.univ.dr.ing. Petrescu Mircea
prof.univ.dr.ing. Svasta Paul
prof.univ.dr.ing. Robu Nicolae

Ziua susţinerii tezei: 19.12.2008

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica" din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.



România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

# Cuvânt înainte

Teza de doctorat a fost elaborată pe parcursul activităţii mele în cadrul Departamentului de Calculatoare al Universităţii „Politehnica" din Timişoara. Doresc să mulţumesc colegilor din cadrul departamentului şi al Facultăţii de Automatică şi Calculatoare pentru sprijinul acordat în aceşti ani.

Mulţumiri deosebite se cuvin conducătorului de doctorat prof.dr.ing. Mircea Vlăduţiu , cel care mi-a îndrumat activitatea pe întreaga durată a doctoratului şi fără de care această teză nu ar fi fost posibilă. Totodată, profesorul Vlăduţiu este cel ce m-a introdus în domeniul proiectării hardware şi al arhitecturilor de calculatoare.

De asemenea, sincere mulţumiri pentru cei doi mari profesori de la Universitatea Politehnica din Bucuresti, pentru că m-au onorat cu participarea dumnealor în comisia de doctorat: prof. dr. ing. Mircea Petrescu şi prof. dr. ing. Paul Svasta. Doresc să îi mulţumesc în mod special rectorului Universităţii Politehnica din Timişoara, prof. dr. ing. Nicolae Robu, pentru sprijinul atât moral cât şi material acordat pe tot parcursul doctoratului.

Activitatea mea de cercetare s-a bucurat de sprijinul a numeroşi specialişti, printre care doresc să îi amintesc pe prof. Mark Arnold de la Lehigh University, USA, pe Michael Higgins de la Universitatea din Limerick, respectiv pe Valentin Mureşan de la Movidia. Nu în ultimul rând, doresc să îi adresez sincere mulţumiri prof. Emanuel Popovici de la University City of Cork, pentru sfaturile care mi-au netezit drumul în ultimele etape ale doctoratului.

Cercetarea mea s-a desfăsurat în cadrul colectivului de cercetare ACSA, căruia doresc să le mulţumesc tuturor în mod deosebit pentru sprijinul acordat. De asemenea doresc să le mulţumesc tinerilor colaboratori, şi în special lui Virgil Petcu, cu care am avut conlucrat la diverse proiecte.

Nu în ultimul rând adresez mulţumiri familiei mele, părinţilor mei, şi în special soţiei mele care m-a sprijinit continuu pe tot parcursul doctoratului.

Timişoara, Decembrie, 2008 Alexandru Amăricăi

Amăricăi, Alexandru

**On the Design of Floating Point Units for Interval Arithmetic**

**Proiectarea Unităţilor de Virgulă Flotantă pentru Aritmetica Intervalelor**

Teze de doctorat ale UPT, Seria 10, Nr. 12, Editura Politehnica, 2008, 178 pagini, 75 figuri, 37 tabele.

ISSN: 1842-7707

ISBN: 978-973-625-795-7

Cuvinte cheie: aritmetică digitală, aritmetica intervalelor, virgulă flotantă

Rezumat,
Teza abordează domeniul proiectării unităţilor de virgulă flotantă pentru aritmetica intervalelor. Sunt abordate pentru implementare trei operaţii: adunarea, înmulţirea şi operaţia combinată de divide-add fused. În ceea ce priveşte adunarea, este propus un sumator ce exploatează paralelismul arhitecturilor pe două căi pentru sumatoarele de virgulă flotantă. Pentru înmulţirea intervalelor, un algoritm nou este conceput şi este propusă o arhitectură bazată pe un înmulţitor modificat de virgulă flotantă. Unitatea de divide-add fused are scopul de a creşte performanţa metodei lui Newton cu intervale.

# ABSTRACT

*This thesis addresses the emerging problem of designing and implementing floating point arithmetic units for interval arithmetic. The present research is important in the present context of the development of the future IEEE 1788 standard for interval arithmetic. This future standard have its basis in the IEEE 754/1985 standard for floating point arithmetic and its extension draft, 754r. Therefore, all the major design decisions, such as interval representation, have been taken into accordance to these two standard proposals.*

*In this thesis, three important issues are tackled. The first two issues regard the design and implementation of hardware units for the most frequent operations, and thus the most important for the performance of any arithmetic system: addition and multiplication. The third issue was the design of a floating point unit which is dedicated for a specific interval arithmetic algorithm: the interval Newton's method. This dedicated arithmetic circuit is the floating point divide-add fused unit, which, to the best of my knowledge, has never been designed before.*

*Regarding the interval addition unit, a novel design has been implemented. This design is based on the architecture of the floating point double path adder. The proposed interval adder exploits the parallel structure of the double path adders. Based on the synthesis results, the best performance-cost tradeoff for interval addition is obtained. Furthermore, the proposed unit can be used also for increasing the performance of the conventional floating point arithmetic, due to its increase throughput and its parallel structure.*

*Interval multiplication is the most difficult interval basic operation, due to its high number of floating point operations required. The proposed interval multiplier implements a developed algorithm, which is based on two interval multiplication methods. The core of this unit is represented by the dual result multiplier (multiplication unit with two differently rounded results). In order to implement such a multiplier, three floating point multiplication rounding schemes have been adapted. Furthermore, a novel rounding scheme for interval arithmetic has been designed, which has the lowest latency and cost. Using the proposed multiplication unit, an increase in worst case performance is obtained. Furthermore, the proposed unit can be used also for set operations and conventional floating point multiplication.*

*The third floating point units is represented by the floating point divide-add fused. This unit has a similar algorithm and architecture with the floating point multiply-add fused. The main difference is the usage of a divider module, instead of multiplication circuits, such as the encoder module or the partial product reduction tree. The purpose of such combined*

*floating point unit is to increase the performance of the interval Newton's algorithm, which has as its core operation a division followed by a subtraction. The main problems regarding the design and implementation of such unit are the number of required quotient bits and the rounding problem. An in-depth analysis of these two problems is performed. Several implementations for such a unit are proposed in this thesis, depending on the desired precision or performance.*

## Published Papers and Impact

This thesis is supported by the following papers:

- ***Amaricai,*** M. Vladutiu, L. Prodan, M. Udrescu, O. Boncalo - **Design of Addition and Multiplication Units for High Performance Interval Arithmetic Processor,** Proceedings 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Krakow, Poland, April 11-13, 2007, pp. 223-226 (ISI Proceedings, IEEEXplore)
- ***A. Amaricai***, M. Vladutiu, L. Prodan, M. Udrescu, O. Boncalo - **Hardware Support for Combined Interval and Floating Point Multiplication,** Proceedings 14th Mixed Design of Integrated Circuits and Systems (MIXDES), Ciechocinek, Poland, June 21-23, 2007, pp. 278-282 (ISI Proceedings, IEEEXplore
- ***A. Amaricai***, M. Vladutiu, L. Prodan, M. Udrescu, O. Boncalo - **Exploiting Parallelism in Double Path Adders' Structure for Increased Throughput of Floating Point Addition** Proceedings 10th Euromicro Symposium on Digital System Design (DSD), Lubeck, Germany, August 29-31, 2007, pp. 132-137 (ISI Proceedings, IEEEXplore)
- V. Petcu, ***A. Amaricai,*** M. Vladutiu - **A Dual-Threaded Architecture for Interval Arithmetic Coprocessor with Shared Floating Point Units** Proceedings 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Bratislava, Slovakia, April 16-18, 2008, pp. 146-150 (ISI Proceedings, IEEEXplore)
- ***A. Amaricai***, M. Vladutiu, M. Udrescu, L. Prodan, O. Boncalo - **Floating Point Multiplication Schemes for Interval Arithmetic** – Proceedings 19th IEEE Conference on Application-Specific Systems, Architectures and Processors (ASAP), Leuven, Belgium, July 2-4, 2008, pp 19-25 (ISI Proceedings, IEEEXplore)
- ***A. Amaricai,*** M. Vladutiu, L. Prodan, M. Udrescu O. Boncalo - **Floating Point Divide-Add Fused Unit for Interval Newton's Method** Proceedings Euromicro Work-in Progress Session, Parma, Italy, September 3-5, 2008

Prior to the thesis, two PhD. Reports had prepared this work:

- **On the Design of Floating Point Units for Interval Addition and Multiplication**
- **On the Design of Floating Point Divide-Add Fused Units**

## TABLE OF CONTENTS

## List of Figures

## List of Tables

# 1. INTRODUCTION

## 1.1. The Need for Reliable Arithmetic

Due to advances in the computing technology, at any level (architectural, gate, transistor, etc), the computational power has increased in an almost exponential way in the last decades [41][53][86]. Therefore, the most powerful computing systems can perform billions of arithmetic operations per second.

A very important place in most computational system is represented by the floating point systems. Floating point numbers are used for real number representation. Unlike the fixed point numbers (used for integer computations), the floating point operations do result in errors [41]. These errors are due to the following reasons: on one hand there is a wide range of real numbers which cannot be represented exactly using floating point representation (for example $\sqrt{2}$, $1/3$, $\pi$ etc) – in this case an approximation of the real number which can be represented in the floating point number system is used; on the other hand, the result from a floating point operation cannot be represented using the floating point representation – in this case rounding or truncation operations are performed. Thus, floating point computations are prone to errors, as shown by the simple example presented in Fig. 1.1.

```
10^20+10.0+137.0-10^20-17.0= -17.0
10^20-10^20+10.0-17.0+137.0= 130.0
10^20+137.0-17.0+10.0-10^20= 0.0
10.0+10^20-10^20-17.0+137.0= 120.0
137.0-17.0-10^20+10^20+10.0= 10.0
```

Figure 1.1 – Example of Erroneous Results Produced During Floating Point Computations [53]

The large amounts of floating point computations also increases the probability of erroneous results, as error may accumulate. Therefore, methods for controlling these rounding and truncation errors must be provided for floating point systems where the reliability represents an important issue.

## 1.2. Interval Arithmetic

One method for controlling errors which can occur in floating point computations is represented by interval arithmetic. Interval arithmetic is the arithmetic of intervals [40]. It does not work with a single floating point number, as in the conventional floating point arithmetic, but uses two floating point numbers which define an interval. The two floating point numbers in majority of applications represent the lower (inferior) bound of the interval and the upper (superior) bound

of the interval. Another less used form of representing an interval consists of using a floating point number which may represent one bound (lower or upper) or the midpoint and another floating point number which represents the width or the radius of the interval [54].

Interval arithmetic does not increase the accuracy or the precision of the computation. However, the interval represents a measure of the error accumulation during the intensive computations, and therefore, measures can be taken during in order to increase the reliability of the operations (like repeating computations with greater precision) [47][53][54][58]. More important, a wide range of specific interval methods have been developed, which provide guaranteed and very reliable results [25].

### 1.2.1 Number Representation

A very important issue regarding interval arithmetic is represented by number representation. Two formats have been proposed for representing floating point numbers used in interval arithmetic.

One format, proposed by Schulte [86][87], is based on variable precision floating point numbers. A number represented in variable precision format is composed from two parts: a header word and a variable size mantissa part (consisting of more words) – Fig 1.2. The header word contains the exponent bits, the sign bit, a type field (which indicates if the number is either a conventional variable precision number, or denotes a special value – zero, infinity, not-a-number), and a field which indicates the number of mantissa words. The mantissa is composed from a variable amount of words, depending on the required precision for the represented number. A normalized mantissa has its value in the $\left[\frac{1}{2},1\right)$ range. The value of the floating point number is given by the following formula:

$$N = (-1)^S * 2^E * M \qquad (1.1)$$

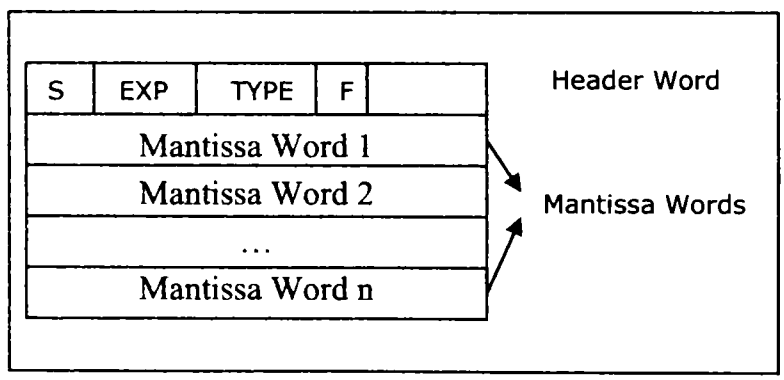



Figure 1.2 – Variable Precision Number Format [86][87]

The main advantage of this representation is the increased accuracy. If the required precision must be increased, mantissa words may be added. Furthermore, rounding and truncation errors are less frequent and have smaller values, because, in case of an operation which needs rounding (the result needs more bits for precision) the number of mantissa words for the result is increased.

The disadvantages of this type of representation are related to the cost and performance of hardware units which implement operations. The reason for this is that the variable precision representation requires operations executed for each word of mantissa.

Another format used for representing intervals is the representation used in IEEE 754 floating point standard [36]. This is the standard used for representing floating point numbers. A number represented in IEEE floating point numbers has three fields: sign, exponent and mantissa. The number of bits used for each of the three fields varies from the specified precision. The IEEE 754 standard defines two types of precision (simple precision on 32 bits and double precision on 64 bits), while IEEE 754r extension draft specifies four types of precision (half precision on 16 bits, simple precision on 32 bits, double precision on 64 bits and quad precision on 128 bits) – Fig 1.4. The four formats are depicted in Fig 1.3. Also, two extended precision formats are specified (simple extended on 44 bits and double extended on 80 bits) [36].

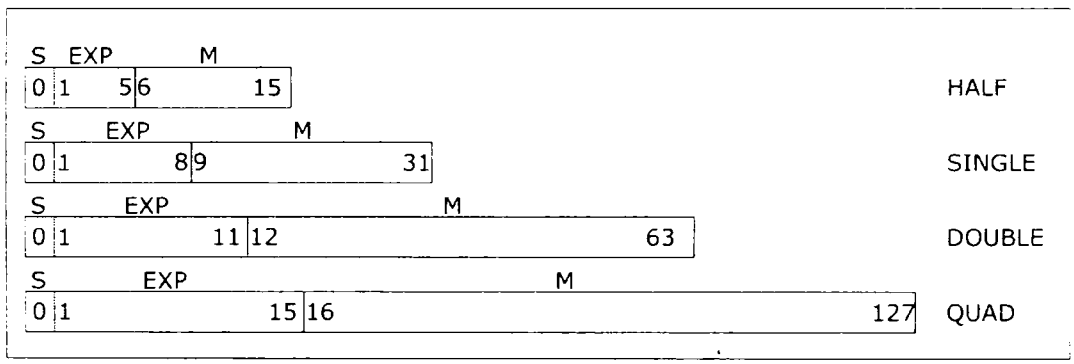



Figure 1.3 – Number Representation in IEEE 754r Standard [36]

The value of a number represented in IEEE 754 representation is given by the following equation:

$$N = (-1)^S * 2^{E-bias} * 1.M \quad (1.2)$$

The exponent's representation is in a biased form. The bias is equal to 15 for half precision, 127 for simple precision, 1023 for double precision and 16383 for quad precision. Another important feature of this standard is represented y the hidden 1 for the mantissa. Using this hidden 1, a bit of precision is thus gained. Also, this hidden one establishes the range for the mantissa value between $[1,2)$ [26][36]. Thus, normalization steps must be performed in every operation, so the mantissa remains in the considered range [26][51][74].

Another important feature of this standard is represented by the special values which are defined. These special values are specified by a specific combination of the three fields of the IEEE 754 floating point numbers. These special values are [36]:

- **Zero**: this special value is determined by zero in the exponent field and zero in the mantissa field. However, the sign bit can be 0 or 1, and thus, we have a signed zero (positive and negative zero).

- **Infinity**: this special value is determined by an all 1 exponent field and a zero mantissa field. As in the case of zero, the infinity special value is signed (positive and negative infinity).

- **Not a Number (NaN)**: this special value is determined by an all 1 exponent field and a non-zero mantissa. This value results when operations with special fields take place (square root from a negative number, $\frac{0}{0}, \frac{\infty}{\infty}, 0 * \infty$, $(+\infty) + (-\infty)$, etc).

- **Denormalized Numbers:** this special value is determined by zero in the exponent field and a non-zero mantissa field. This value results when the normalization cannot be performed (the operation has an underflow result). This special value is important, because it can be avoided, by the gradual underflow mechanism, the total cancellation of the result.

The IEEE 754 also defines five types of exceptions (overflow, underflow, divide by zero, invalid and inexact), allowing thus the implementation of specialized trap handlers for each of the five exceptions [36]. Also, the standard recommends a minimum set of operations to be implemented as instructions: addition/subtraction, multiplication, division, remainder, square root and conversion between floating point and integer. For these operations the standard requires that all the operation to be performed so the result is computed exactly and then rounded according to the desired rounding mode. Furthermore, IEEE 754 standard defines four rounding modes: rounding towards nearest even, rounding towards zero (truncation), rounding towards positive infinity and rounding towards negative infinity [53].

Compared to the variable latency format, the IEEE 754 standard does not have the same accuracy in computations. However, the hardware units which implement the arithmetic operations for IEEE 754 standard have a greater performance.

The IEEE 754 and the extension draft IEEE 754r standard have become the basis for the first interval arithmetic dedicated standard, which is the goal of the IEEE Working Group P1788 approved by the IEEE-SA Standards Board [116]. Therefore, all the designed hardware units in this thesis are designed for IEEE representation of floating point numbers.

### 1.2.2 Interval Operations and Function Evaluation

Given two intervals $[X_{lo}; X_{hi}]$ and $[Y_{lo}; Y_{hi}]$, where $X_{lo}, X_{hi}, Y_{lo}, Y_{hi}$ are floating point numbers, the four basic arithmetic operations are defined as follows [49][53][47][86][94]:

- **Addition**:

$$[X_{lo}; X_{hi}] + [Y_{lo}; Y_{hi}] = [X_{lo} + Y_{lo}; X_{hi} + Y_{hi}] \quad (1.3)$$

- **Subtraction**:

$$[X_{lo}; X_{hi}] - [Y_{lo}; Y_{hi}] = [X_{lo} - Y_{hi}; X_{hi} - Y_{lo}] \quad (1.4)$$

- **Multiplication:**

$$[X_{lo};X_{hi}]*[Y_{lo};Y_{hi}]=\left[\min\left(\prod XY\right);\max\left(\prod XY\right)\right]$$
$$\prod XY \Leftrightarrow X_{lo}*Y_{lo};X_{lo}*Y_{hi};X_{hi}*Y_{lo};X_{hi}*Y_{hi} \quad (1.5)$$

- **Division:**

$$[X_{lo};X_{hi}]\Big/[Y_{lo};Y_{hi}]=[X_{lo};X_{hi}]*1\Big/[Y_{lo};Y_{hi}] \quad (1.6)$$

Undefined for $0\in[Y_{lo};Y_{hi}]$

The rounding modes used in interval arithmetic are rounding towards negative infinity for the lower bound of the result and rounding towards positive infinity for the upper bound of the result [41][47].

As it depicted by the equations (1.5) and (1.6), multiplication and division are not as straightforward as the interval addition and subtraction. The reason for this is that an interval operation is defined as follows [41][47]:

$$[X_{lo};X_{hi}]\circ[Y_{lo};Y_{hi}]=\left[\min(x\circ y),\max(x\circ y)\right],\quad \forall x\in[X_{lo};X_{hi}],\forall y\in[Y_{lo};Y_{hi}] \quad (1.7)$$

These four basic interval operations are commutative and associative, while regarding the distributivity property in the case of interval arithmetic it does not hold. However, the subdistributivity property holds [47][49][53][107]:

$$[X_{lo},X_{hi}]*([Y_{lo},Y_{hi}]+[Z_{lo},Z_{hi}])\subseteq [X_{lo};X_{hi}]*[Y_{lo};Y_{hi}]+[X_{lo};X_{hi}]*[Z_{lo};Z_{hi}] \quad (1.8)$$

A function evaluation for an interval is defined in the same way as an operation [41][47][53][107]:

$$f([X_{lo};X_{hi}])=\left[\min(f(x)),\max(f(x))\right],\forall x\in[X_{lo};X_{hi}] \quad (1.9)$$

Therefore, using (1.8) $\sin([0,2*\pi])=[-1,1]$ and not $\sin([0,2*\pi])=[\sin(0),\sin(2*\pi)]=[0,0]$, while $f([-2,2])=[0,4]\neq[4,4]$ for $f(x)=x^2$ [41][53].

Very important for interval arithmetic are also the set operations. These operations are used in a wide range of interval methods. Among these operations are included [1][91]:

- **Hull**

$$[X_{lo};X_{hi}]\cup[Y_{lo};Y_{hi}]=\begin{cases}\left[\min(X_{lo},Y_{lo}),\max(X_{hi},Y_{hi})\right], \\ \quad \text{when } X_{hi}\geq Y_{lo}\geq X_{lo} \text{ or } Y_{hi}\geq X_{lo}\geq Y_{lo} \\ [X_{lo};X_{hi}]\cup[Y_{lo};Y_{hi}] \quad \text{,otherwise}\end{cases} \quad (1.10)$$

- **Intersection**

$$[X_{lo}; X_{hi}] \cup [Y_{lo}; Y_{hi}] = \begin{cases} [\max(X_{lo}, Y_{lo}), \min(X_{hi}, Y_{hi})], \\ \quad \text{when } X_{hi} \geq Y_{lo} \geq X_{lo} \text{ or } Y_{hi} \geq X_{lo} \geq Y_{lo} \\ \varnothing \quad \text{,otherwise} \end{cases} \tag{1.11}$$

Also used in many interval applications are operations on the interval, off which two are more frequently used [41][47][53][107]:

- **Width**

$$Width([X_{lo}; X_{hi}]) = |X_{lo} - X_{hi}| \tag{1.12}$$

- **Midpoint**

$$m([X_{lo}; X_{hi}]) = \frac{X_{lo} + X_{hi}}{2} \tag{1.12}$$

The first of these two operations is used for determine the accuracy and the error accumulation of the interval computations, while the second is used for approximating an interval with a single floating point number.

### 1.2.3 Applications

Interval methods have been devised for a wide range of applications which require reliable and guaranteed results [25][41][53][54][58][61][107]. Basically, interval arithmetic methods produce numerical proofs. Furthermore, some interval algorithms present even a better performance compared to their conventional counterparts.

Historically, the first time interval methods have been used by Archimedes to determine the value of number $\pi$ [41][53]. However, modern day interval arithmetic has been refined and developed since the works of Ramon E. Moore in the 1960s [61]. Since then, a wide range of specific methods have been developed which can be used in many fields of applications.

One of the most important fields of application is represented by the nonlinear equations and systems of equations. Two types of methods can be used for this type of application: bisection methods and interval Newton's method [47][48][53]. The interval Newton's method has the property of providing a very narrow interval which will surely contain the solution of the equations or providing a certain indication of the no solution situation. This method can be used for all types of nonlinear equations and systems of nonlinear equations, many of which couldn't be solved with numerical methods. Furthermore, interval methods have been developed for solving linear systems of equations, integral equations, initial value problems, etc.

Therefore, these types of methods, and especially interval Newton's method, have a wide range of applications. One field of applications is related to computer graphics, where equations solving represent one of the most common operations. Based on interval methods, algorithms for ray tracing, ray-surface intersection, rendering, collision detection have been developed, many of which presenting a better performance compared to their conventional counterparts [47][84].

Interval methods have also been devised for control theory and robotics. One of the main research program related to this field of application is the COPRIN project developed at INRIA institute [117].

Interval methods have also been used to provide computer assisted proofs for mathematical physics, like the Feigenbaum conjecture, the double bubble conjecture or Keppler conjecture [47][81]. Also, interval arithmetic has been used for determining physical constants, like the Newton's gravity constant. Other applications for interval arithmetic have been developed in chemical engineering, electrical engineering, computer-aided design, fluid mechanics, dynamical systems, air traffic control, etc.

### 1.2.4. Software Support

In order to enable support for these applications, software extensions of the common programming languages which include interval arithmetic have been provided [53][94].

SUN Microsystems have developed, during the Interval Computation project, a C++ extension (Forte Developer 7, Sun Studio 11) and FORTRAN extension for interval arithmetic [97]. The GNU C compiler also has been modified for super-scalar architecture to support interval arithmetic [94]. Furthermore, the INTLAB is an extension of the MATLAB which have been developed at the TU-Hamburg [115].

Support is also provided in specific libraries, which contain interval methods and algorithms (ALIAS [112], CGAL [113], MPFI [114], BOOST Interval [111] etc). Some of these libraries present also optimization techniques particular for interval arithmetic.

Furthermore, scientific extensions of the common programming languages contain support for interval arithmetic. Some of these extensions include the Pascal-XSC (a Pascal extension), C-XSC or ARITH-XSC [41][49][53][94].

## 1.3. Motivation

Although a wide range of applications have been developed based on interval arithmetic, these methods are quite slow and inefficient on modern computers, even if some kind of software support exists for interval arithmetic. The main reason is represented by the lack of hardware support for interval arithmetic [49][53].

Even though the IEEE 754 standard specifies four rounding modes, of which the rounding towards negative infinity and rounding towards positive infinity (the rounding modes used in interval arithmetic), only rounding towards nearest even is implemented within the arithmetic operations. For using other rounding modes, a dedicated instruction must be executed before the operation. Therefore, for interval addition for example, instead of two instructions (one for each floating point operation), four instructions must be used, of which two instruction for changing the rounding mode (instruction for rounding towards negative infinity, addition, instruction for rounding towards positive infinity, addition) [49][53]. Thus, the performance of an interval addition is low compared to that of two conventional floating point additions (rounded towards nearest even). Although interval hardware units have been proposed in literature [1][49][53][86][87][91][94][95][96][107],

there is no commercial processor which offers appropriate support for interval arithmetic.

The reasons for this lack of support for interval arithmetic, as explained by William Walster from SUN Microsystems, are twofold: on one hand, the uncertainty in demand, thus preventing microprocessor companies to implement interval hardware units and interval arithmetic instructions; on the other hand, there is no standard for interval arithmetic [41]. Lack of standard may be the most important reason, as it was in the case of conventional floating point arithmetic: hardware implementations and support for floating point knew a development (almost a boom) only after the emergence of IEEE 754 standard [41].

In order to encourage the development of hardware support for interval arithmetic, in the last year the IEEE Standard Association has appointed the Working Group 1788 to develop a standard for interval arithmetic [54][116]. This standard has its starting point the IEEE 754 standard for floating point and its extension draft IEEE 754r (especially related to number format), while specific issues of the interval arithmetic (like division by an interval which contains zero) must be specified by this standard.

Thus, there is an increasing need for hardware support for interval arithmetic. The arithmetic units must comply to the IEEE 754 number formats, as the interval arithmetic standard will be implemented with this type of number format. Furthermore, it is desirable that conventional floating point operations could be executed on the hardware interval arithmetic units.

## 1.4. Thesis Objectives

This thesis has three major objectives: the design of an interval/conventional floating point addition unit, the design of an interval/conventional floating point multiplication unit and the design of a floating point divide-add fused unit.

Regarding the adder, in conventional floating point arithmetic the operations which are executed on this unit count (additions, subtractions and comparisons) count about 55% from all floating point operations [69][70]. Therefore, the floating point adder is critical to the performance of any floating point system. In interval arithmetic it is also expected that the addition/subtraction to represent as vital operation as in the conventional floating point arithmetic. Therefore, the design of a high performance interval adder represents a priority for any interval arithmetic hardware processor. Furthermore, this interval hardware should be also used for conventional floating point addition. One reason for the combined functionality is that two important functions used in interval arithmetic require additions/subtractions: width of the interval and midpoint of the interval.

Regarding the multiplication, in conventional floating point arithmetic this operation counts about 40% from all floating point operations [69][70]. Therefore, the floating point multiplier is also a very important unit for any floating point system, its performance being critical for the overall performance. In interval arithmetic, it is also expected that the multiplications to be as important as in conventional floating point arithmetic. As in the case of the interval addition unit, the multiplication unit should be able to also perform conventional floating point multiplications.

The floating point divide-add fused will be a dedicated arithmetic unit for increasing the performance of the interval Newton's method [48][49][53]. In this

case, the use of a dedicated combined unit may represent an advantage, as is the case of the floating point multiply-add fused for many applications (digital signal processing, computer graphics, etc). This dedicated combined unit can also be used for addition or division (both conventional and interval), but with lower performance compared to an adder or a divider. Because interval Newton's method represents one of the most important interval algorithms, a dedicated unit for this operation will be beneficial for an interval arithmetic system.

Previous interval designs, such as the ones in [1][49][53][86][87][91][94][95][96][107], used conventional floating point units, without making changes in their architectures. The approach used for designing interval hardware units is based on optimizations in the internal structure of conventional floating point units. Therefore, cost and performance improvements can be obtained by using this approach.

## 1.5. Evaluation

The proposed designs were implemented using VHDL. The role of these VHDL models is both verification and performance and cost evaluation [45].

In order to determine the performance of the proposed designs, IEEE 754 double precision models were built. These models using the double precision were simulated using as a latency measure the logic level (LL) as in the works of Seidel-Even [30][89][90]. The proposed designs were compared to other floating point designs, whose results are reported in literature. This type of comparison is technology independent, but can be inconclusive due to several reasons:

- It does not take into account the wire delays
- It does not take into account delays produced by signal buffers
- Different gates have different delays for different technologies (for example the XOR gate has a larger delay compared to an AND gate, while multiplexers built with pass transistors may be faster than most logic gates [14]).

Therefore a technology dependent analysis had to be performed. For this type of analysis, designs based on IEEE half precision number formats were built. These designs were synthesize using Xilinx Synthesis Tool (XST) from the Xilinx ISE Webpack 10.1 [108]. The synthesis was done for the Xilinx Virtex-4 FPGA family. Models and synthesis results were performed for both the proposed designs and for other designs present in literature.

Table 1.1 – Synthesis results obtained for benchmark circuits

| Benchmark Circuits | Maximum Combinational Delay (ns) | | | Cost (LUT-4) |
|---|---|---|---|---|
| | Logic | Route | Total | |
| C17 | 0.147 | 0.266 | 0.413 | 2 |
| C432 | 2.058 | 6.588 | 8.646 | 67 |
| C499 | 1.176 | 3.510 | 4.686 | 110 |
| C6288 | 4.410 | 15.899 | 20.309 | 493 |
| 74181 | 0.753 | 2.234 | 2.987 | 22 |

Furthermore, to increase the confidence in the performed evaluation five benchmark circuits from the ISCAS'85 benchmark circuits family were modeled in VHDL and synthesized using the same tool for the same technology [39][118]. The VHDL code of this five benchmark circuits is given in Appendix A. The five benchmark circuits considered are:

- C17
- C432 27-bit channel interrupt controller
- C499 32-bit single error correcting circuit
- C6288 16-bit*16-bit array multiplier
- 74181 4-bit arithmetic logic unit

The obtained results are presented in Table 1. The latencies are given in nanoseconds, representing the maximum delay on the longest path of the synthesized circuit. As it can be observed, the maximum delay is composed from a logic delay and a route delay. The area is given in 4-input look-up tables (LUT-4), which represents the basic element of the FPGA family [108]. This way, a technology dependent analysis of the proposed designs could be obtained.

## 1.6. Thesis Outline

This dissertation is organized in three main chapters. These chapters are dedicated each for one interval hardware unit. The second chapter presents the interval addition hardware unit, the third chapter presents the interval multiplication unit, while the fourth chapter is dedicated to the floating point divide-add fused unit.

The second chapter is dedicated to the interval addition unit. This chapter is organized in three main sections. The first section is dedicated to the interval addition. Algorithms are presented and hardware implementations are discussed. The second section is dedicated to the conventional floating point addition. The basic algorithm and its hardware implementation are presented. The main issues regarding the single path adders are than discussed. The final subsection of this section dedicated to the conventional floating point addition represents a critical birds' eye view of the double path adders. This is a very important issue for this thesis, as the proposed adder has as the inspiration point the double path adder. The last section of this chapter dedicated to the interval addition unit is dedicated to the proposed solution. The algorithm and the architecture of the proposed adder are presented. Last, but not least, cost and performance evaluations are made.

Regarding the third chapter, dedicated to the interval multiplication, the outline is similar to the one of the second chapter. Three main sections compose this chapter. In the first section, algorithm and hardware implementations for interval multiplication are discussed and compared. In this chapter, this section is far more elaborated than its counterpart of the addition chapter, due to the fact that interval multiplication is much more complicated compared to interval addition. The second section presents the main issues regarding the conventional floating point multiplication. The integer unsigned multiplication is presented. The focus on this section will be the tree multipliers, every main component of them being analyzed: encoding module and its implemented multiplication algorithm, the partial product reduction tree topologies and the final addition and rounding scheme. The third section of this chapter is dedicated to the proposed solution. A proposed algorithm

and its according architecture are presented. Last, but not least, cost and performance evaluations are made.

The fourth chapter is dedicated to the floating point divide-add fused unit. The first section presents some considerations of the floating point divide-add fused. The second section of this fourth chapter presents the interval Newton's method, which is the algorithm for which the floating point divide-add fused is designed. Thus, motivation for the implementation of such unit will be provided. The third section of this chapter is dedicated to the floating point multiply-add fused. The floating point multiply-add fused unit is very important in this context because it is the only combined operation implemented floating point unit. In the fourth section an insight in the floating point division is given. The main algorithms and design choices are discussed. The fifth section of this chapter presents the proposed solution. The main issues, algorithm and the hardware implementation of this unit are presented. Like the other two previous chapters, this one ends with cost and performance evaluations.

The last chapter of this thesis is dedicated to the concluding remarks. This chapter is organized in three sections. The first section presents the summary of this thesis. The second section is dedicated to the contributions of this thesis. In the last section, open problems and future work are discussed.

# 2. Hardware Interval Addition Unit

## 2.1 Interval Addition

Interval addition and subtraction are defined in equations (2.1) and (2.2) [49][53][86][87][107]:

$$[X_{lo}; X_{hi}] + [Y_{lo}; Y_{hi}] = [RNI(X_{lo} + Y_{lo}); RPI(X_{hi} + Y_{hi})] \quad (2.1)$$

$$[X_{lo}; X_{hi}] - [Y_{lo}; Y_{hi}] = [RNI(X_{lo} - Y_{hi}); RPI(X_{hi} - Y_{lo})] \quad (2.2)$$

As it can be seen from above, the interval addition/subtraction require two floating point operations, one rounded RNI, while the other rounded RPI.

As described in the listed references, two ways for performing an interval addition/subtraction operation are given. One way is to use a single floating point adder (which incorporates both RNI and RPI) [53]. This way, the performance of the interval operations is equal to the performance of two conventional floating point additions/subtractions, while the area required is almost the same as a conventional floating point adder (is a little bit greater because of a pair of multiplexers) [94].

The second way for performing interval addition is by using two floating point adders (one with RNI and one with RPI) – fig. 2.1. This adder requires two pairs of multiplexers to be adder. The performance in this case is equal with respect to the conventional floating point addition, while the area required is more than double [49].

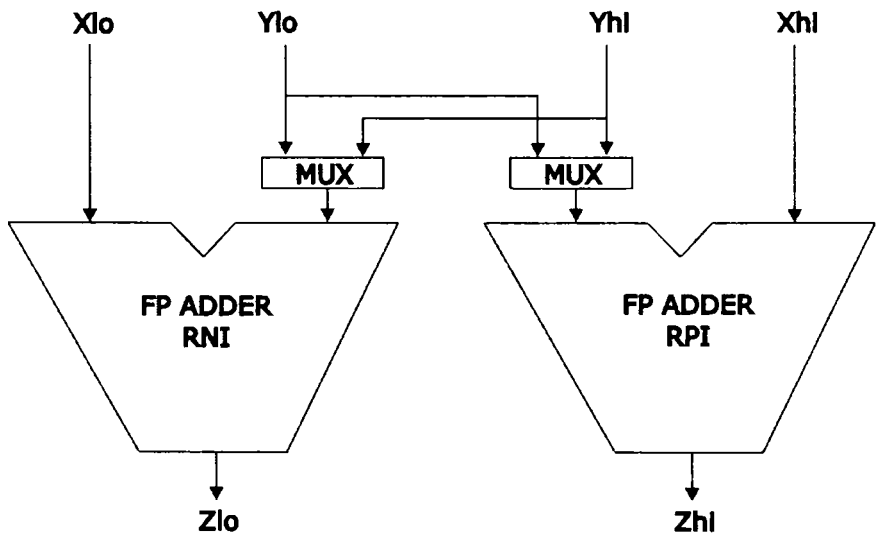



Figure 2.1 – Interval Adder Comprised of Two Floating Point Adders [49]

These two ways for performing interval additions/subtraction do not require major changes in the structure of the floating point adders used (mainly because all IEEE compliant floating point adders include, beside RNE and RZ, RNI and RPI [36]).

## 2.2 Floating Point Addition

### 2.2.1 Basic Algorithm

Floating point addition is one of the most difficult floating point operations. The addition of two IEEE floating point numbers ($F1 = (-1)^{s1} * 2^{E1-bias} * 1.M1$ and $F2 = (-1)^{s2} * 2^{E2-bias} * 1.M2$, with $F1>F2$) is given by the following formula [64]:

$$s3 * 2^{E3-bias} * 1.M3 = F1 \pm F2 = (-1)^{s1} * 2^{E1-bias} * \left(1.M1 \pm 2^{E1-E2} 1.M2\right) \quad (2.3)$$

The actual operation to be performed (addition or subtraction – the effective operation) is determined by the instruction and the sign of the operands (Table 2.1). The two mantissas are in the $[1;2)$ range, so, after an addition/subtraction, the mantissa of the result is in the $[0;4)$ range. Thus, normalization steps are required [69].

Table 2.1 – The Effective Operation in Floating Point Addition

| Operation | Sign 1 | Sign 2 | Effective Operation |
|---|---|---|---|
| Addition | + | + | Addition |
| Addition | + | - | Subtraction |
| Addition | - | + | Subtraction |
| Addition | - | - | Addition |
| Subtraction | + | + | Subtraction |
| Subtraction | + | - | Addition |
| Subtraction | - | + | Addition |
| Subtraction | - | - | Subtraction |

The basic floating point addition, as described in [64][78], consists of the following steps:

**1. Exponent subtraction**. This step is important because of two reasons: the greater exponent will be used for the computation of the result exponent and the difference represents the amount of alignments shift.

**2. Mantissa alignment.** The mantissa of the number with the smaller exponent will be right shifted with the amount given by the difference of the exponents.

**3. Mantissa addition/subtraction.** The two mantissas will be added or subtracted, based on the effective operation.

**4. Result conversion.** In case of a negative result, a two's complement have to be performed.

**5. Leading zero detection.** In case of an effective subtraction, a massive cancellation of the result is possible. In this case, normalization is needed (using left shifting). The amount of left shifts results from this step.

**6. Normalization.** The result of the mantissa is normalized (left shifting in case of leading zeros or 1 position right shift if the mantissa is in the $[2;4)$ range). This step is followed by an exponent update (subtracting from the greater exponent the amount of left shifting in case of cancellation or adding 1 to the greater exponent in case of right shift normalization).

**7. Rounding.** The rounding decision is computed and the 1 ulp addition is performed, if required.

**8. Post-normalization.** A 1 position right shift might be required if the exponent is equal to 2.

This basic algorithm has an unacceptable high latency, as described in [64]. This is due steps 2 and 6 (the massive shifts), steps 3, 4 and 7 (which require large carry propagate adder – step 4 and 7 requiring these adders only to add 1 ulp), and step 5. Significant improvements have to be made in order to obtain the high performance needed for the most frequent floating point operation.

### 2.2.2 Single Path Adders

In order to increase the performance of the floating point addition, several improvements can be made. They rely on removing some high latency steps of the basic algorithm from the critical path, either by reducing them to much lower latency operation, or by performing them in parallel with other steps.

A first improvement can be made by swapping the mantissas, based on the exponents' difference [12][64][78]. This way, the smaller number will be subtracted from the greater number, thus avoiding the two's complement of the result. The only case when this is not possible is the case of equal exponents. Therefore, step 4 of the basic algorithm is not completely removed.

A second improvement can be made by using a leading zero predictor instead of a leading zero detector [12][16][37][63][64][65][73][85][89][90]. This circuit predicts the number of leading zeros based on the two aligned mantissas. Therefore, the leading zero predictor can run in parallel with the mantissa addition/subtraction. Thus, the high latency operation of detecting leading zeros is removed from the critical path of the floating point adder.

A third improvement can be achieved by using a compound adder [12][17][37][63][64][65][66][88][89]. The compound adder is a type of adder which has two results: the sum of two numbers ($A+B$) and the incremented sum ($A+B+1$) [98]. Using such adders, the steps which need the addition of 1ulp (the result conversion step and the rounding step) are reduced to a simple selection (using a single row of multiplexers). In this way the latency is reduced significantly. This module will be detailed in Section 2.2.2.2.

Thus, it is obtained the single path adder. The critical path for the mantissa of this type of adder is build from the exponents' subtractor, the swapping circuit, the alignment shifter, the bit inversion circuit, the compound adder, the rounding and bit inversion multiplexer, the normalization left shifter and the one position right shifter. The overall structure of this type of floating point adder is presented in Fig. 2.2.

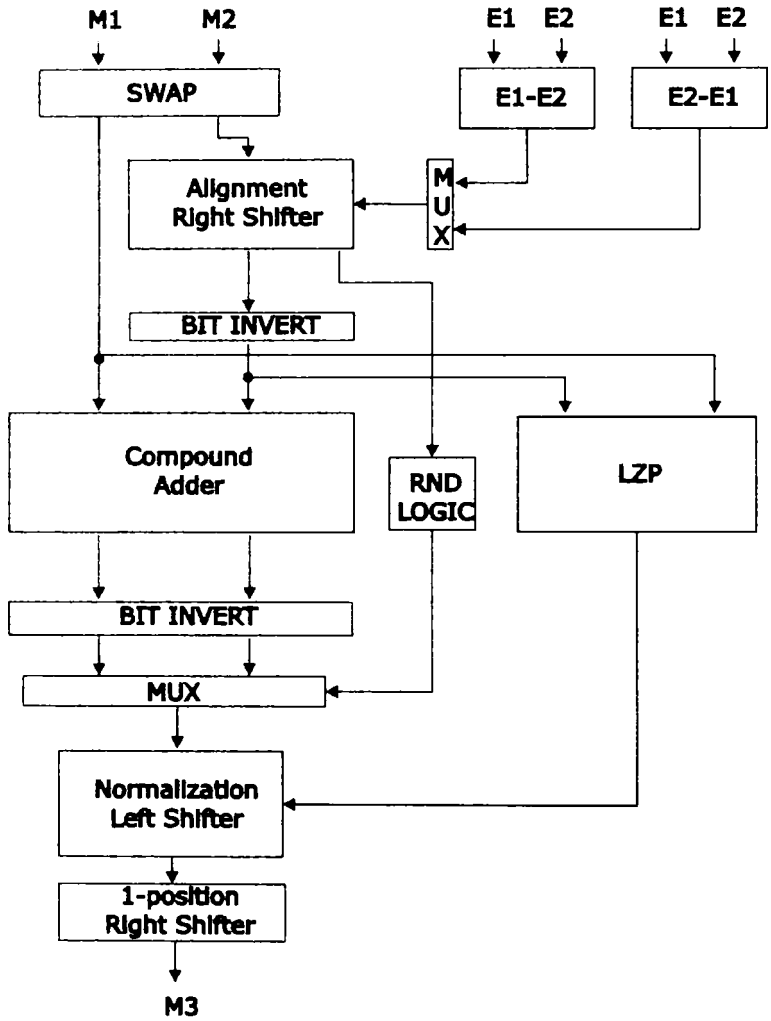



Figure 2.2 – Block Architecture of a Single Path Adder [12]

A typical single path adder is presented in [12], which is similar with the one in Fig.2.2. The design presents all of the improvements presented above. This adder has a performance of three clock cycle (three pipeline stages). In the first clock cycle the exponent subtraction and the alignment shifting are performed. The second pipeline stage is dedicated to the mantissa addition and the leading zero detector. The normalization shift is performed in the last stage. Thus, a significant improvement of the performance of the floating point addition compared to the basic algorithm is obtained.

### 2.2.2.1 Leading Zero Prediction

The leading zero predictor (LZP) (also called leading zero anticipator (LZA)) is a circuit which predicts the number of leading zeros based on the two operands of the subtraction (which will be called in this section *A* and *B*) [16][43][85]. Therefore, the leading zero predictor can run in parallel with the adder which performs the subtraction. Therefore, this dedicated circuit improves the performance, unlike the leading zero detector (LZD) [73], which detects the numbers of leading zeros based on the result of the operation (*A-B*). The difference between the two circuits can be seen in Fig. 2.3. Leading zeros do appear when the result of the operation is positive and leading ones do appear when the result of the subtraction is negative [85]. The leading zero prediction is actually the determination of the most significant one bit. Due to this, in some papers (for example [16]), leading zero prediction is called leading one predictor (prediction of the most significant one digit).

A LZP is composed of three modules:

- encoding logic
- leading zero detection
- correction module.

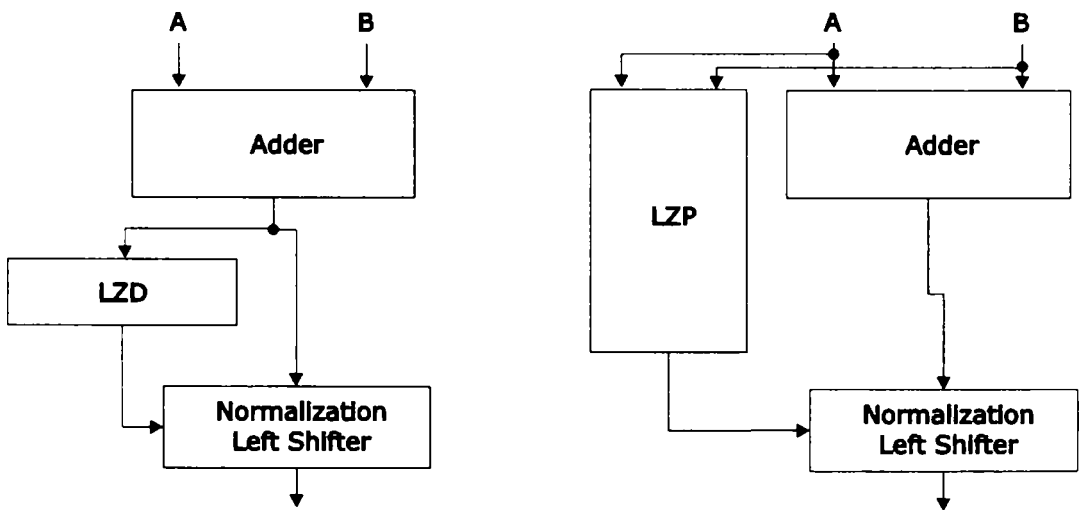



Figure 2.3 – Placement of Leading Zero Detector (LZD) and Leading Zero Predictor (LZP) [80]

Leading zero predictor detects the position of the first leading one by examining the pattern of the two operand bits. The bits of the same weight in the two operands are compared and encoded. One method of encoding is presented in [16]. This type of encoding determines the relative position if the two bits of the same weight (three bits are used, the equal bits ( $e_i = \overline{a_i \oplus b_i}$ ), the greater bits ( $g_i = a_i \overline{b_i}$ ) and the smaller bits ( $s_i = \overline{a_i} b_i$ )). Leading zeros do appear in case of $e^k g s^j$ (a string of $k$ equal bits, followed by a greater bit and $j$ smaller bits) pattern

for the most significant position. Leading ones (in case subtraction has a negative results) do appear in case of $e^k sg^j$ pattern [85]. Based on these encoding bits, a string of bits is generated which indicates if a one may appear on the position (the *f* string)[16][43][85]. The encoding cell for one position is presented in Fig. 2.4.

The second stage of a LZP is the LZD, which is a circuit which detects the position of the most significant one. A typical LZD is described in [73]. In order to decrease the latency of this circuit, the detection of the first significant one is done in a binary tree based structure. At the first level, a group of four bits is analyzed, at the second level the detection is realized for eight bits. For each group of analyzed bits (four for first level, eight for second level, sixteen for third level, etc) a valid bit which indicates if a one is contained and the position of the most significant one are generated. The generation of valid and position bits for a group of four bits are presented in table 2.2. The structure of the four bit detection is presented in Fig 2.5.

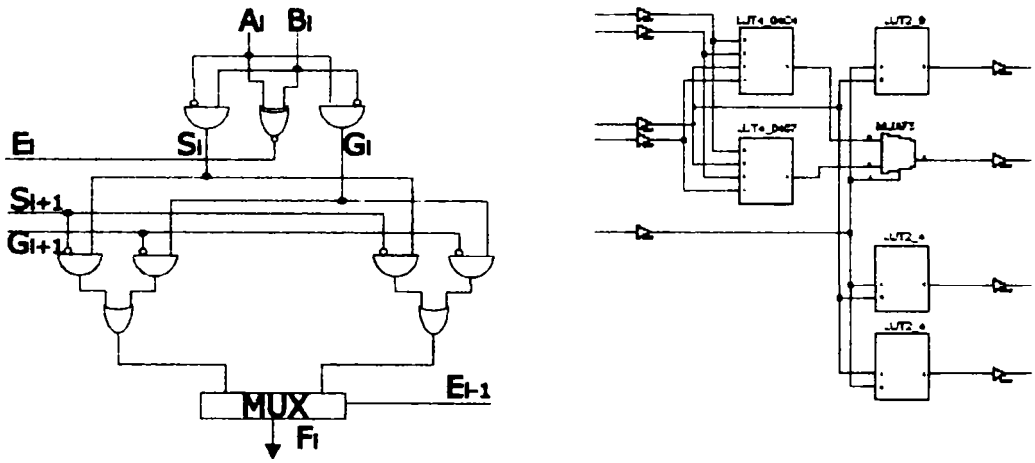



Figure 2.4 - Encoding Cell in LZP (Logic [16] and Virtex-4 FPGA Technology Schematic)

Also in 2.5 is presented a leading zero detector tree for sixteen bits. The entire structure requires $\lceil log_2 N \rceil - 1$ levels (where N is the number of bits). The delay of the first level is 2 LL, while the delay of one tree node is 1 LL.

Table 2.2 - Four Bits Truth Table for LZD [73]

| Pattern | Position bits | Valid |
|---|---|---|
| 1xxx | 00 | 1 |
| 01xx | 01 | 1 |
| 001x | 10 | 1 |
| 0001 | 11 | 1 |
| 0000 | xx | 0 |

The encoding provided by the encoding module does not take account of the carry-in bits for each position which may appear at the subtraction of the two

operands. This may lead to an inaccurate prediction (the inaccuracy being of maximum one position). Thus, in order to obtain an aligned mantissa a correction is needed. There are, however, leading zero predictors which predict exactly the amount of leading zeros, like the one presented in [35]. However, the latency and the area overhead of these types of LZP are great compared to a LZP with inaccurate results [85].

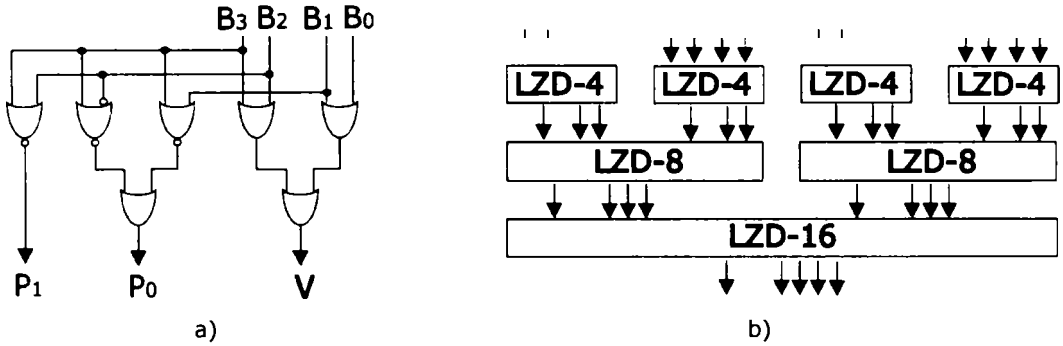



Figure 2.5 – Leading Zero Detection – a) For Four Bits Group b) Tree Detector for 16 Bits[73]

Three ways for performing correction do exist [16][63]. In the first method, the normalization shifting is performed using the determined amount of leading zeros. After the normalization, in case of a leading zero (maximum one) another left shift is performed (Fig. 2.6 – a). This method has the advantage of a low cost, requiring only a single one position left shifter, besides the encoding and the detection tree. The main drawback is that introduces another one position left shifter in the critical path of the floating point adder.

The second method is based on the selection of the appropriate carry from the entire carry chain based on the result of the LZD [43][80]. Based on the selected carry, the correction is made (Fig 2.6 – b). This solution has a higher cost compared to the correction shifting, because of the carry selection. Regarding the added latency to the overall operation, this type of LZP has to wait for the carry chain from the adder. Thus, it cannot provide the correct result at the end of the addition/subtraction. However, the solution for this is to use the most significant bits of the predicted result for course shifting during the correction, while the fine shifting (which require the least significant bits of the predicted result and which may be affected by correction) to be performed at the end of the operation. This way, the latency added to the floating point addition is smaller compared to the correction shifter.

The third method is presented in [16] and is based a parallel detection tree. This detection tree detects whether a correction should be made based on the set of patterns which can be generated in the encoding module (Fig 2.6 – c). The use of another detection tree which runs in parallel is to reduce both latency and cost (otherwise, an exact LZP would be preferred), because analyzing small sets of patterns in parallel require simpler logic. Furthermore, this detection tree is separated into a detection tree for the positive result case and one for the negative result case. This solution is also used in [65]. The main drawback is represented by a doubling in cost. However, the introduced delay in the floating point addition of this module is negligible.

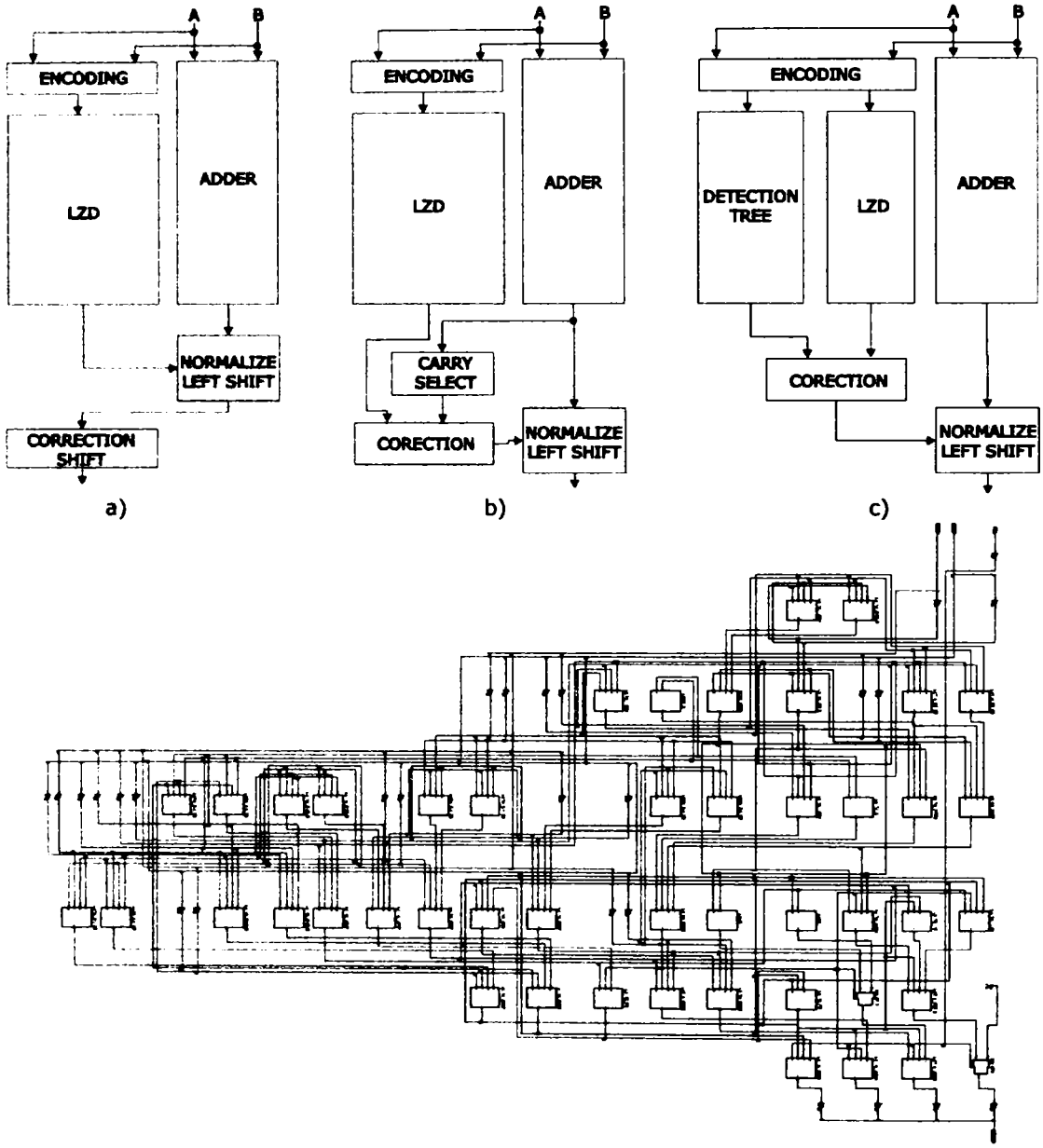



Figure 2.6 – Correction Strategies for LZP [16] a) Correction Shift b) Carry Selection c) Concurrent Position Correction d) Technology Schematic of a LZP Presented in a) Obtained with XST

The choice of the correction strategy influences the area and the performance of the overall floating point addition. Thus, a post correction shift introduces the lowest area overhead. As explained in [16], the delay introduced by this solution seems to be the greatest. However, different floating point adder's designs use this approach, considering that the introduced delay is acceptable for the overall performance of the floating point addition.

### 2.2.2.2 Compound Adder

As presented in Section 2.2.1, three steps in the basic require mantissa length adders: the mantissa addition, the result conversion (two's complementation) and the rounding step. The result conversion and the rounding step require only 1 ulp addition. By swapping the elements based on the exponents' difference the result conversion can be made only in the case of equal exponents. In this case, there is no need for rounding. Thus, the two steps are mutually exclusive. Furthermore, the two steps can be reduced to a selection (a multiplexer) if the result of the mantissa addition ($A+B$) and the incremented result ($A+B+1$) can be obtained in parallel [78]. This can be achieved using either two integer adder which run in parallel or a compound adder [17][98].

The compound adder is an adder which computes the sum of two numbers ($A+B$) and the increment sum ($A+B+1$). This adder is very advantageous to implement using the parallel prefix tree type integer adders, like the Brent-Kung adder [15], or the ones described in [50]. These adders are based on the computation of the carry chain based on the $g_i = a_i b_i$ (generation bits) and $p_i = a_i \oplus b_i$ (propagation bits). Generation and propagation bits for a group of operand bits ($G_{ij}$ and $P_{ij}$) are generated in tree based structures. The carry according to the $i$-th position is determined according to the following equation: $c_i = G_{0(i-1)} + P_{0(i-1)}c_0$ where $c_0$ represents the carry-in of the whole adder. The depth of the tree structures used to compute the carry chain has a logarithmical depth [15][50]. In [98] it was shown that if $c_0$ is 0 then $c_i = G_{0(i-1)}$ and if $c_0$ is equal to 1 then $c_i = G_{0(i-1)} + P_{0(i-1)}$. Thus it can be computed with only a small increase in hardware A+B (the case $c_0$ when is 0) and A+B+1 (the case $c_0$ when is 1). The difference in latency between the two results is only of 1 LL.

Table 2.3 – Obtaining Sum, Sum+1, Sum+2 Using Half Adders and Compound Adder [17]

| LSB | Compound Adder Result (CAR) | Incremented CAR (ICAR) | Sum Selection | Sum+1 Selection | Sum+2 Selection |
|---|---|---|---|---|---|
| 0 | A+B+1 | A+B+2 | CAR,0 | CAR,1 | ICAR,0 |
| 1 | A+B | A+B+1 | CAR,1 | ICAR,0 | ICAR,1 |

Therefore, using a compound adder, rounding and complementation can be reduced to only a simple selection, as follows [17][79]:

- In case of effective subtraction for operands with equal exponents, the result may be negative of positive. In case of a negative result, the sum is selected (which will be later bit-inverted), while in the case of a positive result, the incremented sum is selected.
- In case of rounding to nearest even, the sum and the incremented sum are needed for both effective addition (with both the cases when the result overflows and does not overflows) and effective subtraction (with both cases when the result is denormalized and normalized).

- In case of rounding towards infinity, the sum and incremented sum are needed for the case when the result does not overflow. When the result overflows, the *A+B+2* is needed. In order to solve this problem two solutions have been developed: using two parallel compound adders (with a major increase in cost) or using a line of half adders before the compound addition. This line of half adders allows computing either *A+B* and *A+B+1*, either *A+B+1* and *A+B+2* (see Table 2.3), adding 1 LL in the critical path of the adder.

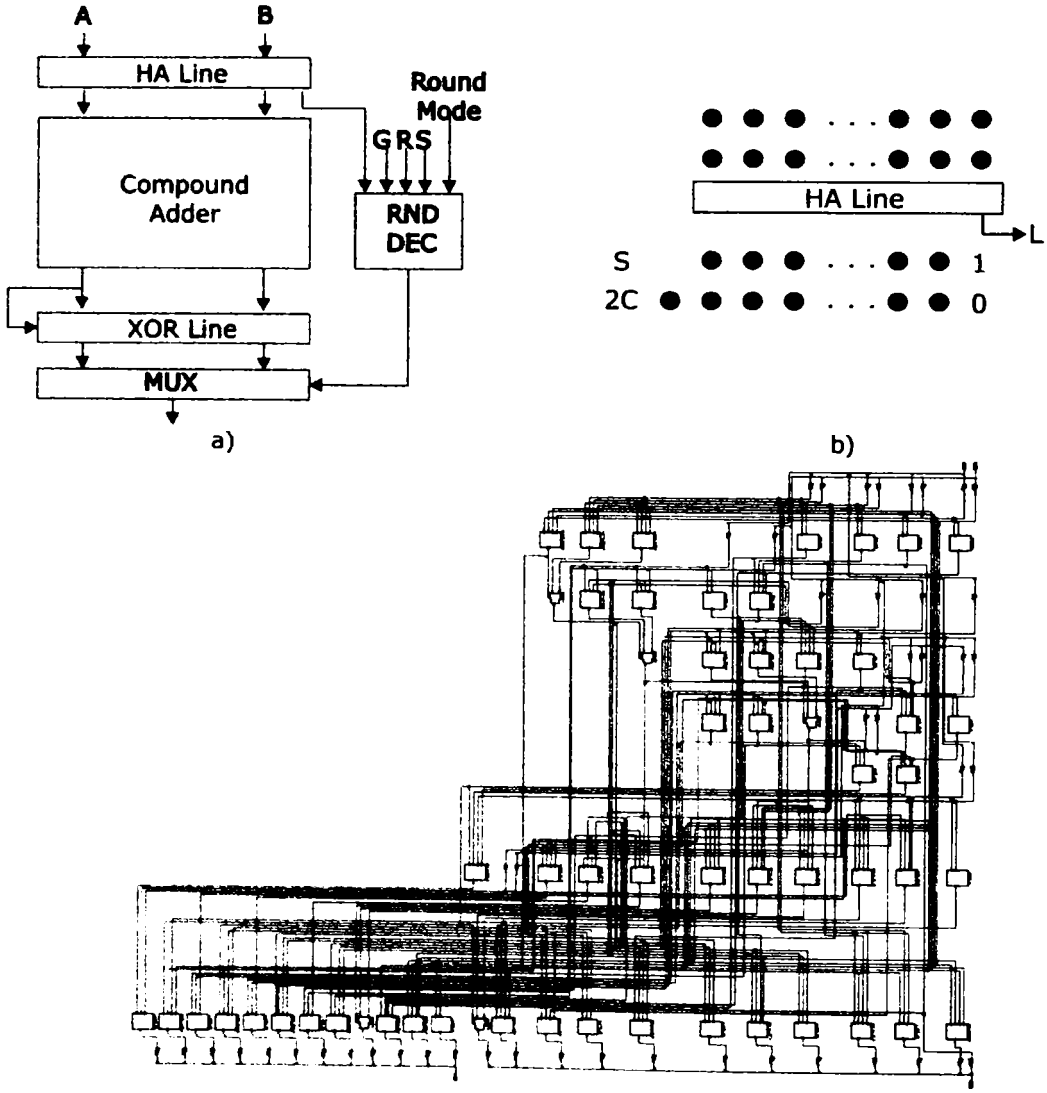



Figure 2.7 – Addition, Complementation and Rounding Module (a) Preparing Operands for Sum, Sum+1, Sum+2 Using Half Adders (b) [17] Technology Schematic of 12-Bit Compound Adder Obtained with XST (c)

The module obtained for is presented in Fig. 2.7. Thus, using a compound adder, a major reduction in cost (because a single compound adder with

multiplexers are used instead of three large carry propagate adders) and latency is obtained.

### 2.2.3 Double Path Adder

Further improvements of the floating point addition performance can be made taking account the following assumptions [31] [65][66]:

- When the exponents' difference is greater than 1, a massive (more than 1 position) right shift mantissa alignment step is needed. However, in case of an effective subtraction, there can be a maximum one position normalization shift.
- When the exponents' difference is o or 1, only one right shift is needed for normalization. However, in case of an effective subtraction, cancellation of the result possible so a massive normalization left shift is needed.

As it can be seen above, a massive alignment shift and a massive normalization shift are mutually exclusive. Thus, the two cases presented above can be separated in two different computational paths. This way, a double path adder is obtained. This design for floating point addition was first proposed by Farmwald ([31]) and became the design with the highest performance, many versions of this type of adder being developed [12][37][44][63][65][66][90][89].

The path which computes the mantissa result when the exponents' difference is greater than 1 is called the FAR path [31]. This computational path is characterized by the right barrel shift, which is used to align the mantissa before addition, and the more complex rounding logic. The major latency modules of this computation path are the right barrel shifter and the compound adder.

The path which computes the mantissa result when the exponents' difference is 0 or 1 is called the CLOSE path [31]. This path is characterized by the leading zero predictor and the left barrel shifter, which is used to normalize the result. The major latency modules of this computation path are the compound adder (which runs in parallel with the leading zero predictor) and the left barrel shifter.

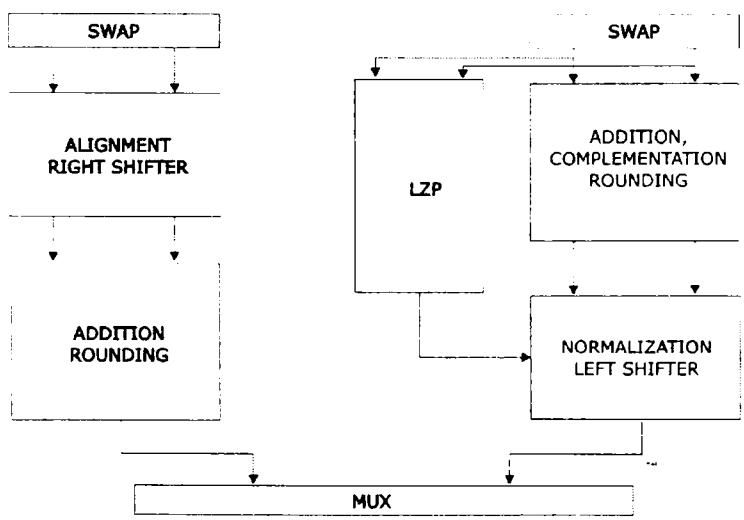



Figure 2.8 – Block Architecture of Mantissa Data Path of the Double Path Adder [26]

As it can be observed from above, each computational path has only two major latency modules, which are placed on the critical path of the adder (Fig. 2.8). The single path adder (presented in section 2.2.2) contains the three large latency modules on the critical path. Therefore, an evident increase in the performance of the floating point addition is achieved. The cost of the double path adder is greater compared to a single path adder, mainly because it has two compound adders compared with one used in single path adders [44][66][90].

The computational flow in the double path adder is the following: the two mantissas are served as inputs in each computational path, two results being computed; in parallel the path selection condition is also computed; in the last stage the correct result is selected.

Table 2.4 - Examples of Floating Operations Performed on FAR and CLOSE Paths

| Condition | Example | IEEE Representation | Path |
|---|---|---|---|
| Exponents Difference | 0.625±0.375 | $(-1)^0 * 2^{-1+127} * 1.25 \pm (-1)^0 * 2^{-2+127} * 1.5$ | CLOSE |
| | 2.125±0.375 | $(-1)^0 * 2^{0+127} * 1.0625 \pm (-1)^0 * 2^{-2+127} * 1.5$ | FAR |
| Exponents Difference+ CLOSE Only for Subtractions | 0.625-0.375 | $(-1)^0 * 2^{-1+127} * 1.25 - (-1)^0 * 2^{-2+127} * 1.5$ | CLOSE |
| | 0.625+0.375 | $(-1)^0 * 2^{-1+127} * 1.25 + (-1)^0 * 2^{-2+127} * 1.5$ | FAR |
| | 2.125±0.375 | $(-1)^0 * 2^{0+127} * 1.0625 \pm (-1)^0 * 2^{-2+127} * 1.5$ | FAR |
| Exponents Difference With No Rounding In CLOSE Path | 0.875-0.375 | $(-1)^0 * 2^{-1+127} * 1.75 - (-1)^0 * 2^{-2+127} * 1.5$ | FAR |
| | 0.625-0.375 | $(-1)^0 * 2^{-1+127} * 1.25 - (-1)^0 * 2^{-2+127} * 1.5$ | CLOSE |
| | 0.625+0.375<br>2.125±0.375 | $(-1)^0 * 2^{-1+127} * 1.25 + (-1)^0 * 2^{-2+127} * 1.5$<br>$(-1)^0 * 2^{0+127} * 1.0625 \pm (-1)^0 * 2^{-2+127} * 1.5$ | FAR |

An important feature of the double path adder is represented by the path selection condition. The first condition used in [31] was the exponents' difference. This condition was also used in the variable latency adder [64][66]. Another condition for path selection is used in [44][65] and involves exponents' difference and the effective operation. The CLOSE path is destined only for effective subtractions when the exponents' difference is 0 or 1, while on the FAR path are performed all the effective additions and the subtractions when the exponents' difference is greater than 1. This is possible because the effective additions do not result in a number with leading zeros, thus the normalization left shift is not required. Because on the CLOSE path are performed only subtractions, there is no possibility for an overflow in the mantissas addition, thus there is no need for a 1-position right shift in this case (decreasing thus the latency of this path). To further increase the performance of the CLOSE path, in [12][37][63][89][90] another path selection criterion is used. The CLOSE path will perform only the effective subtractions with exponents' difference 0 or effective subtraction with exponents' difference equal to 1 which will result with leading zeros. This way, the rounding step is removed from the CLOSE path. Examples of different additions and subtractions and their execution path are presented in Table 2.3.

### 2.2.3.1 Adelaide1999 Adder

This floating point adder has been designed at the University of Adelaide and is presented in [12]. The CLOSE path will perform all the subtractions when the exponents' difference is 0 and the subtractions when the exponents' difference is 1, but which will result in denormalized number (it has leading zeros).

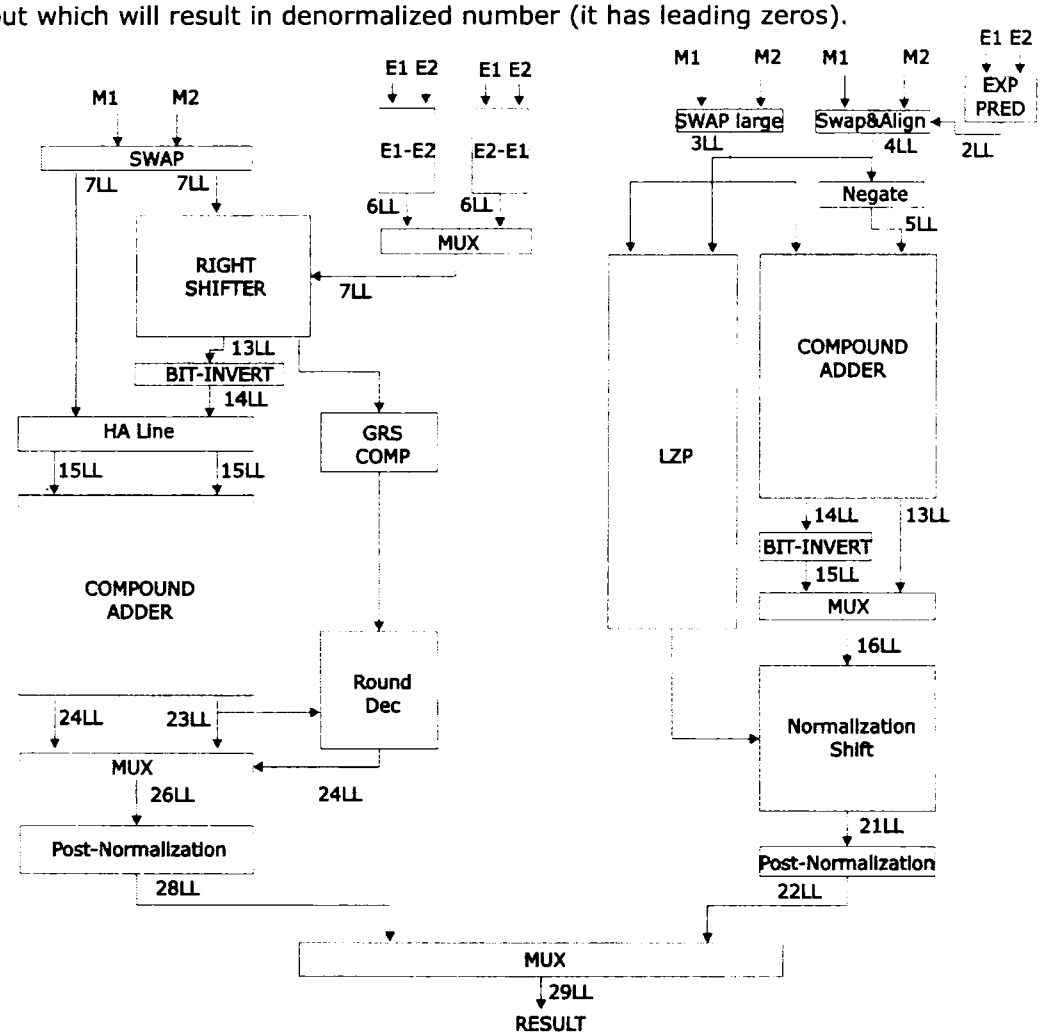



Figure 2.9 – Structure of the Adelaide Adder [12]

This is determined by an examination of the MSBs of the mantissa in the unpacked form (the mantissa without the hidden 1). Thus, there is no need for rounding in the CLOSE path. The leading zero correction is done using a post-normalization shifter.

The FAR path will compute all the operations when the exponents' difference greater than 1, all the effective additions and the subtractions which will not result in leading zeros. Two exponents' difference subtractor circuits are used (one for *E1-E2* and the other for *E2-E1*), thus there is no need for complementation of the exponents' difference in case it is negative.

### 2.2.3.2 SUN1998 Adder

This adder represent a SUN Microsystems patent [37] and was also presented in [89]. This adder is similar to the Adelaide adder regarding the path selection criterion and the CLOSE path.

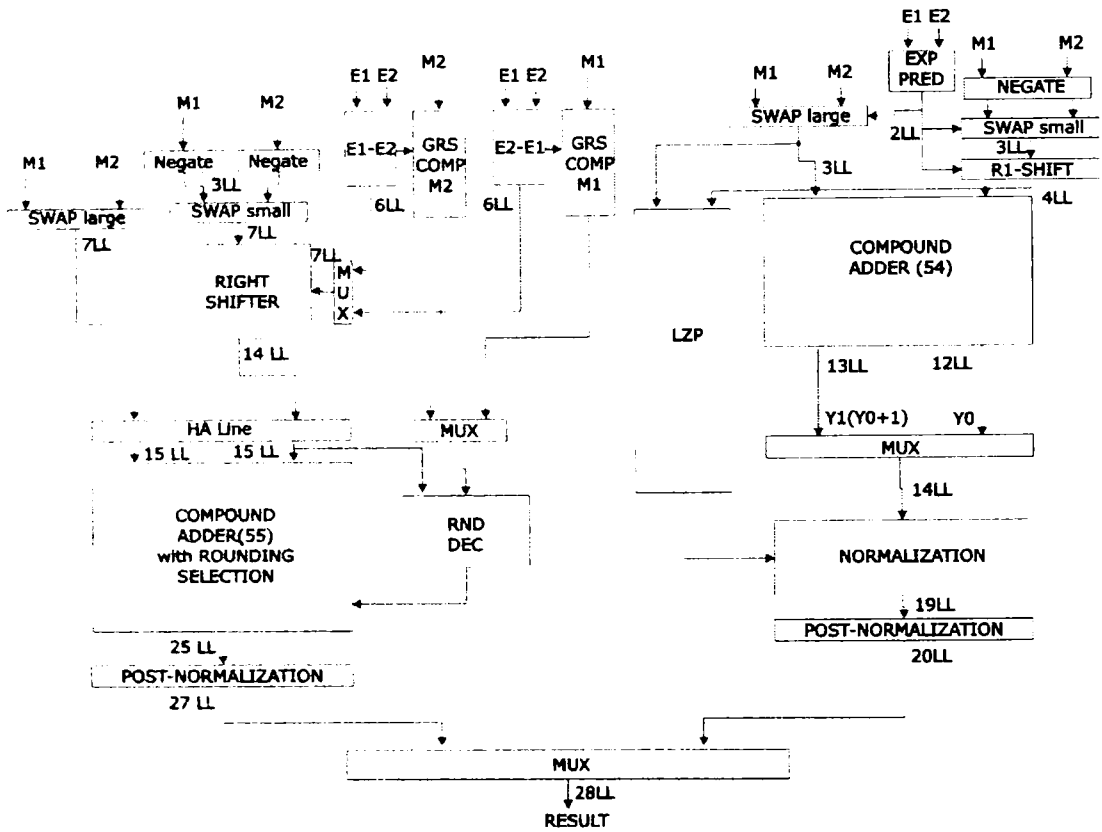



Figure 2.10- Structure of the SUN1998 Adder[37]

The main contributions of this type of adder are implemented in the FAR path. One contribution is represented by the way of computing the rounding needed bits (G, R and S). Two series of rounding bits are computed in parallel, for each mantissa, before any alignment shift. Two exponents' differences are computed, the two sets of rounding are obtained based on these results [89]. The correct series is chosen based on the sign of exponents' difference. Another contribution of this design is to use a special dedicated carry propagate adder with selection incorporated. The reason for using such adder is to easily split it for pipeline architectures, as this type of adder contains three pipeline stages (which are very fast).

Although it uses a similar path selection criterion like the Adelaide adder, it is however implemented rather complicated, as it performs a more complicated analysis of the two mantissas.

### 2.2.3.3 AMD2000 Adder

This adder was designed by Stuart Oberman and represents an AMD patent [65] and is also described in [89]. This adder is quite different from the other two described in previous sections, mainly due to the path selection criterion. The CLOSE path is used for all effective subtractions when the exponents' difference is 0 or 1. Therefore, a rounding unit is needed in the CLOSE path. Thus, an increase in latency can be observed in the CLOSE path compared to the other two designs.

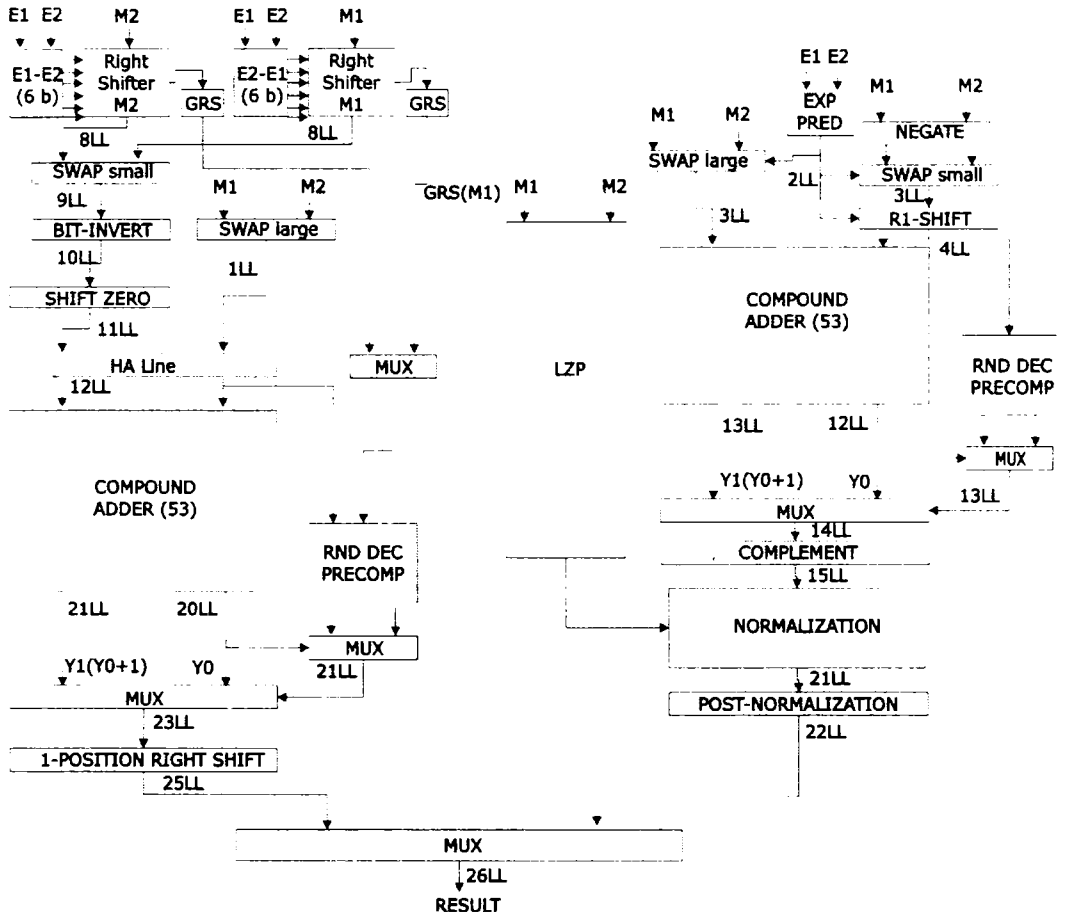



Figure 2.11 – Block Structure of AMD Adder

The main contributions of this design rely are located on the FAR data path, especially regarding the alignment shifting. The alignment shift in two steps: the first step involves shifting if the exponents' difference is smaller or equal to 63 (because the mantissa is represented on 53 bits – including the hidden one) [89]. This shift is performed for both cases when the exponents' difference is positive or negative for both operands. The shifting on each weight is performed after each bit of the exponents' difference is available (thus, not waiting for the entire subtraction). The second step for shifting is performed after the exponents' difference is available. In case that exponents' difference is greater than 63, the smaller mantissa will be arithmetically right shifted. This step is performed by the

SHIFT ZERO block (actually the number is shifted with the sign and not with 0). This way a decrease in the latency of the FAR data path can be obtained.

### 2.2.3.4 Seidel-Even Adder

This adder is presented in [89][90] and implies several techniques to decrease the latency. The path selection criterion is similar to the one used in [12][37], and the CLOSE path is thus very similar to the SUN and Adelaide adders.

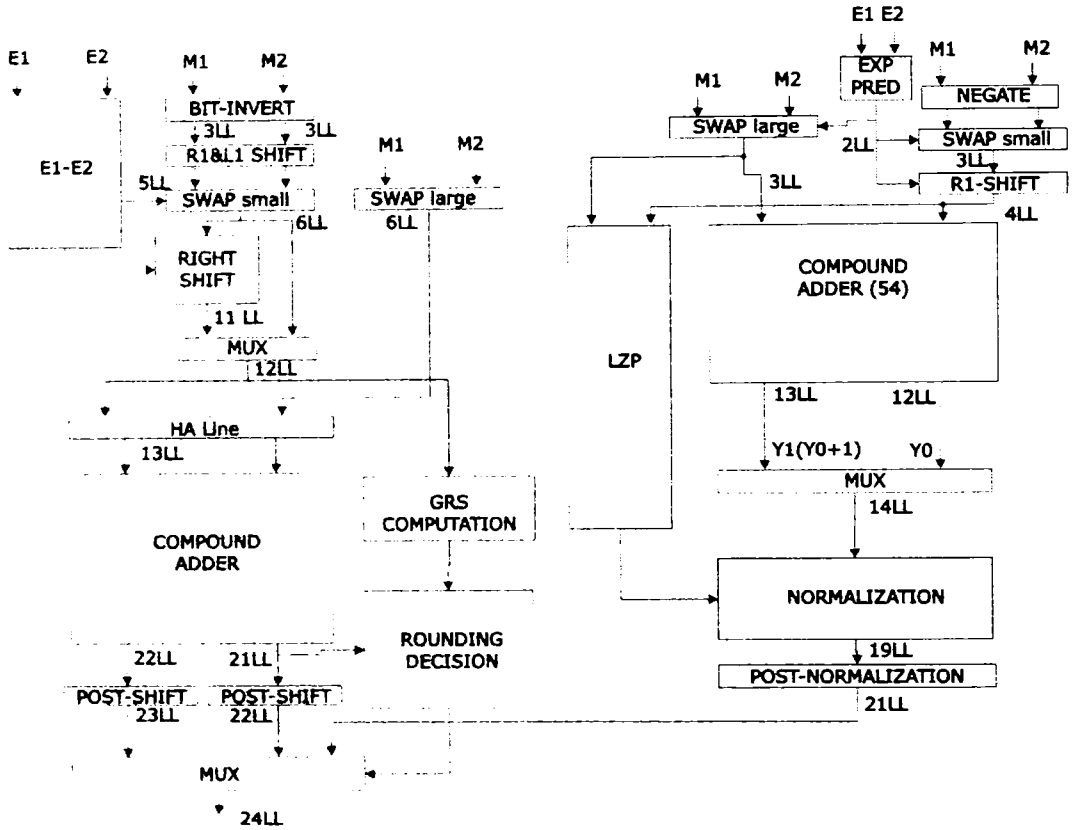



Figure 2.12 – Block Structure of the Seidel-Even Adder [90]

Regarding the FAR path, several optimization techniques are used. Because only one adder is used for exponents' difference (thus needing a two's complement) a speculative one position left shift is performed before exponents' difference is available (because exponent's difference will come in one's complement form and not two's). Also, the massive right shift is performed (for the case when the exponents' difference is greater than 63) in a speculative manner. The shifting for the case when exponents difference is smaller than 64 is performed in a similar way to the one performed at the AMD adder (as each bit of the exponents' difference is available) [90]. Another optimization is carried out in the rounding and post-normalization shift. The rounding decision is computed in two paths, for the case of no overflow and for overflow of the addition result. The post-normalization shift is performed for each of the two results of the compound adder, before the selection

of the appropriate result. The reason is that there is enough time for a one position right shift until the rounding decision is available.

The last optimization technique used in this adder is implemented in the result selection multiplexer. The result in the CLOSE path and one result of the compound adder (the sum) are available before the other result of the compound adder. Thus, a selection between the two is performed. The result of the operation is obtained by selecting between the shifted incremented sum of the compound adder and the selected result in the previous multiplexer.

### 2.2.3.4 Variable Latency Adder (Stanford)

The variable latency adder was proposed in [64][66], as part of the SNAP project developed at Stanford University. This adder exploits the fact that not all operations are ready at the same time.

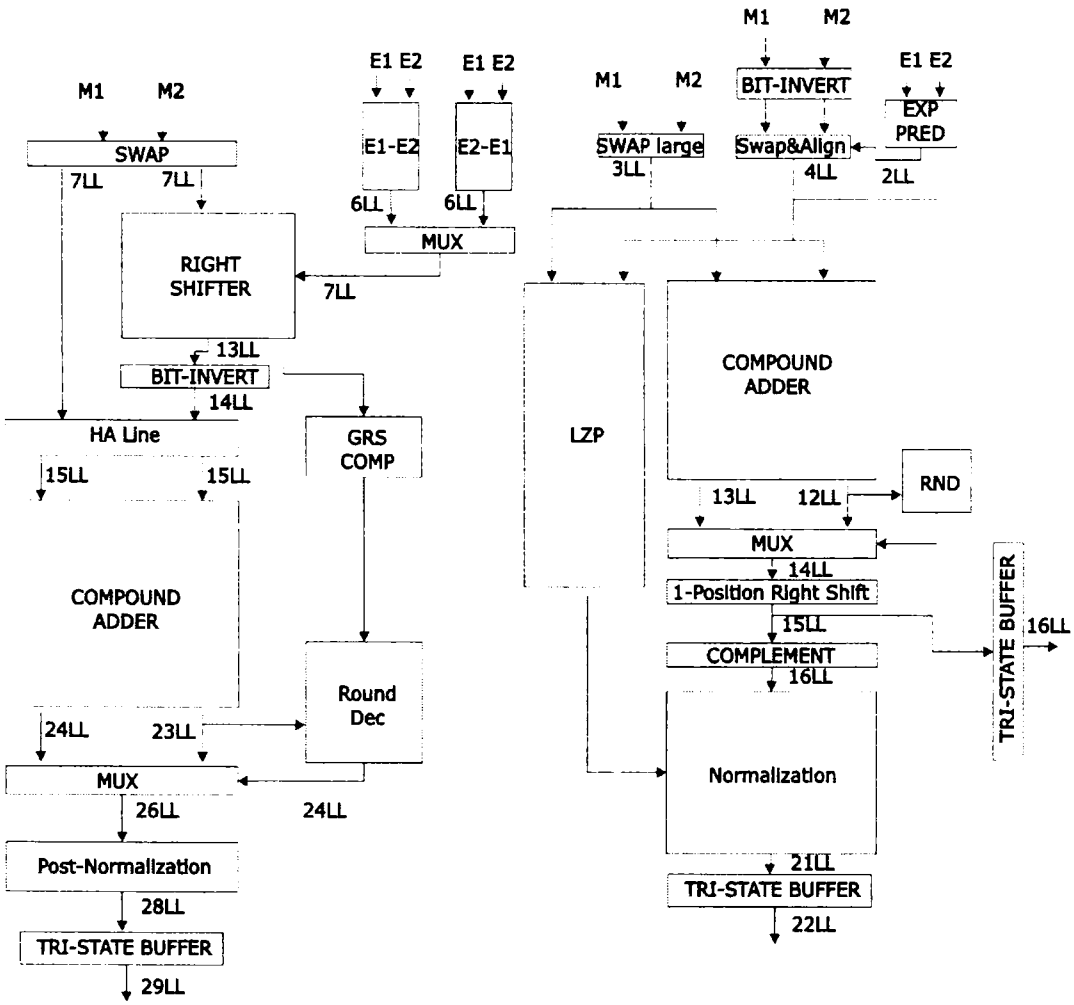



Figure 2.13 – Variable Latency Adder Structure (Stanford)

The path selection criterion is the exponents' difference. Thus, the effective additions when the exponents' difference is 0 or 1 are performed on the CLOSE path. In this case, the operations which must be performed are the one position alignment shift, addition, rounding and the normalization shifting (in case of overflow), not needing the normalization left shifting. After these operations, the result is available and the operation is ended. Furthermore, in case of effective subtraction when exponents' difference is smaller or equal to 1, the result is available earlier compared to an operation that is performed on the FAR path [64][66].

Tri-state buffers are place at each location when an operation might be over. Thus, the floating point addition may be over earlier or later, depending on the execution path and on the effective operation.

### 2.2.3.5 Comparisons between Double Path Adder Designs

In the following table a comparison between the five presented double path adders design is realized. The adders are compared in terms of latency (expressed in logic levels), path selection criterion, operations in CLOSE path, rounding in CLOSE path.

Table 2.5 - Double Path Adders Comparison

| Adder | Latency | Path Selection Criterion | Rounding in CLOSE | Operations in CLOSE |
|---|---|---|---|---|
| Adelaide'99 | 29 | Exponents difference+ Effective operation+ Leading zeros in CLOSE | No | Effective subtraction |
| Sun'98 | 28 | Exponents difference+ Effective operation+ Leading zeros in CLOSE | No | Effective subtraction |
| AMD 2000 | 26 | Exponents difference+ Effective operation | Yes | Effective subtraction |
| Seidel-Even | 24 | Exponents subtraction+ Effective operation+ Leading zeros in CLOSE | No | Effective subtraction |
| Variable Latency | 29 | Exponents difference | Yes | Effective subtraction + Addition |

Performing the addition in a speculative manner on two computational paths, an increase of the performance of the floating addition is obtained. This way, high performance for the most frequent floating point operation is provided. However, this increase in performance came at almost a double cost and power consumption compared to the single path floating point adders.

## 2.3 Proposed Adder

### 2.3.1 Interval Addition

As it can be observed in Section 2.1, an interval addition/subtraction requires two floating point operations. These two floating point operations can be executed either in parallel using two floating point adders, or sequentially using one floating point adder. In both cases, in order to obtain better performance, a double path floating point adder is recommended in both cases.

The floating point adders presented are designed to increase the performance of only a single floating point addition. Because of this features, one path is virtually not used when executing one operation (because it will produce the wrong result). For example, in case of an addition when the exponents' difference is greater than 1, the CLOSE path is of no interest for the result of operation.

The proposed adder tries to exploits the parallel structure of the double path adder, therefore, performing the two floating point operations needed for an interval addition/subtraction simultaneous, one on the CLOSE path and the other one on the FAR path. In order to perform two floating point operations, the modules which compute the exponents and the signs must be duplicated [3][5].

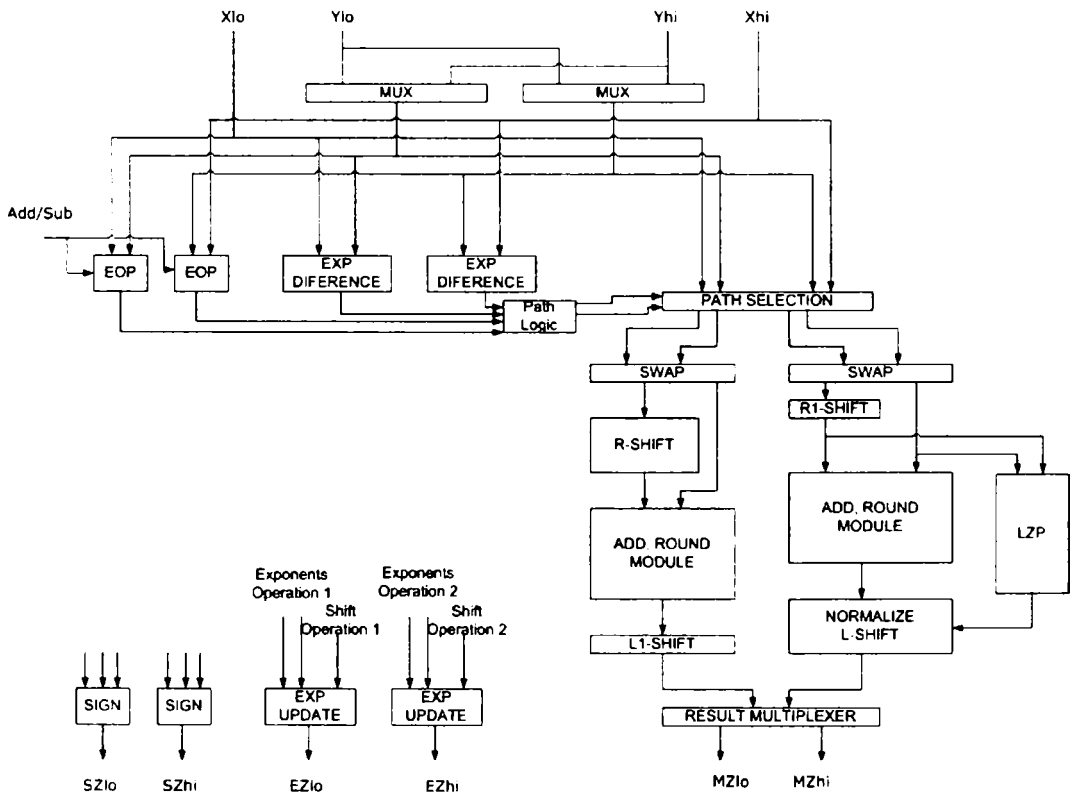
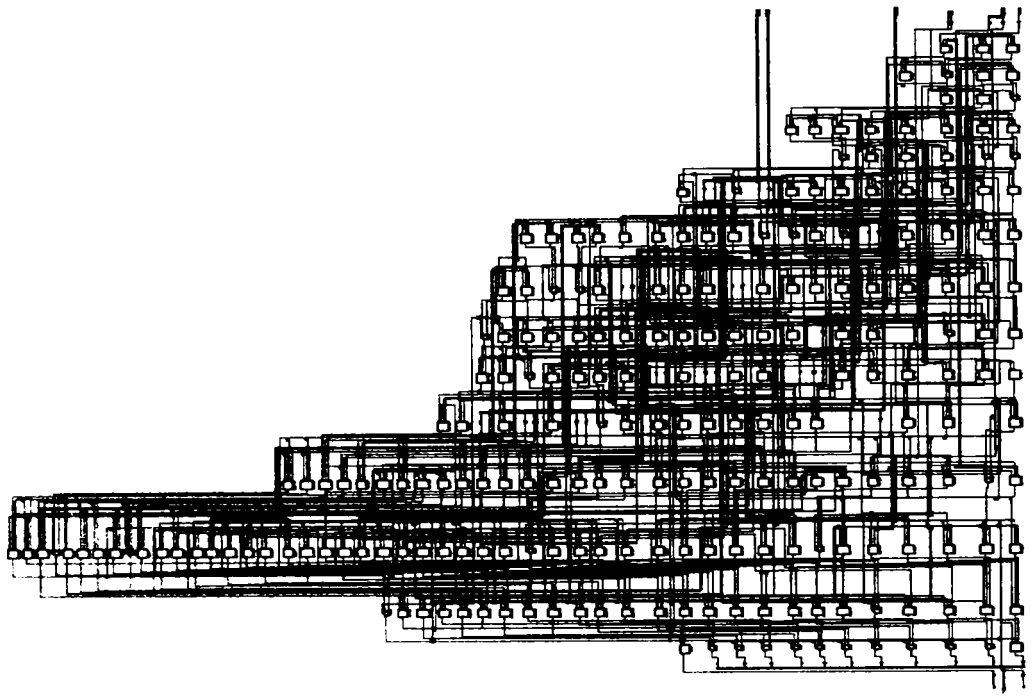
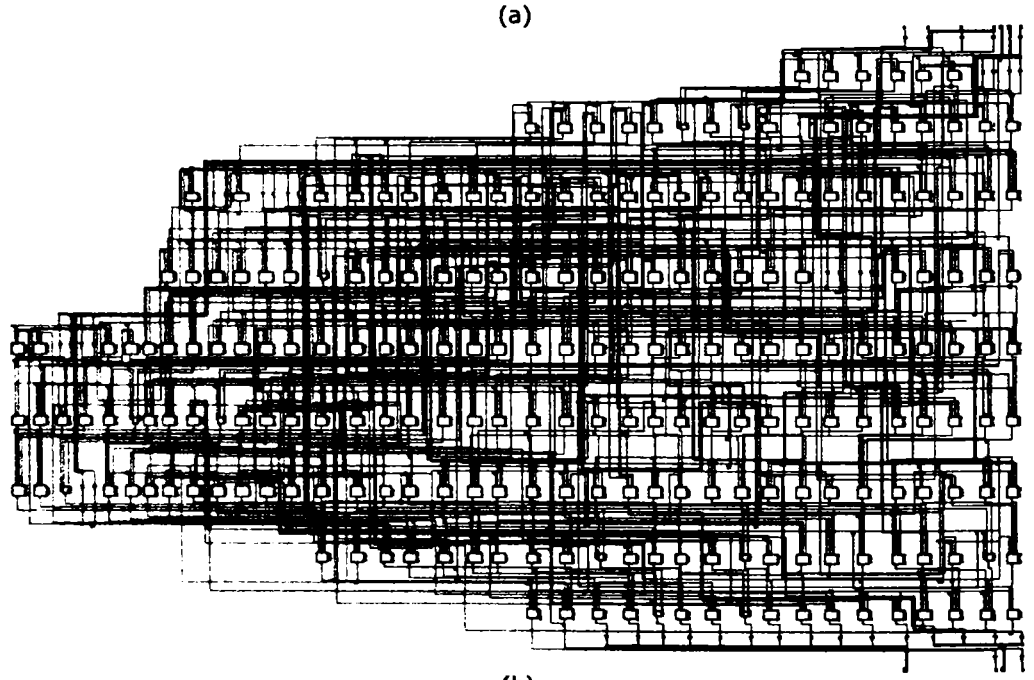



Figure 2.14 – Block Structure of Proposed Interval Adder [3][5]



(a)



(b)

Figure 2.15 – Technology Schematic Obtained with XST for FAR path (a) and CLOSE path (b)

The overall architecture is presented in Fig 2.14. Several changes do appear compared to other double path adder designs:

1. In the proposed adder, the corresponding path for each operation must be known before the splitting into two paths. The reason for this is that two pairs of operands, along with the appropriate effective operation and the appropriate rounding mode, must be transmitted on the corresponding path. Therefore, a set of multiplexers have to be placed before each path. The signals which command the set of multiplexers are generated according to the path selection criterion.

2. Two (pairs of) exponents' differences have to be computed, because two selection criterions must be computed. Another important feature is that the exponents' differences are computed before any mantissa computing in either the FAR path or CLOSE path. This is due to the fact that there is no speculative operation like in other floating point adder's designs. Because exponents' difference are computed before any processing of mantissa, no exponent difference prediction is required in the CLOSE path, and no technique for improving the alignment shift in the FAR path (like the ones in AMD adder and Seidel-Even adder) can be applied.

3. The logic for the path selection criterion is founded on the critical path of the floating point adder. Therefore, a simpler path selection criterion means a smaller latency. However, a more complex path selection criterion increases the possibility that two floating point operations can be performed simultaneously.

The path selection criterion used in [3] is the exponents' difference. Using this criterion, on the FAR path will be performed the operations which have exponents' difference greater than 1, while on the CLOSE path will be performed the operation which have exponents' difference smaller or equal to 1. If we use a criterion based on the exponents' difference and the effective operation, as presented in [5], then on the FAR path will be executed all the effective additions and the subtraction when the exponents' difference are greater than 1, while on the CLOSE path will be executed all the additions and subtraction when the exponents' difference is smaller or equal to 1. In this case, the logic for path selection is more complex. In both cases the FAR path and the CLOSE path have the same structure.

The CLOSE path's structure enables the execution of both additions and subtractions when the exponents' difference is 0 or 1. The swap and alignment are based on the exact exponents' difference (because it is available) and not on an prediction based on the least significant bits as it is done in other double path adder's design. Because effective additions may be performed on the CLOSE path, 1-position normalization right shifter is needed, because an overflow may occur. Furthermore, rounding logic is also needed. Thus, the CLOSE path will have an increased latency compared to Adelaide, SUN or Seidel-Even adders.

On the FAR path can be executed all the operations when the exponents' difference is greater than 1 and the effective additions when the exponents' difference is 0 or 1. The alignment shift is performed on the exact difference of the exponents (which is available). This is the single major difference between the proposed adder's FAR path and other adder designs, like the AMD, SUN or Seidel-Even adders. The detailed structure of the mantissa data path is presented in Fig 2.15, while in Fig 2.16 the technology schematic for the FAR and CLOSE path of the proposed adder is presented.

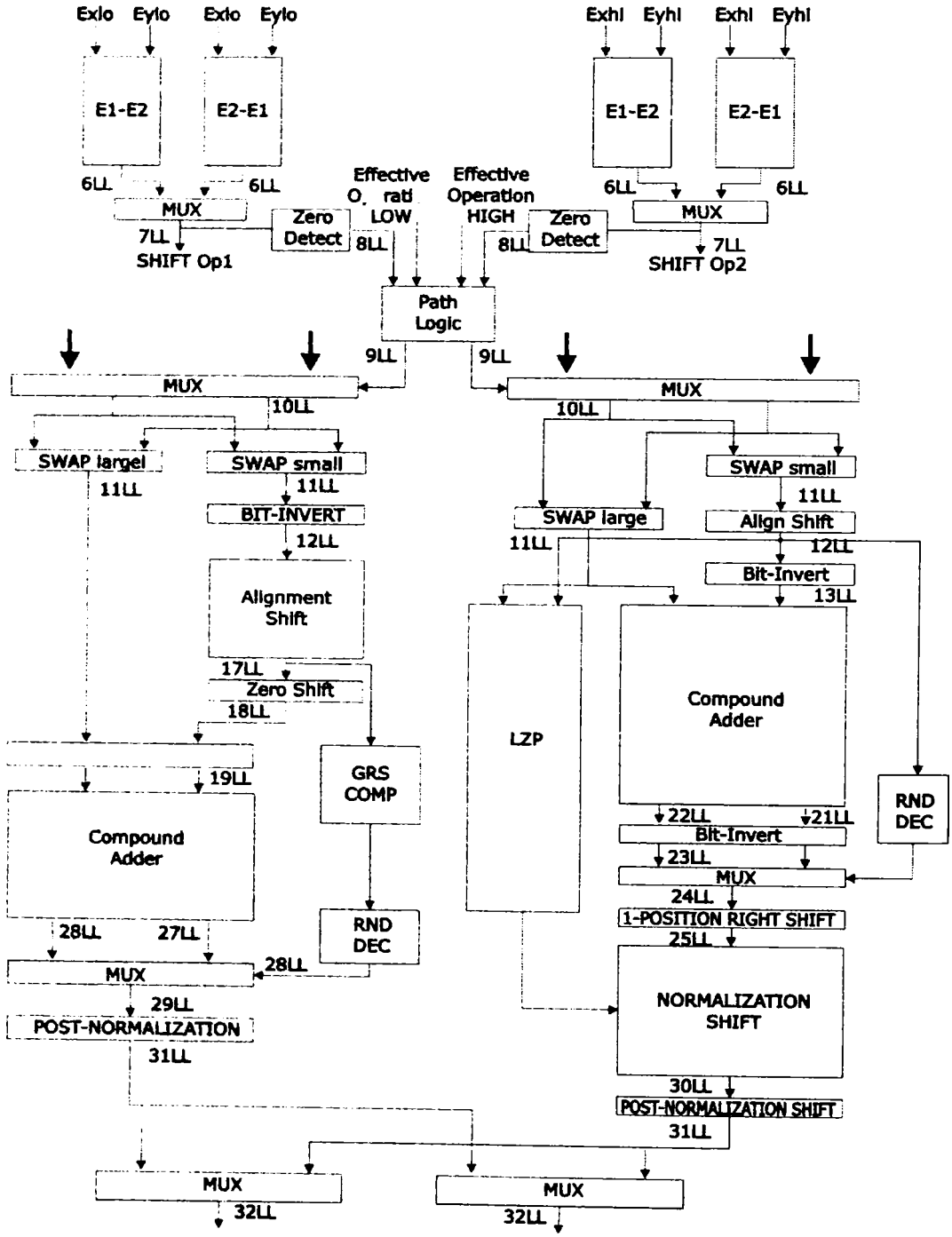



Figure 2.16 – Detailed Structure of Mantissa Data Path in Proposed Adder [5]

Table 2.6 Examples of Favorable Cases of Interval Addition
(D stands for exponents' difference)

| $X_{lo}$ | $Y_{lo}$ | Operation | Path | $X_{hi}$ | $Y_{hi}$ | Operation | Path |
|---|---|---|---|---|---|---|---|
| 0.625 | +0.25<br>-0.375 | D=1 | CLOSE | 2.125 | +0.375<br>-0.25 | D=3 | FAR |
| 0.625 | 0.375 | Effective Addition<br>D=1 | CLOSE | 0.75 | 0.5 | Effective Addition<br>D=0 | FAR |
| 0.625 | -0.375 | Effective Subtraction<br>D=1 | CLOSE | 0.75 | 0.25 | Effective Addition<br>D=1 | FAR |

Table 2.7 Examples of Unfavorable Cases of Interval Addition
(D stands for exponents' difference)

| $X_{lo}$ | $Y_{lo}$ | Operation | Path | $X_{hi}$ | $Y_{hi}$ | Operation | Path |
|---|---|---|---|---|---|---|---|
| 0.625 | -0.375 | Effective<br>Subtraction<br>D=1 | CLOSE | 0.75 | -0.25 | Effective<br>Subtraction<br>D=1 | CLOSE |
| 1.75 | 0.375 | D=2 | FAR | 2.125 | 0.5 | D=2 | FAR |

Using the proposed adder, the two floating point operations needed for an interval addition/subtraction can be performed simultaneously, like in the solution based on two floating point adders, or can be performed sequentially, like in the solution based on a single floating point adder. Therefore, we have favorable cases and unfavorable cases. The probability of occurrence of a favorable case depends on the selection criterion, the one which is based on both exponents' difference and effective operation having a greater probability. In tables 2.6 and 2.7 some examples of favorable and unfavorable cases for the exponents' difference and effective operation based path selection criterions are presented. In case an unfavorable does occur, one operation will have priority (for example the operation for the lower end of the result interval), while the other one will wait. This unfavorable case can be signaled to the control unit by a dedicated flag.

### 2.3.2 Increasing Throughput of Conventional Floating Point Addition

The adder architecture proposed in Fig 2.14 can also be used for increasing the performance of conventional floating point addition by increase in throughput realized by this type of adder, as it is presented in [5]. This can be achieved by implementing rounding towards nearest even. Two floating point additions/subtractions can be performed simultaneously on the proposed architecture. The proposed adder is suitable to for superscalar, multi-threaded (which require two or three functional units) or can be used in dynamic scheduling schemes (like the Tomasulo's scheme)[42]. The difference compared to the structure depicted in Fig 2.14 is lack of multiplexers before the adder itself. Both path selection criterions (only exponents' difference and exponents' difference plus effective operation) may be used, but the second is more appropriate because it will increase the probability of performing two floating point operations simultaneously.

## 2.4 Evaluation

### 2.4.1 Cost Evaluation

In order to perform a cost evaluation we used an independent technology metric, the gate count. This may not be the most conclusive metric for VLSI technology, as it does not take into account the wirings. However, this metric was also used in [88]. Table 2.9 presents the gate of the five described adders and the proposed adder.

Table 2.9 Gate Count for Floating Point Adders
(Double Precision Format)

| Adder | FAR | CLOSE | Exponent, Sign &Path Select | Total |
|---|---|---|---|---|
| Adelaide | 1753 | 2298 | 349 | 4500 |
| SUN | 1828 | 2302 | 456 | 4586 |
| AMD | 1985 | 2305 | 453 | 4743 |
| Variable Latency | 1806 | 2510 | 392 | 4708 |
| Seidel-Even | 1912 | 2298 | 381 | 4591 |
| Proposed | 1753 | 2457 | 1122 | 5332 |

As it can be seen from table 2.6, an increase of about 17% for double precision of the proposed design compared to other floating point adder's designs can be observed.

Regarding interval addition, the proposed adder has an increase of about 17% with respect to the interval adder based on a single floating point adder. However, the proposed adder's gate count is about 57% of the gate count of the two floating point adder's solution for interval addition.

### 2.4.2 Latency Evaluation

The latency was estimated using an independent technology metric, the logic levels. The estimation was realized IEEE double precision numbers. This metric was also used in [89][90]. In table 2.10 the latency of the proposed adder and the other five floating point adders.

Table 2.10 Latency Estimates for Floating Point Adders

| Adder | Latency (LL) |
|---|---|
| Adelaide | 29 |
| SUN | 28 |
| AMD | 26 |
| Variable Latency | 29 |
| Seidel-Even | 24 |
| Proposed | 32 |

As it can be observed, the latency of the proposed solution is higher with respect to other floating point adders. The reason for this increased latency is that the exponents' difference is performed before any mantissa processing. In other floating point adders' design, on the CLOSE path an exponent prediction is used, while on the FAR path techniques for parallel alignment shift with exponent difference are used.

### 2.4.3 Synthesis Results

Two floating point double path adder designs were modeled and synthesized using the Xilinx ISE Webpack 10.1 and Xilinx Synthesis Tool for IEEE half precision formats: the proposed design and the AMD 2000 double path adder. The two adder designs were verified using Modelsim simulations. The results are presented in Table 2.11. The two designs have in common several features:

- The compound adder used is based on Brent-Kung carry lookahead adder structure
- The alignment shifter in the FAR path and the normalization shifter in the CLOSE path is based on the barrel shifters
- The rounding computation decisions in the FAR path in both designs is the same, while in the CLOSE path is different due to the fact the AMD adder implements only effective subtractions with exponents' difference equal to 0 or 1, while the proposed adder implements both effective additions and subtractions when exponent's difference is equal to 0 or 1
- The exponent's subtraction modules were implemented using Brent-Kung carry lookahead adders
- Both designs implement the same leading zero prediction schemes.

Table 2.11 – Synthesis Results Obtained for Double Path Adders

| Double Path Adders | Maximum Combinational Delay (ns) | | | Cost (LUT-4) |
|---|---|---|---|---|
| | Logic | Route | Total | |
| AMD | 7.821 | 12.986 | 15.83 | 446 |
| Proposed Adder | 9.138 | 15.221 | 19.779 | 568 |

The synthesis results show an increase of about 27% for the proposed adder compared to the AMD 2000 [65] adder design. The difference between the estimated results presented in section 2.4.1 and the synthesis results is due to the fact that for IEEE half precision adders the ratio between the exponent's size and the mantissa's size is 5/11 (~0,5), while for IEEE double precision format the ration between exponent's size and mantissa's size is 11/53 (~0,2), so the mantissa computational path has bigger weight in the total cost of the double path adder for IEEE double precision formats. Both adder's have a similar mantissa computation in terms of cost, while the exponents and sign computation modules are duplicated for the proposed design. The synthesis results show that for the proposed adder the mantissa computational path counts for about 70% (402 LUT-4) of the all total number of LUT-4, while for the IEEE double precision estimates the mantissa computational path counts near to 80% from total cost of the proposed adder.

Regarding the interval addition, the proposed adder has an overall cost 27% higher compared to the single floating point adder solution, while compared to the

solution based on two floating point adders (892 LUT-4) the proposed adder the proposed adder has a decrease of about 35% (Fig 2.17).

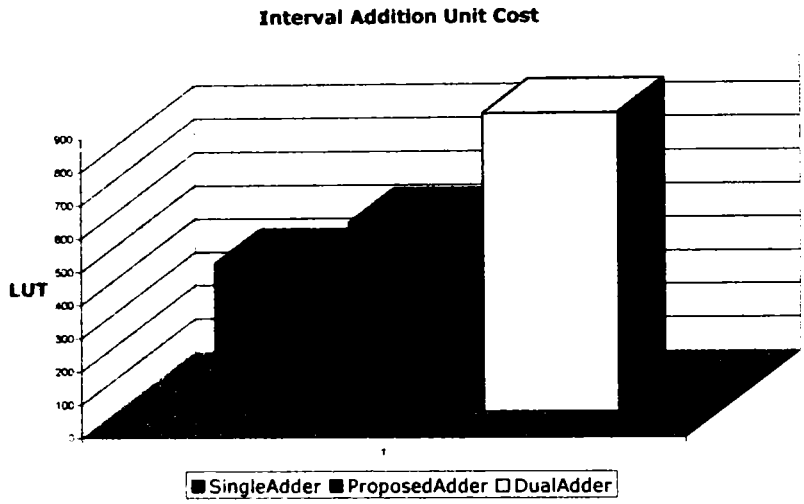



Figure 2.17 – Cost of Interval Addition Hardware Units

Regarding the latency, the synthesis results obtained with XST, show a 4 ns increase in the latency for the proposed adder (25% higher). Regarding interval addition, pipeline versions will be considered (the delay on the pipeline registers for Virtex-4 FPGA family is equal to 0.555 ns). Three types of interval adders will be considered:

- Interval adder based on a single floating point AMD double path adder with 2 pipeline stages of 8.5 ns pipeline stage
- Interval adder based on two floating point AMD double path adders with 2 pipeline stages of 8.5 ns pipeline stage
- Interval adder based on the proposed adder with 3 pipeline stages of 7.5 ns (a worst case scenario).

As it can be observed, the proposed adder can work at slightly higher frequencies compared to the AMD adder. As presented in [12][90], most double path adders have a two pipeline stages construction because each pipeline stage present the carry computation logic of the compound adder (with or without the generate and propagate bits logic (one line of AND for generate bits and of OR for propagate bits before the carry computation circuit), and the sum bits computation logic (one line of XOR after the carry computation circuit) and plus some additional logic (the compound adder in the CLOSE path is found in the first pipeline stage, while the compound adder in the FAR path is found in the second pipeline stage). Regarding the CLOSE path, the first pipeline stage incorporates also from the exponent prediction logic, complementation logic, swapping and shifting multiplexers (which are found before the compound adder). Regarding the proposed adder, the two compound adders are also each in one pipeline stage. Before these two pipeline stages another pipeline stage is built based on the exponents' subtraction modules, path selection and path multiplexers. Also in this pipeline stage, logic for the exponent prediction, swapping and right shifting can be placed. Therefore, the stage delay in the pipeline stage containing the compound adder

from the CLOSE path of the proposed adder is thus reduced. Furthermore, this stage has the biggest delay from all three pipeline stages (or from all two in the other double path adders). Therefore, the proposed adder can operate at a lower clock rate.. One exception is represented by the SUN double path adder, which has three pipeline stages, because it uses a modified compound adder which integrates rounding logic [89].

The three considered adders will have a workload of *n* interval consecutive and independent additions (one addition has operands independent of the results from previous additions). The total duration of the entire workload of additions is depicted in Table 2.12 and while a comparison between the performances for the analyzed interval addition units based on the discussed double path adders is depicted in Fig 2.18.

Table 2.12 Required Latency for Performing *n* Additions (ns)

| | Single Adder | Dual Adder | Proposed Adder | | | | |
|---|---|---|---|---|---|---|---|
| | | | *f*=0.2 | *f*=0.4 | *f*=0.5 | *f*=0.6 | *f*=0.8 |
| *n*=5 | 121 | 72 | 99 | 90 | 85,5 | 81 | 72 |
| *n*=10 | 231 | 132 | 180 | 162 | 153 | 144 | 126 |
| *n*=20 | 451 | 252 | 342 | 306 | 288 | 270 | 234 |
| *n*=50 | 1111 | 612 | 828 | 738 | 693 | 648 | 558 |
| *n*=75 | 1661 | 912 | 1233 | 1098 | 1030,5 | 963 | 828 |
| *n*=100 | 2211 | 1212 | 1638 | 1458 | 1368 | 1278 | 1098 |

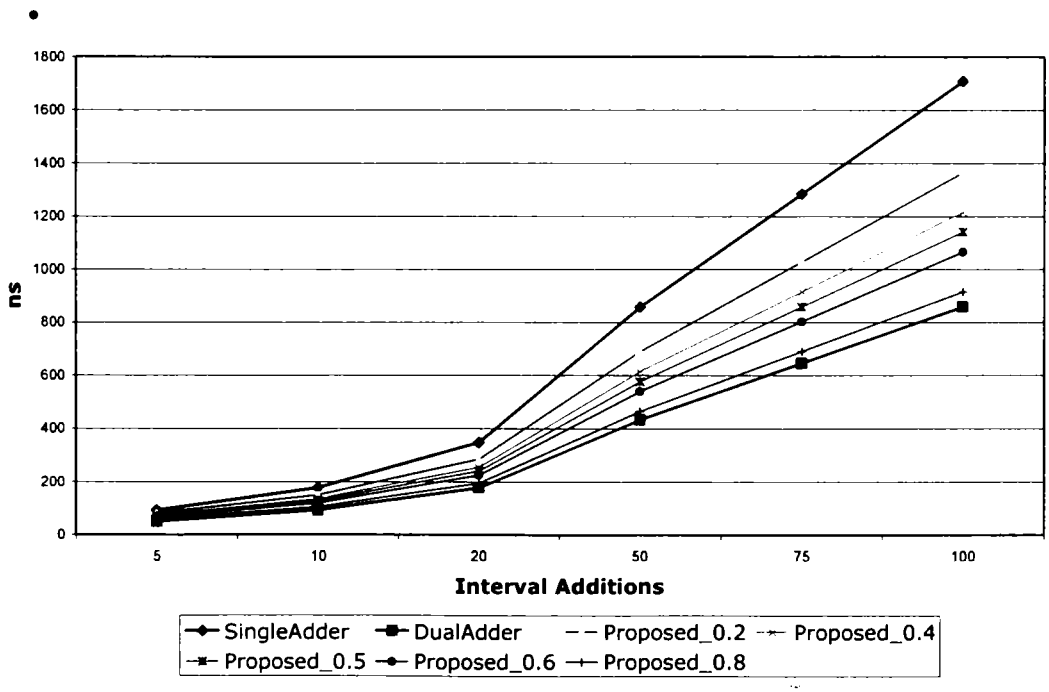



Figure 2.18 – Performance Comparison of the Three Interval Adders

As it is highlighted both in Table 2.12 and in Fig 2.18, the performance of the proposed adder depends on two important factors:

- The percentage of the favorable cases *f:* these are the cases when the two floating point additions needed for an interval operation are performed simultaneously.
- The number *n* of consecutive and independent interval additions

As shown in Fig 2.18, the performance of the proposed adder is lower with respect to the performance of the interval addition unit based on the two double path adders, but is higher compared to the interval unit based on a single double path adder. Furthermore, as the percentage of the favorable cases is higher, the performance of the proposed adder increases. Thus, when 80% of all interval additions are favorable for the proposed implementation, the performance can be close to the one of the dual adder solution.

Figure 2.19 presents the speed-ups of the interval adders implemented with two parallel AMD adders and with the proposed adder (for 20%, 40%, 50%, 60% and 80% favorable cases) with respect to the interval adder implemented with a single AMD double path adder. In this figure it is highlighted the increase in performance for the proposed adder with the increase of the number of consecutive additions. Thus, the proposed adder is suitable for dealing with series of large number of consecutive additions (as found in the vector processors), and not for isolated additions.

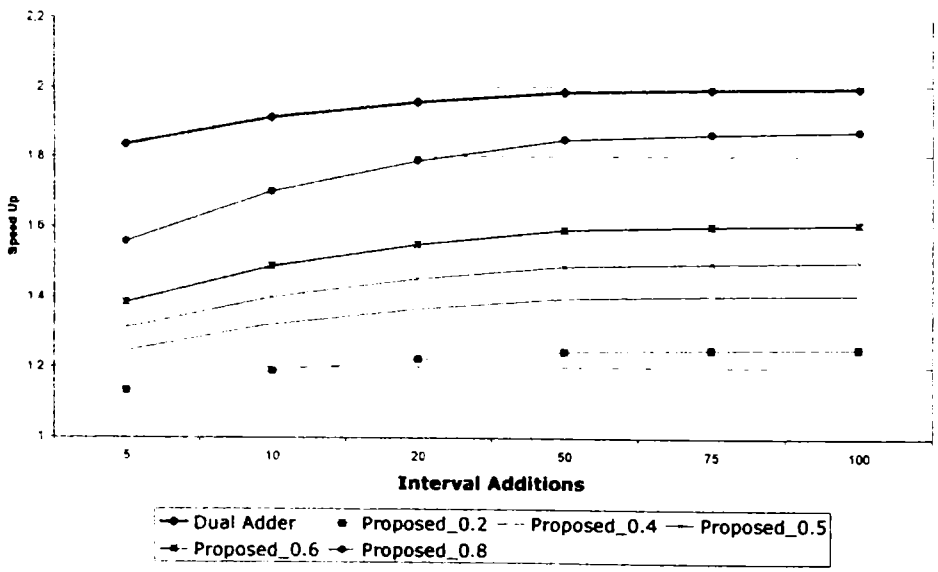



Figure 2.19 - Relative Speed-up of Interval Adder Implementations

In order to determine which of the three interval adder solutions presents the best performance-cost tradeoff a well known latency*cost metric based on the synthesis results. The latency was considered as the total duration (in ns) of *n* consecutive and independent additions. Figure 2.20 depicts the obtained results. The analysis for the proposed adder was performed for five percentages of favorable

cases (20%, 40%, 50%, 60%, 80%). The results show that for a 20% percentage of favorable cases, the proposed adder has a performance-cost trade-off close to the interval adders based on single double path adder or based on two double path adders. When the percentage of the favorable cases is higher than 20%, the proposed adder presents a better performance-cost tradeoff compared to the other two interval addition units.

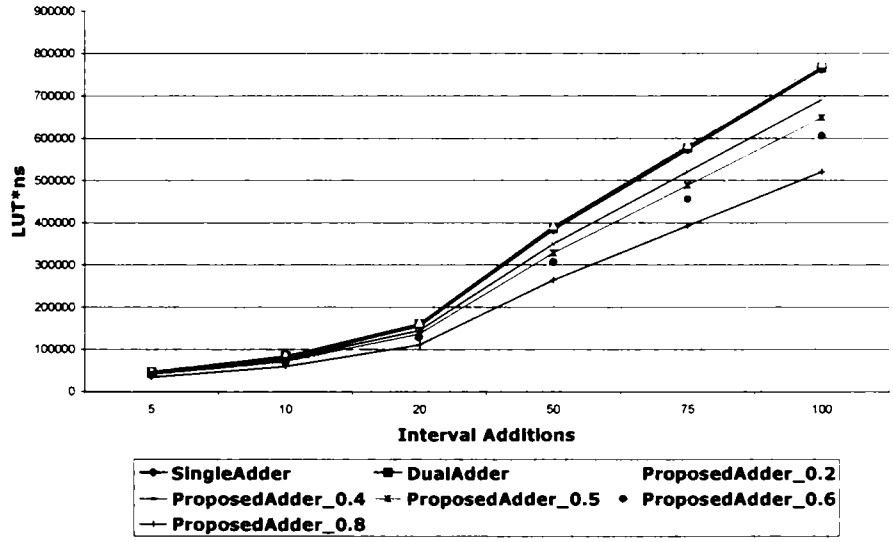



Figure 2.20 – Cost*Latency for Three Interval Adders

Based on the synthesis results, an analysis of the proposed design for conventional floating point arithmetic has been also carried out. The compared floating point adders were the AMD adder with 2 pipeline stages and the proposed adder with 3 pipeline stages.

Based on synthesis results, the latency for $n$ consecutive and independent additions was determined ($n$=5, 10, 20, 50, 75, 100). For the proposed adder the following SPEC92 FPU benchmark results were considered [69]:

- n 57% of all floating point additions and subtractions the exponents' difference is greater than 1
- 20% of all floating point addition and subtractions are effective additions with exponents' difference equal to 0 or 1
- 23% of all floating point addition and subtractions are effective subtractions with exponents' difference equal to 0 or 1

Considering these results, the percentage of favorable cases is about 62%, while the percentage of unfavorable cases is 38%.

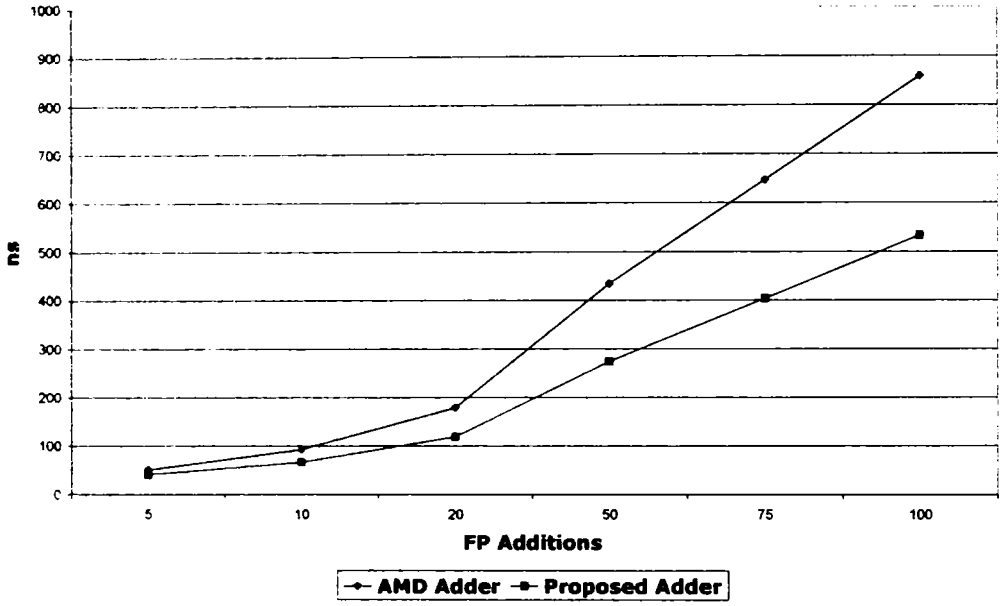



Figure 2.21 - Total Latency for *n* Consecutive FP Additions on AMD and Proposed Adder

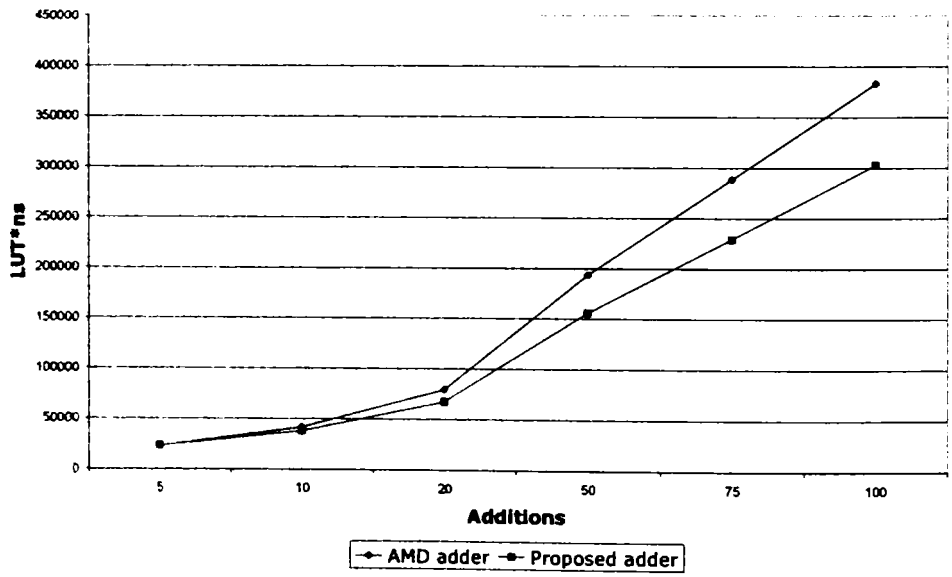



Figure 2.22 - Cost*Latency for AMD and Proposed Double Path Adder

The results of the analysis are presented in Fig 2.21. The performance of the proposed adder compared to the performance of AMD adder can be as high as 1.6 times higher. Also, as in the case of the interval addition, the proposed adder is suitable for series of large numbers of consecutive additions. When dealing isolated additions, the performance of the proposed adder is lower. The reason for this is

that the proposed adder has a higher throughput, due to the fact that in some cases (over 62%) it can perform two floating point additions simultaneously.

In Fig 2.22, a cost*latency product analysis is performed. This analysis shows that the proposed double path adder represents the best performance-cost tradeoff when dealing with series of large numbers of additions.

## 2.5 Summary

In this chapter an overview of the interval addition/subtraction hardware units is realized. For this operation two solutions have been devised: one based on a single floating point adder and one based on two floating point adders.

The next section is dedicated to the conventional floating point addition. The basic algorithm is presented. The improvements of this algorithms and the single path adder are presented. Next, a detailed presentation of the double path adder is realized. The double path adder presents special interest because the proposed solution is based on it. Five designs of double path adders are presented: three designs presented at different editions of Symposium of Computer Arithmetic (ARITH) – the Adelaide adder, variable latency adder and the Seidel-Even adder – and two patented designs by SUN and AMD.

The main contribution is presented in section 2.3. The proposed adder for interval addition is based on a double path adder. Unlike other double path adders, the proposed design exploits the parallel structure of it, by trying to perform the two operations required for an interval addition/subtraction. Several differences do appear compared to other double path adders: two exponents and sign computation circuits are needed; the exponents' difference and the path selection criterion have to be computed before any operation of the mantissas; a row of multiplexers for path distribution are placed before the two paths. Several path selection criterions may be used, but the one based on the exponents' difference and the effective operation is preferred because it has a relative simple logic and it increases the probability of occurrence of a favorable case. When performing an interval addition/subtraction on the proposed adder two cases can occur: a favorable case, when the two floating point operations can be performed simultaneously, and an unfavorable case, when the two floating point operations must be performed sequentially. The proposed adder can also be used for increasing the performance of the conventional floating point addition, by increasing the throughput of this operation.

The fourth section is dedicated to cost and performance evaluations, which were performed using both technology independent metrics and synthesis results for the Xilinx Virtex-4 FPGA family. Regarding the technology independent metric, the cost was estimated using gate count. The estimations showed that the proposed adder has an area overhead of about 20% compared to other floating point adder for double precision format. However, compared to the two adders based solution, the proposed adder cost is about 57% of the cost of the interval adder. Synthesis results performed for IEEE half precision confirmed the technology independent estimates. Regarding the latency, both technology independent metric and the synthesis results based evaluations showed that the proposed adder has a higher latency with respect to the five floating point adders studied. The overall performance of the proposed adder depends heavily on the percentage of the favorable cases. Also, the performance analysis showed that the proposed adder is suitable for series of large number of additions and not isolated additions. A

performance to cost ratio evaluation has been performed. The evaluation showed that in case of at least 20% favorable cases the proposed adder has the highest ratio, thus ensuring the best performance-cost trade-off. Furthermore, performance evaluations based on SPEC FPU benchmark results for conventional floating point addition have been performed. The analysis shows that the proposed adder is suitable when dealing with series of large numbers of additions, not with isolated additions.

# 3. Hardware Interval Multiplier

## 3.1 Interval Multiplication Algorithms

Interval multiplication is, by far, the most difficult from all four basic operations (addition, subtraction, multiplication, division). It is defined in (3.1) [49][53][86][94][96][107].

$$[X_{lo}; X_{hi}] * [Y_{lo}; Y_{hi}] = [RNI\ min(X_{lo} * Y_{lo}; X_{lo} * Y_{hi}; X_{hi} * Y_{lo}; X_{hi} * Y_{hi});\ RPI\ max(X_{lo} * Y_{lo}; X_{lo} * Y_{hi}; X_{hi} * Y_{lo}; X_{hi} * Y_{hi})] \quad (3.1)$$

In order to perform an interval multiplication, four floating point multiplications with no rounding, four comparisons for minimum and maximum and two rounding operations have to be conducted. This leads to an unacceptable low performance for the interval multiplication, mainly due to two reasons:

1. the high number of floating point operations
2. rounding is performed at the end of the operation which means that:
   - speed-up algorithms for rounding in floating point multiplication ([30][38][63][79][109]) cannot be applied
   - the floating point comparisons must be performed with numbers which are not rounded, which have a mantissa double in size
   - a dedicated rounding module (based on a large carry propagate adder) and a normalization module are needed

| a) | b) |
|---|---|
| 1. $p = X_{lo} * Y_{lo}$ | 1. $p = X_{lo} * Y_{lo}$ $r = X_{lo} * Y_{hi}$ |
| 2. $r = X_{lo} * Y_{hi}$ | 2. $q = X_{hi} * Y_{lo}$ $t = X_{hi} * Y_{hi}$ $m = min(p,r)$ $M = max(p,r)$ |
| 3. $q = X_{hi} * Y_{lo}$ $m = min(p,r)$ $M = max(p,r)$ | 3. $m = min(m,q)$ $M = max(M,q)$ |
| 4. $t = X_{hi} * Y_{hi}$ $m = min(m,q)$ $M = max(M,q)$ | 4. $m = min(m,t)$ $M = max(M,t)$ |
| 5. $m = min(m,t)$ $M = max(M,t)$ | 5. $Z_{lo} = RNI(m)$ $Z_{hi} = RPI(M)$ |
| 6. $Z_{lo} = RNI(m)$ $Z_{hi} = RPI(M)$ | |

Figure 3.1 – Interval Multiplication Pipelined Algorithm [53]
a) Using One Multiplier b) Using Two Multipliers

A pipelined version of this algorithm is presented in [53]. This version can be adapted to a hardware implementation consisting of one multiplier, two

comparators and two rounding units (fig. 3.1. a) or two multipliers, two comparators and two rounding units.

In order to reduce the penalty imposed by the rounding at the end of the operation, the floating point multiplication, the (3.1) is transformed in (3.2) [107]:

$$[X_{lo}; X_{hi}] * [Y_{lo}; Y_{hi}] = [min(RNI(X_{lo} * Y_{lo}); RNI(X_{lo} * Y_{hi}); RNI(X_{hi} * Y_{lo}); RNI(X_{hi} * Y_{hi})); max(RPI(X_{lo} * Y_{lo}); RPI(X_{lo} * Y_{hi}); RPI(X_{lo} * Y_{lo}); RPI(X_{lo} * Y_{hi}))] \quad (3.2)$$

In order to perform the interval multiplication accordingly with (3.2), eight floating point multiplications are required, three comparisons for minimum and three comparisons for maximum. In this case the main disadvantage is represented by the high number of floating point operations – fourteen. An improving can be achieved by reducing the eight floating point multiplications to four using floating point multiplier with two differently rounded results for the same multiplication [107]. In this case the number of floating point operations is ten (four multiplications and six comparisons).

A reduction of the number of floating point operations can be obtained by using an algorithm which performs only floating point multiplication with RPI [107]. This algorithm is presented in fig. 3.2.a. It requires four floating point multiplications, four comparisons and one RNI operation - nine floating point operations. The algorithm is based on the fact that the RNI result can be computed from the RPI (as presented in fig. 3.2.b). The performance of this algorithm is improved compared to the first two ways of performing the multiplication. However, it has a major disadvantage, due to the fact that it requires a dedicated RNI module, which means to a large carry propagate adder.

1. $p1 = RPI(X_{lo} * Y_{lo})$
2. $p2 = RPI(X_{lo} * Y_{hi})$
3. $p3 = RPI(X_{hi} * Y_{lo})$
4. $p4 = RPI(X_{hi} * Y_{hi})$
5. $m1 = min(p1, p2)$
   $M1 = max(p1, p2)$
6. $m2 = min(p3, p4)$
   $M2 = max(p3, p4)$
7. $M = max(M1, M2)$
8. $mi = max(m1, m2)$
9. $m = RNI(mi)$

$[X_{lo}; X_{hi}] * [Y_{lo}; Y_{hi}] = [m; M]$

a)

$RNI(X * Y) \le X * Y \le RPI(X * Y)$

$RPI(X * Y) = X * Y \Rightarrow RNI(X * Y) = X * Y \Rightarrow$

$\Rightarrow RNI(X * Y) = RPI(X * Y)$

$RPI(X * Y) > X * Y \Rightarrow$

$\Rightarrow RNI(X * Y) = pred(RPI(X * Y))$

*pred(X)* – the previous floating point number of *X* (*X* – 1 ulp)

b)

Figure 3.2 – Interval Multiplication RPI Algorithm [107] (a). RNI from RPI [107](b)

Significant improvements can be obtained by examining the sign of the operands [49][53][94][96]. Nine cases of interval multiplications are obtained (Table 3.1). In the first eight cases (when at least one of the intervals does not contain the number zero) the number of the operations is reduced to only two floating point multiplications, one with RNI, while the other to RPI. In the ninth case, when both intervals contain zero, the number of floating point operations is six: two multiplications with RPI, two multiplications with RNI and two comparisons. Using sign examination, the average performance of the interval multiplication is significantly increased. The main disadvantage of sign examination is that the number of steps differ from the first cases to the nine case (in the ninth case there are three times more floating point operations.

Table 3.1 Interval Multiplication with Sign Examining [107]

| Nr. | $X=[X_{lo};X_{hi}]$ | $Y=[Y_{lo};Y_{hi}]$ | $Z_{lo}$ | $Z_{hi}$ |
|---|---|---|---|---|
| 1 | $X_{lo}>0$ | $Y_{lo}>0$ | $RNI(X_{lo}*Y_{lo})$ | $RPI(X_{hi}*Y_{hi})$ |
| 2 | $X_{lo}>0$ | $Y_{hi}<0$ | $RNI(X_{hi}*Y_{lo})$ | $RPI(X_{lo}*Y_{hi})$ |
| 3 | $X_{lo}>0$ | $Y_{lo}<0<Y_{hi}$ | $RNI(X_{hi}*Y_{lo})$ | $RPI(X_{hi}*Y_{hi})$ |
| 4 | $X_{hi}<0$ | $Y_{lo}>0$ | $RNI(X_{lo}*Y_{hi})$ | $RPI(X_{hi}*Y_{lo})$ |
| 5 | $X_{hi}<0$ | $Y_{hi}<0$ | $RNI(X_{hi}*Y_{hi})$ | $RPI(X_{lo}*Y_{lo})$ |
| 6 | $X_{hi}<0$ | $Y_{lo}<0<Y_{hi}$ | $RNI(X_{lo}*Y_{hi})$ | $RPI(X_{lo}*Y_{lo})$ |
| 7 | $X_{lo}<0<X_{hi}$ | $Y_{lo}>0$ | $RNI(X_{lo}*Y_{hi})$ | $RPI(X_{hi}*Y_{hi})$ |
| 8 | $X_{lo}<0<X_{hi}$ | $Y_{hi}<0$ | $RNI(X_{hi}*Y_{lo})$ | $RPI(X_{lo}*Y_{lo})$ |
| 9 | $X_{lo}<0<X_{hi}$ | $Y_{lo}<0<Y_{hi}$ | $m1$ | $m2$ |

$m1$ - $min(RNI(X_{lo}*Y_{hi});RNI(X_{hi}*Y_{lo}))$

$m2$ - $max(RPI(X_{lo}*Y_{lo});RPI(X_{hi}*Y_{hi}))$

Different hardware designs have been presented for this algorithm. The hardware designs presented in [49][107] rely on two multiplexers, two floating point multipliers and one floating point comparators. The design in [96] relies on two multiplexers and one floating point multiplier (Fig 3.3). The ninth case will be handled in software.

Table 3.2 – The Subdivisions of the Ninth Case of Interval Multiplication [107]

| Nr | $X=[X_{lo};X_{hi}]$ | $Y=[Y_{lo};Y_{hi}]$ | $Z_{lo}$ | $Z_{hi}$ |
|---|---|---|---|---|
| 1 | $\lvert X_{lo}\rvert<X_{hi}$ | $\lvert Y_{lo}\rvert<Y_{hi}$ | $m1$ | $RPI(X_{hi}*Y_{hi})$ |
| 2 | $\lvert X_{lo}\rvert<X_{hi}$ | $\lvert Y_{lo}\rvert>Y_{hi}$ | $RNI(X_{hi}*Y_{lo})$ | $m2$ |
| 3 | $\lvert X_{lo}\rvert>X_{hi}$ | $\lvert Y_{lo}\rvert<Y_{hi}$ | $RNI(X_{lo}*Y_{hi})$ | $m2$ |
| 4 | $\lvert X_{lo}\rvert>X_{hi}$ | $\lvert Y_{lo}\rvert>Y_{hi}$ | $m1$ | $RPI(X_{lo}*Y_{lo})$ |

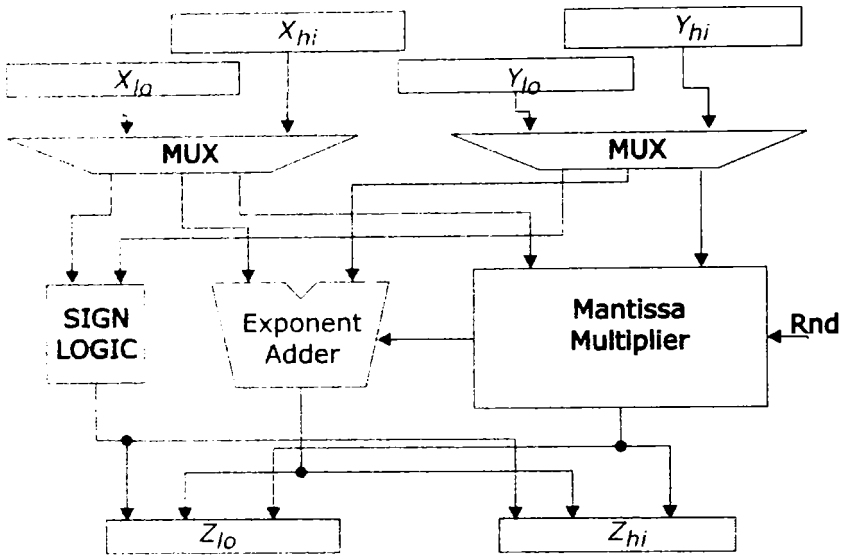



Figure 3.3 – Interval Multiplication Unit [96]

The ninth case can be divided further into four cases (Table 3.2) [107]. In these four cases, the number of operations is reduced to four: three rounded multiplications and one comparison.

However, this further division comes at a cost. As shown in Table 3.2, two more comparisons have to be performed before the multiplications, so the number of floating point operations is increased by two. Another major disadvantage is that the comparisons are performed both before and after the multiplications.

Table 3.3 presents a comparison between the presented algorithms. In this table are presented the pros and cons for each algorithm.

Table 3.3 – Comparisons between the Interval Multiplication Algorithms

| Algorithm | Pros | Cons |
|---|---|---|
| Basic Algorithm | | 10 floating point operations<br>Rounding in last stage<br>No pipelining |
| Pipelined Algorithm [53] | Easy Pipelining | 10 floating point operations<br>Rounding in last stage |
| Eight Products | Rounding within multiplication | 14 floating point operations<br>No pipelining |
| RPI Only [107] | 9 floating point operations | Needs a dedicated RNI hardware module |
| Sign Examining | 2 floating point operations in eight cases, 6 in the ninth case | Different number of steps from case to case – difficult to pipeline |
| Stine Sign Examining [94][96] | 2 floating point operations | Ninth case done in software |
| Sign Examining with Ninth Case Division [107] | 2 floating point operations in eight cases, 4 in ninth case | Comparisons done before and after the multiplication in ninth case – difficult to pipeline |

## 3.2 Floating Point Multiplication

### 3.2.1 Algorithm and Architecture

Floating point multiplication is maybe the most simple floating point operations. The multiplication of two IEEE floating point numbers ($F1 = (-1)^{s1} * 2^{E1-bias} * 1.M1$ and $F2 = (-1)^{s2} * 2^{E2-bias} * 1.M2$) is given by the following formula [26]:

$$s3 * 2^{E3-bias} * 1.M3 = F1 * F2 = (-1)^{s1 \oplus s2} * 2^{E1+E2-bias} * (1.M1 * 1.M2) \quad (3.3)$$

As it can be observed in the (3.3) the sign of the result is an exclusive-or between the two sign, the exponent is obtained by adding the two exponents and subtracting the bias, while the mantissa of the result is obtained by multiplying the two mantissas.

Because the result has also to be represented in IEEE 754 format the following steps are also requires [26]:

**1. Normalization of the mantissa** – because the mantissas of the two results are within $[1;2)$ interval, the result of their multiplication is in the range $[1;4)$; if the result is in the range $[2;4)$ a normalization left shift and incrementing the exponent are required.

**2. Rounding** – the mantissa are represented on $n$ bits, the result of their multiplication will be on $2*n$ bits; because the mantissa of the result must be represented on $n$ bits, rounding is thus required.

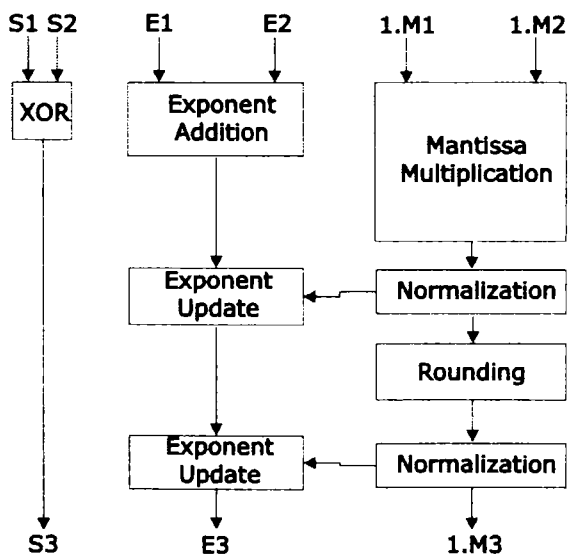



Figure 3.4 – Floating Point Multiplication Unit [74]

The basic architecture of a floating point multiplier is depicted in Fig 3.4. The largest delay module in this architecture is the mantissa multiplier. The mantissa multiplier has to be a fast parallel unsigned integer multiplier. Furthermore, the rounding unit significantly contributes to an increase in the delay of the unit, because it requires a large carry propagate adder.

In the next section, a detailed presentation of the mantissa multiplication unit will be performed. This presentation will include the general architecture, the design choices and the changes of an unsigned integer multiplication unit for floating point.

### 3.2.2 Mantissa Multiplication Unit

The mantissa multiplication unit is the largest delay module of the whole floating point multiplication unit. A right design for this module is essential for the performance of the floating point multiplication. The design choices for the mantissa multiplication unit are the tree multipliers (like Dadda trees, Wallace trees, binary trees, etc) due to the following reasons:

- greater performance than other type due to the fact that the partial products are reduced in parallel
- high performance rounding schemes (like EvenSeidel, YuZyner, Quach) for floating point multiplication can be included, eliminating the need of an extra large carry propagate adder.

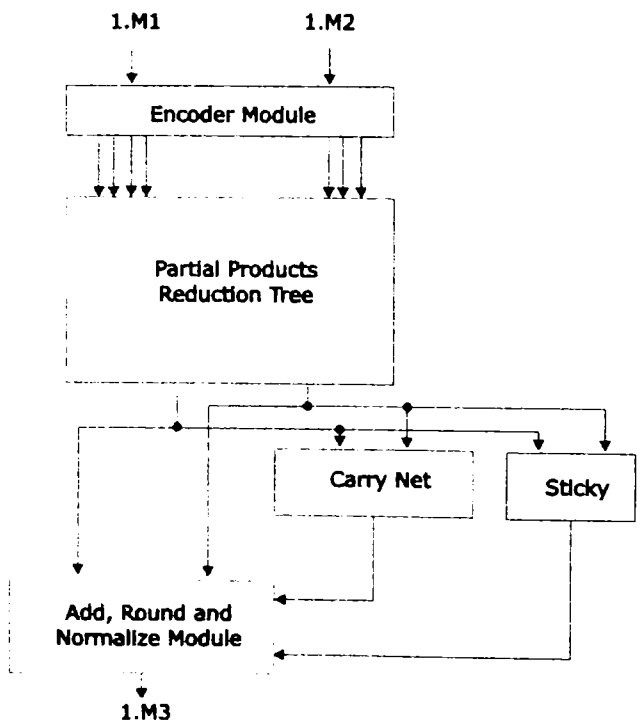



Figure 3.5 – Tree Multiplier for Mantissa Multiplication for Floating Point Numbers [26]

In Fig. 3.5 the architecture of a tree multiplier is depicted. The overall structure of such multiplier contains three major blocks [2][13]:

- partial products generation scheme (encoder module) – this module has as inputs the two multiplicands and, depending on the implemented multiplication algorithm, generates several partial products
- the partial product reduction tree – it reduces in a parallel manner the partial products generated by the previous module into two final partial products
- the final carry propagate adder – in an integer multiplier, this adder is a fast 2*n bit carry propagate adder (carry lookahead, conditional sum, prefix-adders); however; in the floating point multipliers, this adder is replaced with a more complicated scheme (that includes a compound adder, a carry-net circuit, the sticky bits computation circuit, the rounding logic) which has also the role to perform the rounding step almost simultaneously with the addition [26].

In the next three sections, each of the three major modules will be presented in detail. Different solutions, algorithms and implementations are discussed and compared for each of these modules.

### 3.2.3 Partial Product Generation Scheme

The partial product generation scheme (encoder module) has the role to encode the multiplicand ($Y$) based on the value of the multiplier ($X$) and on the multiplication algorithm in order to produce a vector of several partial products [13]. The most important design choice for this circuit is the implemented multiplication algorithm. The multiplication algorithm influences two parameters which affect the performance and the area of the entire multiplication unit:

- the number of partial products – it affects the size (both area and performance) of the partial product reduction tree
- the complexity of encoding logic – it affects the area and performance of the partial product reduction scheme

Thus, the evaluation of the encoder module will be performed in terms of the number of partial products and in terms of complexity of encoding logic.

The simplest scheme is obtained by applying a paper-and-pencil algorithm [2][14]. This algorithm is based on the fact that if a bit of the multiplier ($X$) is zero than a partial product equal to zero is generated, otherwise a partial product equal to the multiplicand ($Y$) is generated. The resulting circuit is an array of AND gates of size $n*n$. The module is depicted in Fig 3.6. The number of partial products in this case is $n$. As it can be observed in Fig. 3.6 the bits of the multiplier are entries for the rows, while the bits of the multiplicand are entries for the column. The main advantage of this scheme is its low complexity of the encoding logic (the multiplier is not encoded at all, while the multiplicand is encoded using only one AND gate) which means a low latency (only 1 LL) and a low area ($n*n$ AND gates). The main disadvantage is its high number of partial products ($n$ partial products).

A more complicated scheme is the one based on the Booth multiplication algorithm. This algorithm relies on the comparison of two consecutive bits [51][75]

[100]. If there is a string of zero's or a string of one's than the generated partial product is equal to zero. Table 3.3 presents the Booth encoding. This multiplication is suited for signed numbers, because it does not require correction additions.

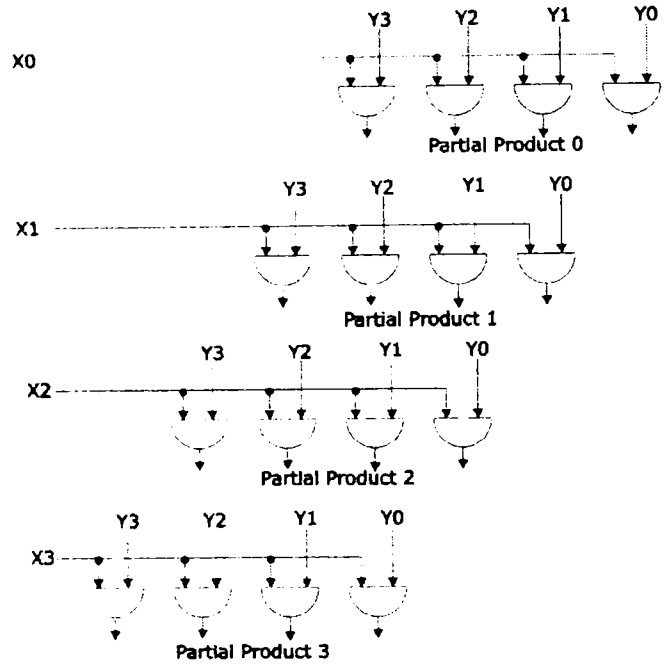



Figure 3.6 – Encoder Module with AND Gate Array for 4-Bits Multiplicands [13]

However, for floating point multiplication (where unsigned multiplication is used), this algorithm is worst compared to the simpler paper-and-pencil algorithm due to the following reasons [14]:

- it requires an encoding of the multiplier, which leads to a more complex encoder module (as seen in Fig 3.7)
- it involves subtractions, which require obtaining two's complement numbers, which means an increase of the partial products by 1 (the extra 1 comes to ensure that the last operation is addition so the overall result is positive)

Table 3.4 – Booth's Algorithm [51]

| $X_i$ | $X_{i+1}$ | Comments | Operation | Partial Product |
|---|---|---|---|---|
| 0 | 0 | String of 0's | No operation | 0 |
| 0 | 1 | Beginning of a string of 1's | Addition | $Y$ |
| 1 | 0 | End of a string of 1's | Subtraction | $\overline{Y}+1$ |
| 1 | 1 | String of 1's | No operation | 0 |

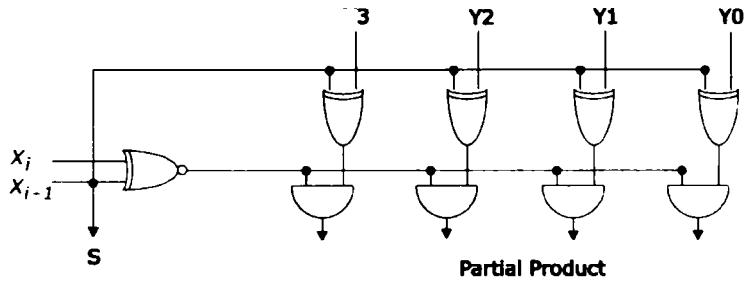
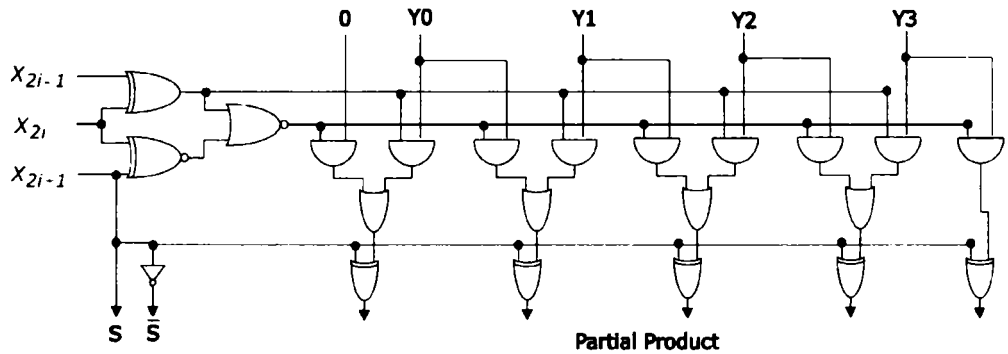
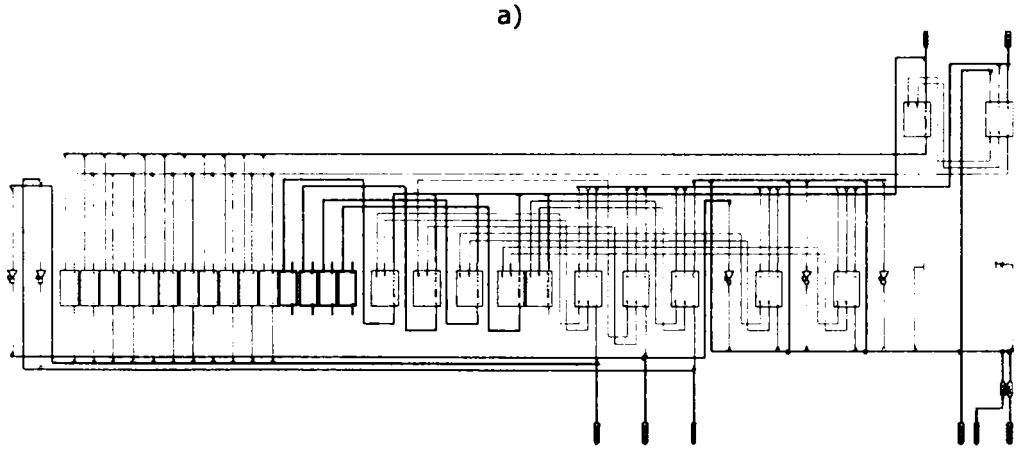



Figure 3.7 - Partial Product Generation Line for 4 Bit Multiplicands Using Booth

A method of generating fewer partial products is by using a higher radix. A very important algorithm used is Booth radix 4 [51] or Booth 2 [13][14]. Table 3.4 depicts the encoding of Booth 2. In Fig. 3.8 are depicted the encoding logic of the multiplier and while in Fig 3.9 is presented the dot diagram (which illustrates the generation of the partial products) for this algorithm. The number of partial products generated by this algorithm is $\left\lceil \frac{n+1}{2} \right\rceil$, so a reduction by almost a factor of two is obtained compared to the simple paper-and-pencil algorithm. However, the encoding logic is more complex, as shown in Fig 3.8, the delay being of 5 LL.




a)



b)

Figure 3.8 - Partial Product Generation Line for 4 Bit Multiplicand Using Booth 2 [13] (a) Technology Schematic of Booth 2 Encoder Module (obtained with XST) (b)

Table 3.5 - Booth 2 Algorithm [13]

| $X_{2i+1}$ | $X_{2i}$ | $X_{2i-1}$ | Operation | Partial Product |
|---|---|---|---|---|
| 0 | 0 | 0 | No operation | 0 |
| 0 | 0 | 1 | Addition of $Y$ | $Y$ |
| 0 | 1 | 0 | Addition of $Y$ | $Y$ |
| 0 | 1 | 1 | Addition of $2Y$ | $2Y$ |
| 1 | 0 | 0 | Subtraction of $2Y$ | $\overline{2Y}+1$ |
| 1 | 0 | 1 | Subtraction of $Y$ | $\overline{Y}+1$ |
| 1 | 1 | 0 | Subtraction of $Y$ | $\overline{Y}+1$ |
| 1 | 1 | 1 | No operation | 0 |

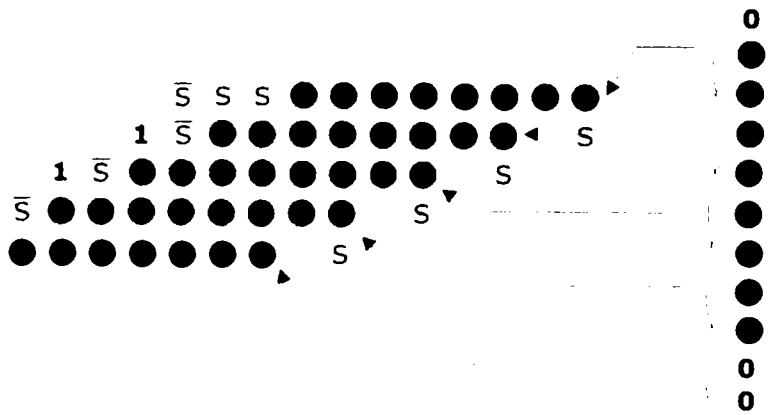



Figure 3.9 - The Dot Diagram of Booth 2 Algorithm for 8 Bit Multiplicands [13]

A further decrease in the number of partial products can be obtained by using high radices, like Booth radix 8 (Booth 3) or Booth radix 16 (Booth 4). With Booth 3 the number of partial products is $\left\lceil \frac{n+1}{3} \right\rceil$ [2][13]. The Booth 3 encoding is presented in Table 3.5. As it can be observed in Table 3.5, this algorithm requires obtaining the $3Y$ multiple. This is a major drawback of this algorithm, because obtaining $3Y$ multiple (which is called hard multiple) requires a large carry propagate adder (unlike $2Y$ or $4Y$ which can be obtained by shifting). Higher radices (16, 32, etc) further require $5Y$ , $7Y$ multiples, which makes this algorithm not viable and quit unfeasible. Furthermore, the encoding logic is very complicated compared to the simple paper-and-pencil and Booth 2.

An improvement of Booth 3 is proposed in [13], by using a redundant representation of the partial products (redundant Booth). This algorithm makes use of small adders with no carry propagation between them. The redundant form is treated as two separate numbers with one of them with large gaps of zero. In order to avoid that the large gaps of zero will become large gaps of ones for negative multiplicands (which are obtained by a bit-inversion), a constant is added to each small adder.

Table 3.6 – Booth 3 Encoding [2]

| $X_{3i+2}$ | $X_{3i+1}$ | $X_{3i}$ | $X_{3i-1}$ | Partial Product | $X_{3i+2}$ | $X_{3i+1}$ | $X_{3i}$ | $X_{3i-1}$ | Partial Product |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | $\overline{4Y} + 1$ |
| 0 | 0 | 0 | 1 | $Y$ | 1 | 0 | 0 | 1 | $\overline{3Y} + 1$ |
| 0 | 0 | 1 | 0 | $Y$ | 1 | 0 | 1 | 0 | $\overline{3Y} + 1$ |
| 0 | 0 | 1 | 1 | $2Y$ | 1 | 0 | 1 | 1 | $\overline{2Y} + 1$ |
| 0 | 1 | 0 | 0 | $2Y$ | 1 | 1 | 0 | 0 | $\overline{2Y} + 1$ |
| 0 | 1 | 0 | 1 | $3Y$ | 1 | 1 | 0 | 1 | $\overline{Y} + 1$ |
| 0 | 1 | 1 | 0 | $3Y$ | 1 | 1 | 1 | 0 | $\overline{Y} + 1$ |
| 0 | 1 | 1 | 1 | $4Y$ | 1 | 1 | 1 | 1 | 0 |

Compensation constant is added to all partial products in order to obtain a net result added to the partial products reduction tree equal to zero. Table 3.6 presents the redundant Booth 3 algorithm, while Fig. 3.10 depicts the dot diagram of this algorithm.

Table 3.7 – Redundant Booth 3 Encoding [13]

| $X_{3i+2}$ | $X_{3i+1}$ | $X_{3i}$ | $X_{3i-1}$ | Partial Product | $X_{3i+2}$ | $X_{3i+1}$ | $X_{3i}$ | $X_{3i-1}$ | Partial Product |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $k + 0$ | 1 | 0 | 0 | 0 | $k + \overline{4Y} + 1$ |
| 0 | 0 | 0 | 1 | $k + Y$ | 1 | 0 | 0 | 1 | $k + \overline{3Y} + 1$ |
| 0 | 0 | 1 | 0 | $k + Y$ | 1 | 0 | 1 | 0 | $k + \overline{3Y} + 1$ |
| 0 | 0 | 1 | 1 | $k + 2Y$ | 1 | 0 | 1 | 1 | $k + \overline{2Y} + 1$ |
| 0 | 1 | 0 | 0 | $k + 2Y$ | 1 | 1 | 0 | 0 | $k + \overline{2Y} + 1$ |
| 0 | 1 | 0 | 1 | $k + 3Y$ | 1 | 1 | 0 | 1 | $k + \overline{Y} + 1$ |
| 0 | 1 | 1 | 0 | $k + 3Y$ | 1 | 1 | 1 | 0 | $k + \overline{Y} + 1$ |
| 0 | 1 | 1 | 1 | $k + 4Y$ | 1 | 1 | 1 | 1 | $k + 0$ |

Table 3.8 - Comparison between Simple, Booth 2 and Redundant Booth 3 Algorithms for 53 Bits operands

| Algorithm | Latency (LL) | Partial Products | Gate Number |
|---|---|---|---|
| Simple | 1 LL | 53 | 2809 |
| Booth 2 | 5 LL | 27 | 5832 |
| Booth 3 | 7 LL | 18 | 6495 |

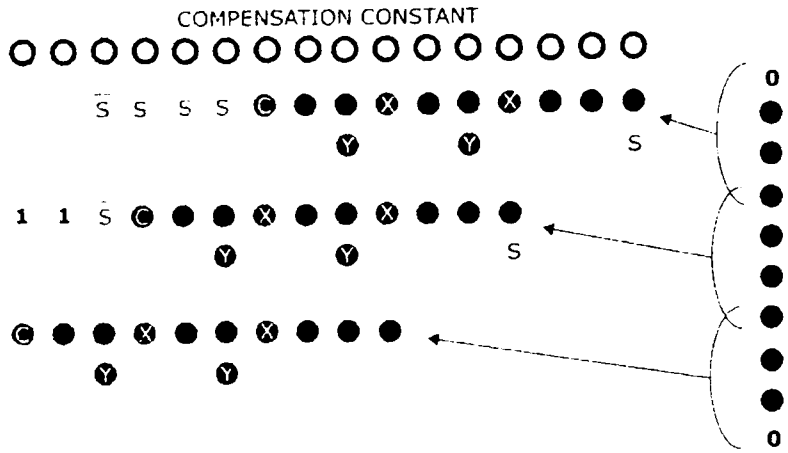



Figure 3.10 – The Dot Diagram of Redundant Booth 3 [13]

The impact of these implementations on the performance and area of the whole multiplication depends on the technology choice and on the partial products reduction tree. A comparison between the simple paper-and-pencil and the Booth 2 algorithm was made in [14] using a reduction tree based on [4:2] compressors showed that the simple algorithm had a higher performance compared to Booth 2. In [13][75] the Booth algorithm shows a higher performance than the simple paper-and-pencil algorithm (although not very significant). The designs in [35][63] used the Booth encoding instead the simple algorithm, this being a proof that Booth algorithm shows a higher performance for their technology. Table 3.8 presents a comparison between the three algorithms for multiplication, in which the logic levels, the partial products and the gate count are presented.

### 3.2.4 Partial Product Reduction Tree

The partial product tree reduces the several partial products which are produced by the encoding module to only two final partial products, which must be added using a carry propagation adder in order to obtain the product. The reduction is done after several levels, using a series of ($m$, $n$) counters or compressors (where $m>n$, and $m$ is number of input vectors, while $n$ is number of output vectors). The counters on same level work in parallel. The number of levels of such trees is logarithmic dependent on the number of partial products ($O\left(\log_{m/n} p\right)$, where $p$ is the number of partial products).

The most simple type of counter is (3,2) counter, presented in Fig. 3.11 which is basically a carry-save adder [26]. It is built of a number of full adder cells (FAC), with no carry propagation between them. It has three vectors as inputs, and two vectors as outputs (sum and carry).

The simplest tree that can be built with (3:2) counters is the Wallace tree [101]. The number of levels in a Wallace tree is equal to $\left\lceil \log_{3/2} p \right\rceil$-1. A Wallace tree for 18 partial products is presented in Fig. 3.12. In terms of logic levels measured latency the Wallace tree has the minimum delay [2].

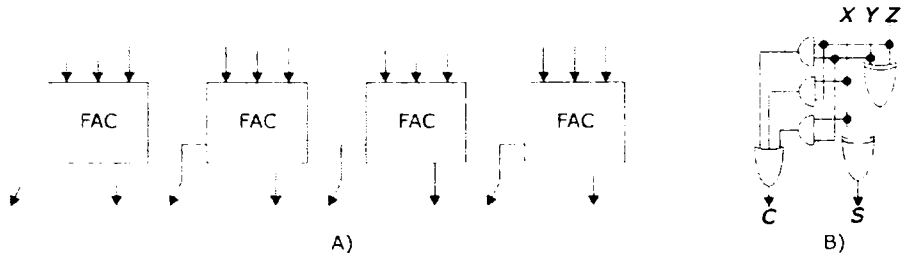



Figure 3.11 – The (3:2) Counter
A) The Block Structure for 4 Bit Vectors B) Internal Structure of a FAC

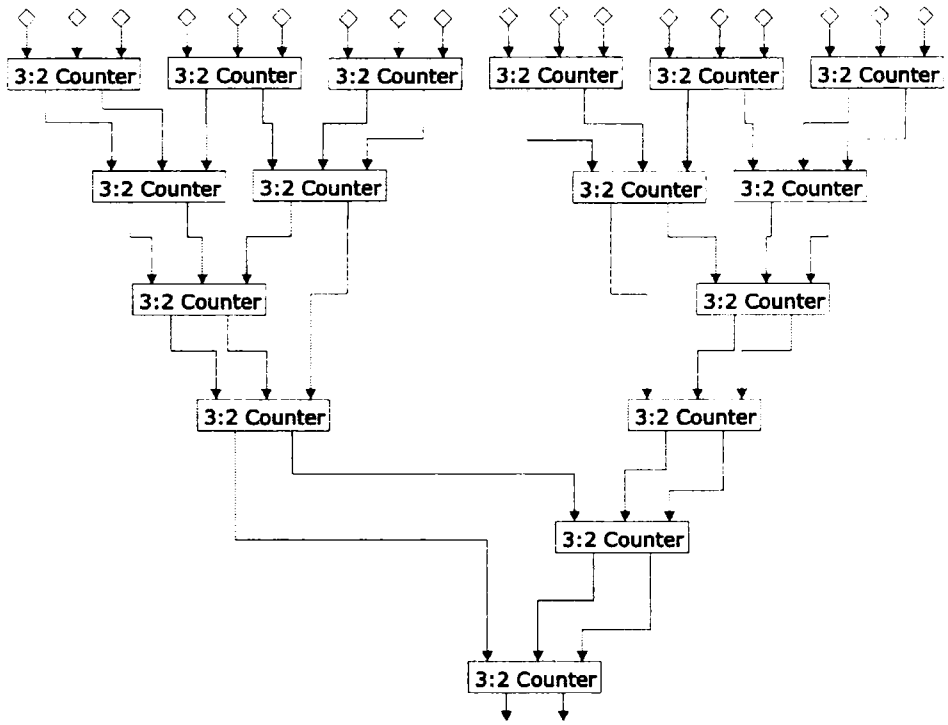



Figure 3.12 – The Wallace Tree for 18 Partial Products (◇ - Partial Product) [2]

A version of the Wallace tree is the Dadda tree [23], which presents a lower number of counters. However, this kind of tree has a major drawback: its topology is considered highly irregular, because it does not specify a systematic method for counter connections. Furthermore, a Wallace tree requires a large number of wiring tracks, which means a large area [2][51]. Furthermore, this irregularity can also increase the delay, if we were to consider the wiring delays.

Two regular topologies can be obtained using the (3:2) counters. The first one is the overturned staircase tree [62]. This type is being build from a root (which is the counter on the last level) and a body of height $k$ (which is the equal to the number of counter levels of the body). The number of counter levels is $k+1$. A body of height $k$ is built from a body of height $k-1$, a connector (built of two counters)

and a branch (which is built from serially line of *k-2* counters). Thus, a recursive method for building the overturned staircase tree is obtained. The regularity of this topology can also be seen from the Fig. 3.13, which is a tree for 18 partial products. This type of tree can have in some cases the same number of counter levels as the Wallace tree (for 18 partial products both trees have 6 levels). However, the number of wiring tracks is significantly lower, thus a reduced area is obtained and wiring delays are also significantly slower. The tree depicted in Fig. 3.13 is an overturned staircase tree of order 1. By replacing each branch with an overturned staircase of order 1, an overturned staircase of order 2 can be obtained. This type of tree has a smaller number of levels compared to the order 1, but it needs a greater number of wiring tracks [2].

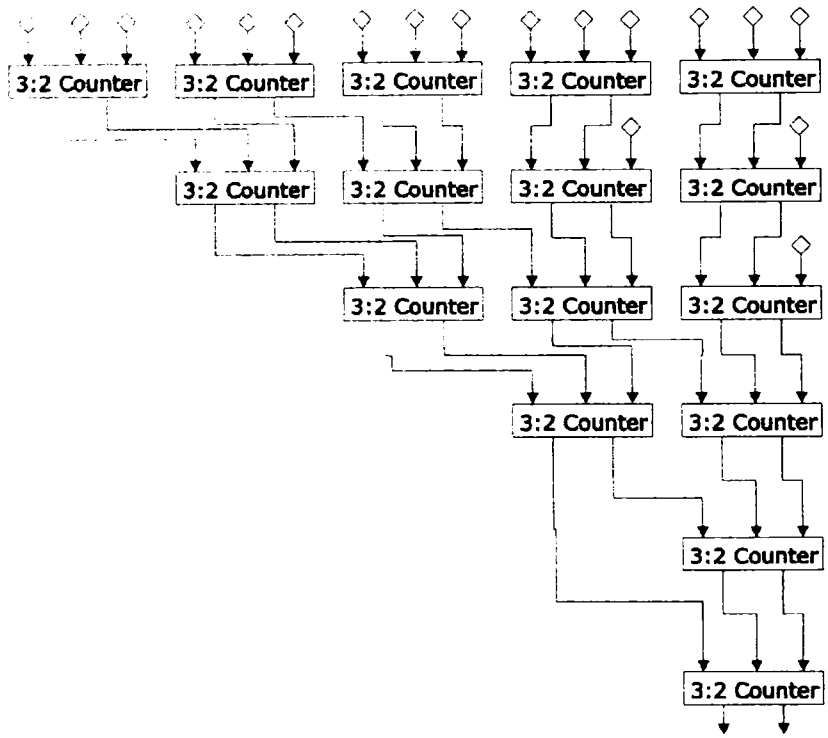



Figure 3.13 – The Overturned Staircase Tree for 18 Partial Products
(The ◇ Represents the Partial Product) [2]

Another regular topology based on the (3:2) counter is the balanced delay tree [110] – Fig. 3.14. A balanced tree of height *k+2* is constructed by connecting a branch (a chain of serially connected counters) of height *k* with a balanced tree of height *k*.

As with the overturned staircase tree, the balanced delay tree constructed in this way is of order 1. An order 2 balanced tree is built by replacing each branch with an order 1 balanced delay tree. This type of tree has the highest regularity from all types of trees and requires the smallest number of wires tracks. In terms of counter levels, this type has a bigger number compared to Wallace tree or overturned staircase tree. A very important aspect of the balanced delay tree and of the overturned staircase tree is that the wiring track is dependent on the order of the tree and not on the number of partial products [2].

Another type of counter used in reduction trees is the (4:2) counter or (4:2) compressor. First described in [106], the (4:2) compressor is depicted in Fig. 3.15 a. It is built from two (3:2) counters, thus the delay of this type of counter is doubled compared to the (3:2) counter. Another version of the (4:2) compressor is presented in Fig. 3.15. b [72]. This version may constitute an improvement if the delay of the XOR gate is considerable higher than the OR and AND gates; the critical path contains three XOR gates in this version; while in Fig. 3.14.a the critical path contains four XOR gates.

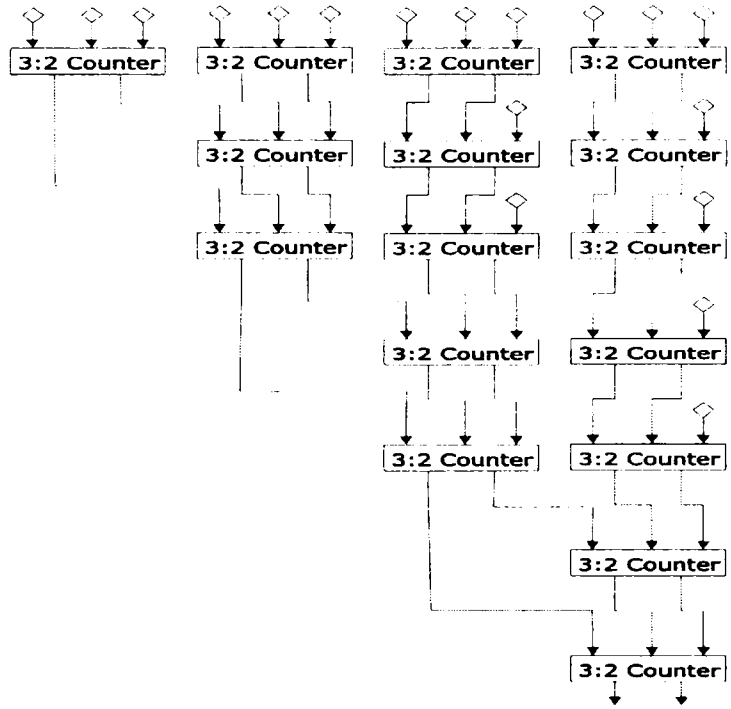



Figure 3.14 – The Balanced Delay Tree for 18 Partial Products
(The ◇ Represents the Partial Product)[2]

A very important design feature of the (4:2) compressors and also of higher order counters ((*n*: 2) counters, where n>4) is that the carry out of the counter (Cint-o in Fig 3.14) is independent from the carry in (Cint-I in Fig 3.14). In this way, it is avoided the ripple carry affect which propagates the delays across the chain of counters [2].

With the (4:2) compressor a binary tree can be built [72][106]. Figure 3.16 depicts a binary tree for 16 partial products .This type of tree presents a number of $\lceil log_2 p \rceil - 1$ compressor levels. This means a major reduction compared to Wallace tree or other (3:2) counter based trees. However, each compressor level presents an increased delay (1.5 to 2 times higher) compared to the delay of the levels on (3:2) counters based trees. Therefore, for a smaller number of partial products (integer multiplication, simple precision floating point multiplication), the (3:2) counters based trees have a higher performance, while for a large number of partial products the binary tree has a smaller delay. In terms of topology, the binary tree is considered a regular topology, however the number of wired tracks is greater

compared to overturned staircase tree or balanced delay tree, and lower compared to the Wallace tree.

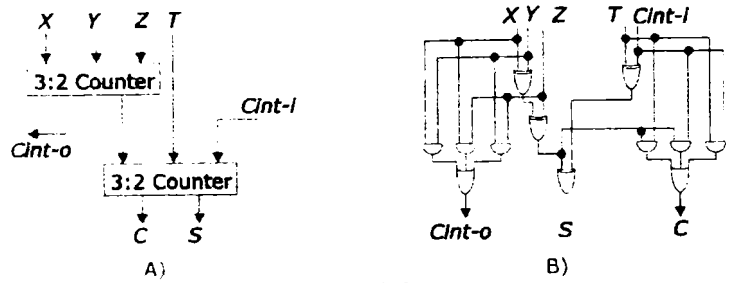



Figure 3.15 – 4:2 Compressors
a) [106] Compressor b) [72] Compressor

With higher order counters, like (7:3) counters, (9:2) counters, and the number of counters levels is further reduced. However, the delay of these counters is significantly increased.

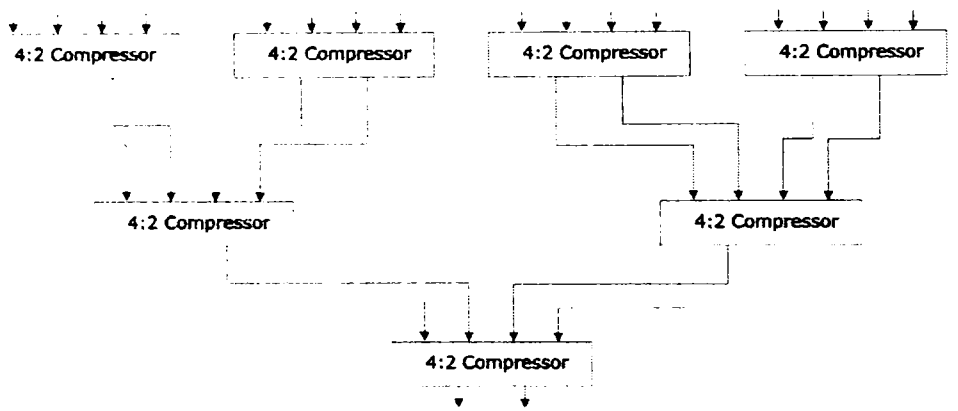



Figure 3.16 – The Binary Tree for 16 Partial Products [2]

Table 3.9 – Comparison of Partial Product Reduction Trees for 27 Partial Products

| Reduction Tree | Counter Levels | Delay (LL) | Gate Count | Wired Tracks |
|---|---|---|---|---|
| Wallace Tree | 7 | 14 | 10734 | 12 |
| Overturned Staircase | 8 | 16 | 12750 | 6 |
| Balanced Delay | 9 | 18 | 11316 | 5 |
| Binary Tree | 4 | 12 | 11532 | 10 |

In table 3.9 a comparison between the four types of partial products reduction trees is done. The features followed are number of counter levels, delay in LL, gate count and wired tracks. The comparison is done for trees used to reduce 27 partial products (this number is obtained by a Booth 2 encoding for double precision numbers). Although the number of wired tracks is an important aspect in VLSI design, for the rest of our analysis it will not be considered this aspect, because the

choice of the partial products reduction tree does not affect the final addition and rounding step, where my contribution is proposed; the analysis of the overall performance and cost of the existing and proposed multipliers will be performed taking in consideration that both will have the same type of partial product reduction tree. In my implementations, I used the Wallace tree, thus all the comparisons are done for multipliers with Wallace trees.

### 3.2.6 Final Addition and Rounding

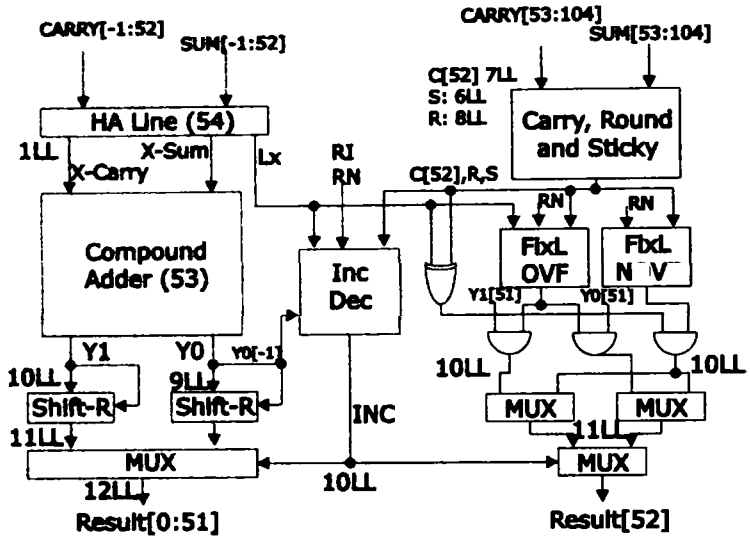
The partial product reduction tree reduces the several partial products into only two final partial products, in carry and sum format. These final partial products are represented on a double number of bits compared to the two operands [26][30]. Thus for the final addition, a double size carry propagate adder must be used. Furthermore, the final product will be double size compared to the operands. When multiplying IEEE 754 mantissas, the final result have to be on the same number of bits as the operands. Thus, a rounding step must be performed. Furthermore, because IEEE 754 mantissas are in the range $[1;2)$, the result mantissa will be in the $[2;4)$ range. Thus, a normalization left shift is also required. Thus, the hardware architecture resulted for multiplication of IEEE 754 floating point numbers is presented in Fig. 3.4. This structure presents a high latency due to:

- the double size final carry propagate adder
- the computation of rounding and sticky bits
- a large carry propagate adder for rounding

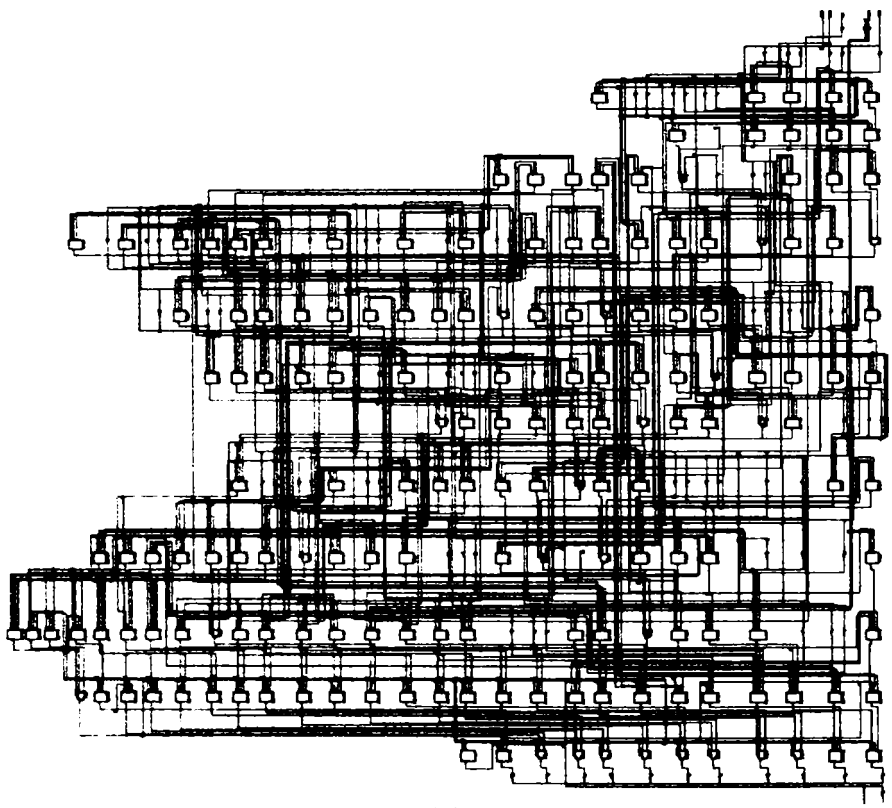
In order to increase the performance of the floating point multiplication, several rounding algorithms for multiplication have been developed, like the ones in [30], [79], [109]. These algorithms are based on a separation of the two final partial products (denoted SUM and CARRY) into a lower and an upper part. The most significant parts are added with a compound adder, while the lower parts are used to compute (in parallel with the addition) the rounding decision. The rounding decision is computed for both the case of when the final partial product is in $[1;2)$ range and $[2;4)$ range. This way the final addition, rounding and normalization are included into a single step. This type of final addition and rounding is much faster because:

- the size of the carry propagate adder is almost the size of the mantissa
- the rounding decision (including rounding and sticky bits) is computed in parallel with the addition
- the rounding is reduced to a simple selection, thus there is no need for the large carry propagate adder specially for rounding

The (Even-Seidel [30]) algorithm is based on the addition of an injection constant, in order to reduce all the rounding modes to RZ. The injection value is 0 for RZ, $2^{-53}$ for RNU, and $2^{-53}+2^{-104}$ for RI. RNE can be obtained from RNU, by a correction of the least significant bit when this is bit is equal to 1 and the unrounded result is exactly at half distance between the two numbers. RNI and RPI can be obtained form RI and RZ after a sign examination of the result (RNI is RZ when the result is positive and RI when the result is negative). The injection is added as another partial product. This implies that the number of counter levels can increase by 1.




(a)



(b)

Figure 3.17 – The Even-Seidel Rounding Scheme [30] (a)
Technology Schematic Obtained with XST (b)

The Even-Seidel algorithm for IEEE 754 double format, as described in [30], proceeds as follows:

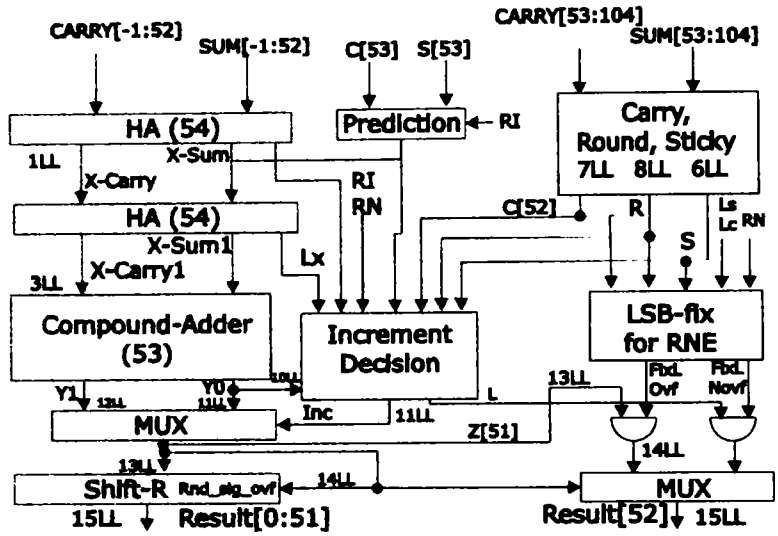
1. The SUM and CARRY are divided into a high part consisting of 54 bits and a low part consisting 52 bits.
2. With the low part are computed C – carry bit, R – round bit and S – sticky bit. The C bit is the carry out if the two lower parts would have been added. The R bit is the most significant bit of the sum if the lower parts would have been added. The S bit is obtained by an OR of all other bits of the sum if the lower parts would have been added.
3. The higher part is input to a line of half adders and produces X-Sum and X-Carry on 53 bits and the sum bit of the least significant position (Lx).
4. The X-Sum and X-Carry are inputs of a compound adder. The results are Y0 and Y1 (where Y1=Y0+1 ulp).
5. An increment decision box receives the R bit, the C bit, the Lx bit and the most significant bit of Y0. The increment decision is taken on two paths (one on the assumption that no overflow will occur, while the other on the assumption that the overflow will occur). The most significant bit of Y0 will make the selection between the two paths. The increment decision signal is defined by:

$$INC = \begin{cases} L_X C & \text{if MSB(Y0)=0 or RZ} \\ L_X + C & \text{if MSB(Y0)=1 or RI} \\ R + L_X + C \geq 2 & \text{if MSB(Y0)=1 or RNE} \end{cases}$$

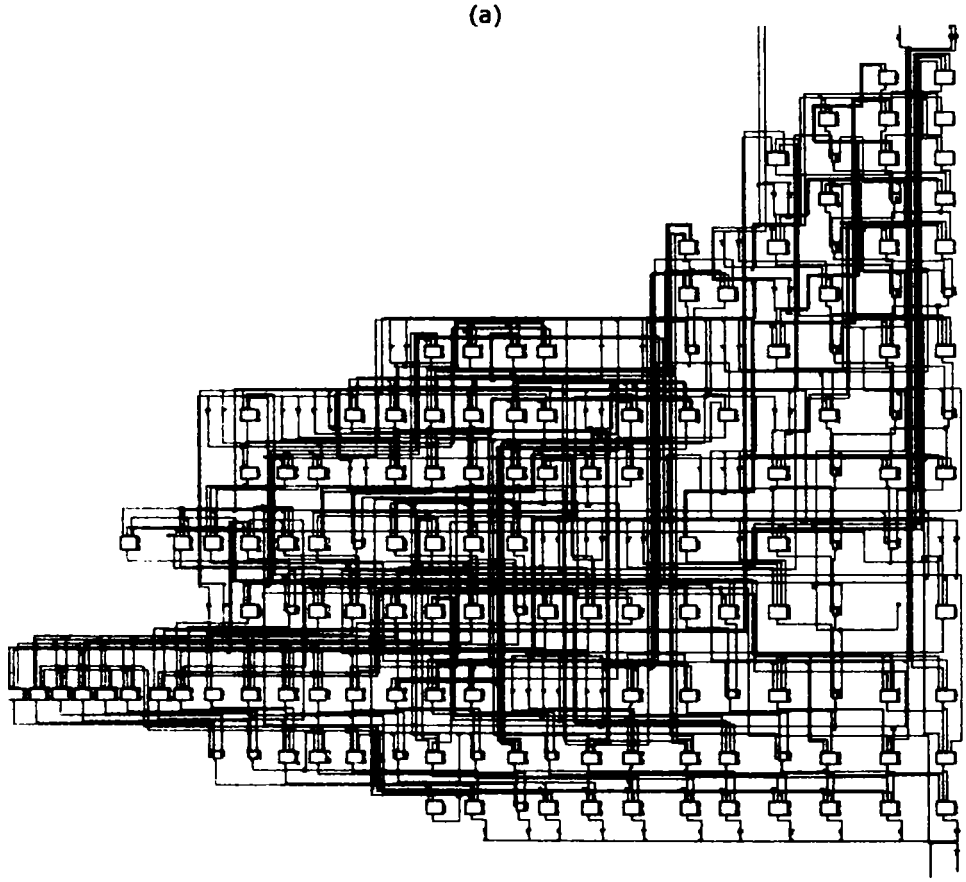
6. The most significant bits of Y0 and Y1 indicate whether these two results are in the range $[2;4)$; if Y0 or Y1 is in this range, it is shifted with one position towards right.
7. The most significant 52 bits of the rounded result are selected between the possibly shifted Y0 and Y1. The selection is made based on the increment decision.
8. In case the rounding mode is RNE, the least significant bit may be corrected. The possible correction is computed on two paths, one on the assumption that there is no overflow, while the other on the assumption there is overflow. The least significant bit is computed for the two paths and selected based on the increment decision.

Fig 3.17 presents the hardware structure which implements this algorithm. The delay of this structure is 12 LL, if the injection doesn't increase the number of counter levels of the partial product reduction tree, or 14 LL if the injection does increase the number of the counter levels by 1.

1. Another rounding algorithm for floating point multiplication is based on reducing the rounding modes on RZ, RNU and RI and on the injection of a prediction bit based on the rounding mode. Thus, this rounding mode is similar to the Even-Seidel algorithm.




(a)



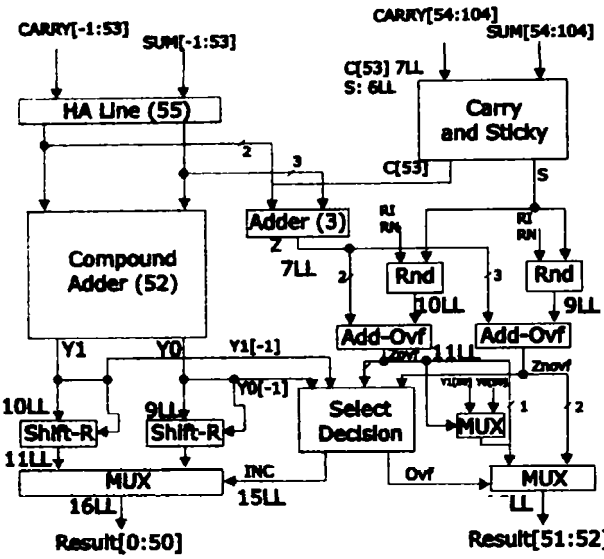
(b)

Figure 3.18 – The Quach Rounding Scheme [79] (a)
Technology Schematic of Quach Rounding Scheme Obtained with XST(b)

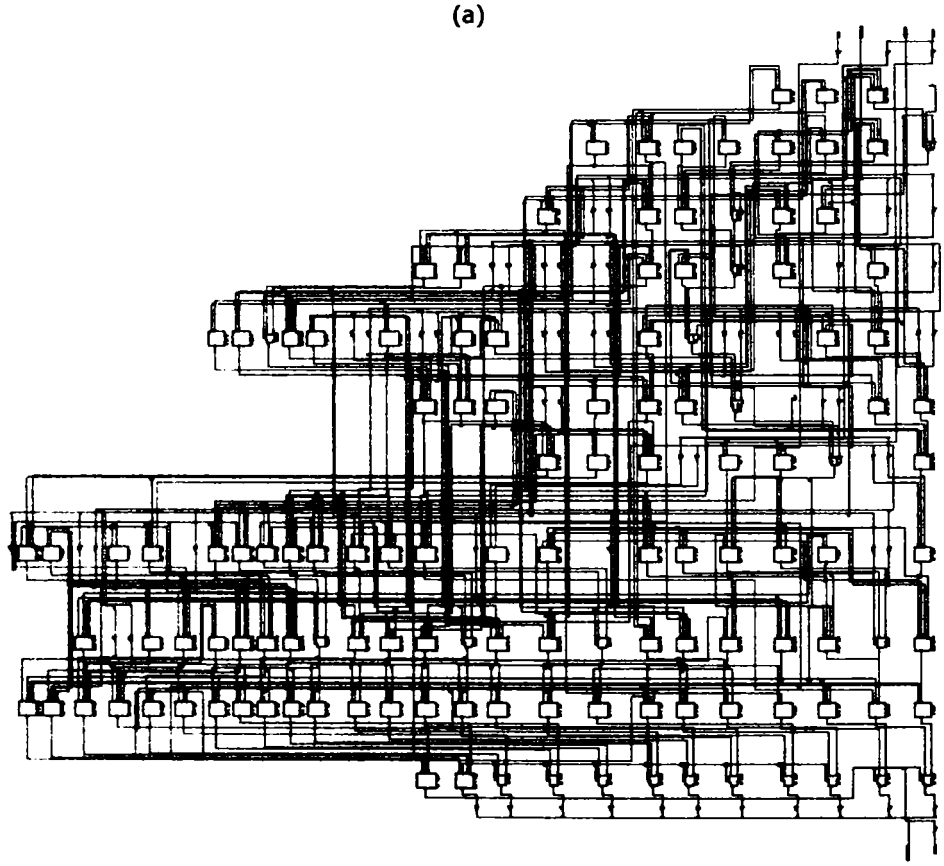
This algorithm was developed at Stanford University and was proposed by Quach, Takagi and Flynn [79] and also optimized by Even and Seidel [30]. This algorithm proceeds as follows, as described in [30]:

1. The SUM and CARRY are divided into a high part consisting of 54 bits and a low part consisting 52 bits.
2. With the low part are computed C – carry bit, R – round bit and S – sticky bit. The C bit is the carry out if the two lower parts would have been added. The R bit is the most significant bit of the sum if the lower parts would have been added. The S bit is obtained by an OR of all other bits of the sum if the lower parts would have been added.
3. A prediction bit (PRED) is computed.
4. The higher part is input to a line of half adders and produces the X-Sum on 54 bits an X-Carry on 53 bits.
5. The X-Carry string is completed with the PRED, and with the X-SUM are inputs for another line of 54 half adders. The results are X'-Sum and X'-Carry on 53 bits, while the least significant bit of the sum is Lx.
6. The X'-Sum and X'-Carry are inputs for a compound adder. The results are Y0 and Y1 (which is equal to Y0+1).
7. An increment decision is computed, based on the rounding mode, the Lx bit, the C, R, S bits and on the most significant bit of Y0. The increment decision is computed on two paths, one on the assumption that overflow occurs, while the other on the assumption that no overflow occurs.
8. The least significant bit is computed for the case that no overflow occurs:
9. In case of RNE the fixing of the least significant bit of the result might be needed. The fixing is also computed on two paths and two signals are generated (FIX-Lovf, FIX-Lnovf).
10. Based on the INC signal, a selection between Y0 and Y1 is performed. The result of this selection is Z (on 53 bits)
11. The normalization of the Z is produced in case that an overflow occurs (the most significant bit of Z is 1). In this way the most significant 52 bits of the result are obtained.
12. The least significant bit is obtained by a selection based on the most significant bit of Z (which determines whether an overflow occurred). The least significant bit of the result is either the corrected least significant bit of Z, either the corrected L.

The performance of this algorithm is 15 LL. Figure 3.18 depicts the implementation of this rounding algorithm. This algorithm uses the prediction for the case of RI. This prediction is needed in order to reduce the number of possibilities of the rounded result. A slight modification of this algorithm is presented in [38]. The modification relies on reducing the two half adder lines into a row of 53 half adder, while the prediction is added using a full adder cell.




(a)



(b)

Figure 3.19 – The Yu-Zyner Rounding Scheme [30] (a)
Technology Schematic Obtained with XST(b)

Another rounding algorithm for floating point multiplication is the YuZyner algorithm [109] , which was implemented on Sun ULTRASparc processors and optimized by Even and Seidel [30].The algorithm proceeds as follows, as described in [30]:

1. The SUM and CARRY strings are divided into two parts: the higher part contains the most significant 55 bits, while the lower part the least significant 51 bits.
2. With the low part are computed C – carry bit and S – sticky bit.
3. The higher part is input to a line of half adders and produces the X-Sum on 55 bits an X-Carry on 54 bits.
4. The most significant 52 bits of the X-Sum and X-Carry are inputs into 52 bit compound adder. The results are Y0 and Y1 (which is equal to Y0+1). Based on the most significant bits, a normalization right shift may be done for Y0 and Y1.
5. The last 3 bits of X-Sum and the string formed by the C bit as least significant bit and the last 2 bits of X-Carry as most significant bits are inputs to a 3 bit adder. The result is Z0. In this way, a prediction of the bits needed for RNE (least, guard and sticky) is achieved.
6. The processing of the Z0 is split into two paths: one on the assumption that overflow occurs, while the other on the assumption that overflow will not occur. A rounding decision is taken on both paths, based on the rounding mode, on the S bit, and on the Z0. In the overflow situation, the rounding decision is added on the most significant 2 bits of Z0 and Z0-*ovf* is obtained, while in no overflow situation the rounding decision is added on the most significant 3 bits of Z0 and Z0-*novf* is obtained.
7. An overflow selection decision is taken based the most significant bits of Y0 and Y1 and on the Z0-*novf:*
8. An increment decision is taken based on the OVF signal and on the Z0-*novf* and Z0-*ovf*:
9. The most significant 51 bits of the correct result is selected between the (possible) shifted Y0 and Y1 based on the INC signal
10. The least significant 2 bits of the result is selected based on the OVF signal.

The structure based on this algorithm is depicted in Fig. 3.19.The delay of the hardware scheme which implements this algorithm is 16 LL. However, this algorithm does not require an addition of any kind of constant which depends on the rounding mode, like the injection which is required in the Even-Seidel algorithm or the prediction bit in the Quach algorithm.

Table 3.10 – Comparison between the Three Rounding Algorithms

| Algorithm | Latency | Gate Count |
|---|---|---|
| Even-Seidel | 12 LL | 2090 |
| Quach | 14 LL | 1820 |
| Yu-Zyner | 16 LL | 1806 |

A comparison of the three rounding algorithm is done in Table 3.10. The algorithms are compared for their latency (measured in LL) and for their gate count.

## 3.3 Proposed Multiplier

### 3.3.1 Algorithm

For interval multiplication, a new algorithm which combines the pipelined algorithm presented in [53] (depicted in Fig. 3.1) with the eight products algorithm (3.2) is proposed. The resulted algorithm is presented in Fig. 3.20 [3][4].

1. $p1=RNI(X_{lo} * Y_{lo})$ $p2=RPI(X_{lo} * Y_{lo})$
2. $q1=RNI(X_{lo} * Y_{hi})$ $q2=RPI(X_{lo} * Y_{hi})$
3. $r1=RNI(X_{hi} * Y_{lo})$ $r2=RPI(X_{hi} * Y_{lo})$ $m=min(p1,q1)$ $M=max(p2,q2)$
4. $t1=RNI(X_{hi} * Y_{hi})$ $t2=RPI(X_{hi} * Y_{hi})$ $m=min(m,r1)$ $M=max(M,r2)$
5. $m=min(m,t1)$ $M=max(M,t2)$

$Z_{lo} = m$ $Z_{hi} = M$

Figure 3.20 – Proposed Interval Multiplication Algorithm

As depicted in Fig 3.20, this algorithm requires fourteen floating point operations (four multiplication with RNI, four multiplications with RPI and six comparisons), as in the eight products algorithm. This is the major disadvantage of this algorithm, because it might imply lower performance. However, using two multipliers or one multiplier with two differently rounded results (dual result multipliers), and two floating point comparators (one for minimum and one for maximum) the number of the steps is reduced to only five.

This algorithm presents two major advantages. The first advantage (which is not present at the eight products algorithm) is that it is suitable for pipeline structures. Therefore, pipeline multiplier architectures can be easily incorporated in the overall architecture. The second advantage is the rounding step is performed within the floating point multiplications and not as a separate step. Thus, the very efficient rounding algorithms, like the ones presented in Section 3.2.6 can be applied.

### 3.3.2 Overall Architecture

In order to apply the algorithm described in Fig. 3.20, two floating point multipliers or a dual result multiplier has to be used. Furthermore, two floating point comparators must be used. In order to save area, the dual result multiplier is chosen. Therefore, the proposed architecture for the interval multiplier is composed of a dual result multiplier and two floating point comparators [3][4].

The dual result multiplier is similar to a tree multiplier. The encoding module and the partial product reduction tree are identical to the ones used in a conventional floating point multiplier. The reason is that the two results of the multiplier are obtained from the same pair of operands (the results differ because

the rounding modes differ). Thus, the partial products obtained from the encoding module and the two final partial products obtained form the tree are the same. A new final addition and rounding unit must be used in order to provide the two differently results for the same operands. This unit will be presented in detail in Section 3.3.3.

The overall architecture of the proposed multiplier is depicted in Fig. 3.21. In this figure, the structure of the dual result multiplier is given. In order to perform the proposed algorithm, the two comparators are provided with two feedback paths. The structure of the two comparators is similar as the one presented in [93]. The comparators for the IEEE floating point numbers rely on sign comparators, an exponent subtractor and a mantissa subtractor.

### 3.3.3 Final Addition and Rounding

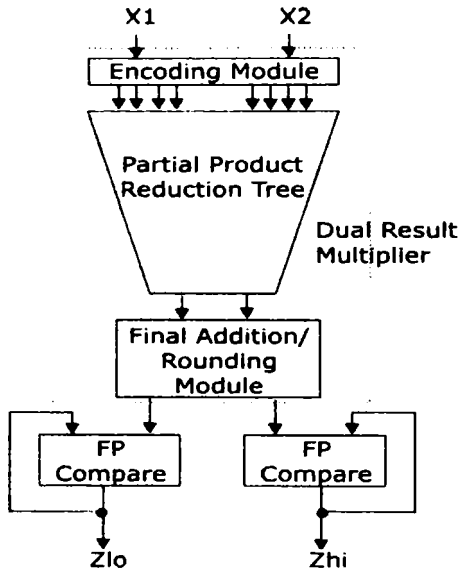
As it can be observed from Fig. 3.21, the overall architecture of the proposed interval multiplier relies on a dual result multiplier. The main features of this type of multiplier is that the encoding module and the partial product reduction tree are kept in common, while some parts of the final addition and rounding module must be duplicated. The goal for such a multiplier is to duplicate as few modules as possible of the rounding scheme, but without affecting the performance. In this section, four rounding schemes are presented for this type of multiplier, three of them being adapted from the ones presented in Section 3.2.6, and a new one is proposed.

In order to apply the Even-Seidel algorithm, two injections have to be applied (0 for RZ and $2^{-52} - 2^{-104}$ for RI). Therefore, the injections cannot be generated as partial products, because in the worst case two trees must be used. Thus, a CSA level must be used for RI (in order to add the required injection), while for RZ no such CSA level is required (because the injection is 0). This will result in two pairs of final partial product (two pairs of SUM and CARRY). The two pairs of SUM and CARRY will need two different rounding schemes (two compound adders, two C[52], S and R circuits, two increment decision modules, four right shifters, but no LSB fixing modules – RNE is not used) [6].

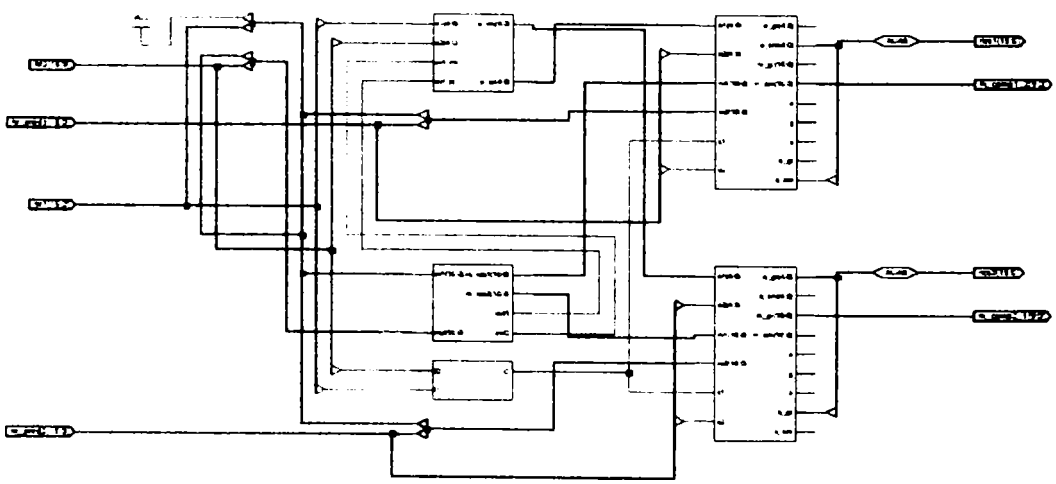
An improvement can be obtained using Quach algorithm, because the two partial products used are identical for both rounding cases. Thus, the carry, round and sticky bits generator is common for both rounding modes. However, because of the prediction, two compound adders must be used (one for RZ when there is no need of injection and one for RI when the prediction bit occurs). The hardware scheme is presented in Fig. 3.22. The duplicated components are the compound adder, the increment decision block (which is much simpler) the right-shifter and the selection multiplexers [6]. The two rows of half adders do not need the duplication, because the LSB of the first row sum output is used as Lx for RZ.

The Yu-Zyner scheme avoids the need of a prediction or injection. This is possible because the most significant carry from the lower part (C[53]) is added with bits from the higher part. In this way, the compound adder has the same inputs (and thus the same results) independent of the rounding mode (as in Even-Seidel or in Quach algorithm). In this way, the same compound adder can be used for both RI and RZ. The results of the compound adder (Y0 and Y1) are processed twice, for each of the two rounding modes. Thus, a major cost saving is achieved, by using a single compound adder. Furthermore, a single carry and sticky generator is used. The only duplicated components are rounding decision modules, the

selection decision circuit and the multiplexers used for selecting the final result [6]. In Fig.3.23 the hardware scheme is presented.




(a)



(b)

Figure 3.21 – Proposed Interval Multiplier Architecture [4] (a)
RTL Schematic Obtained with XST (b)

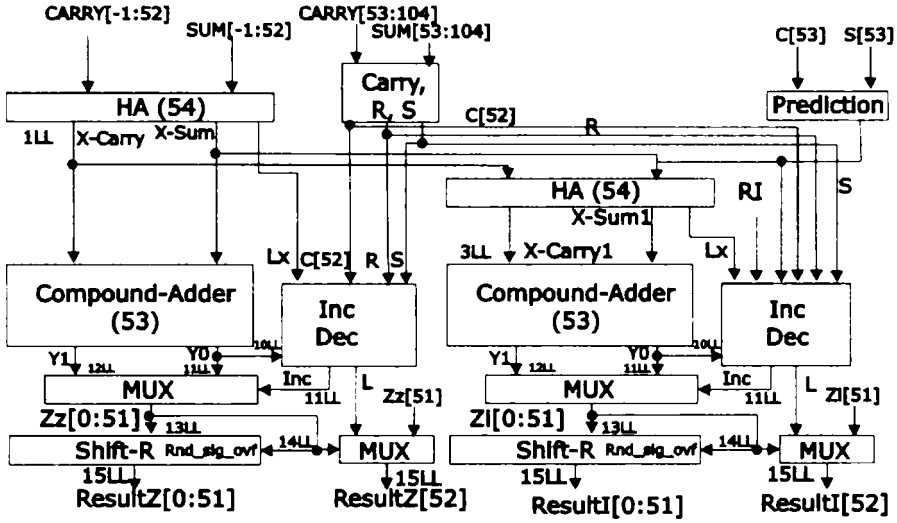
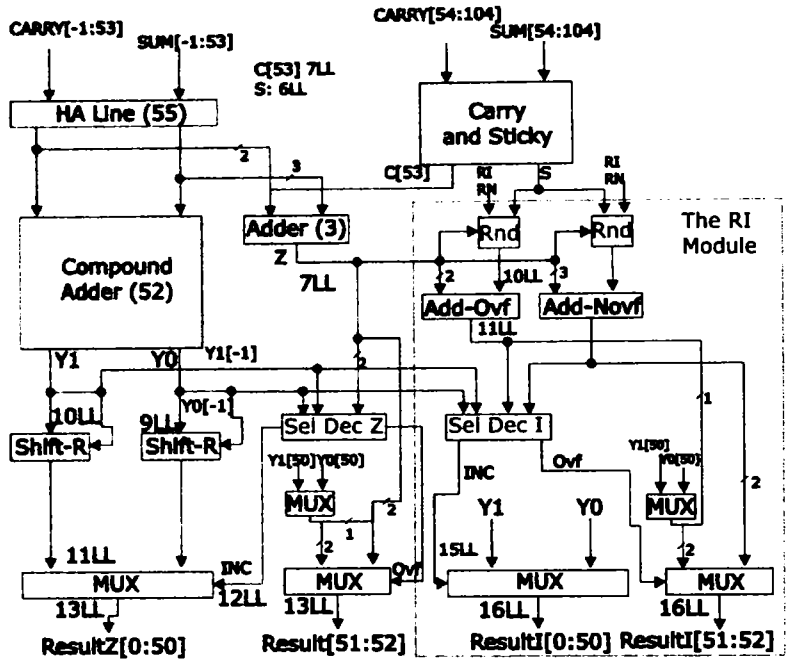


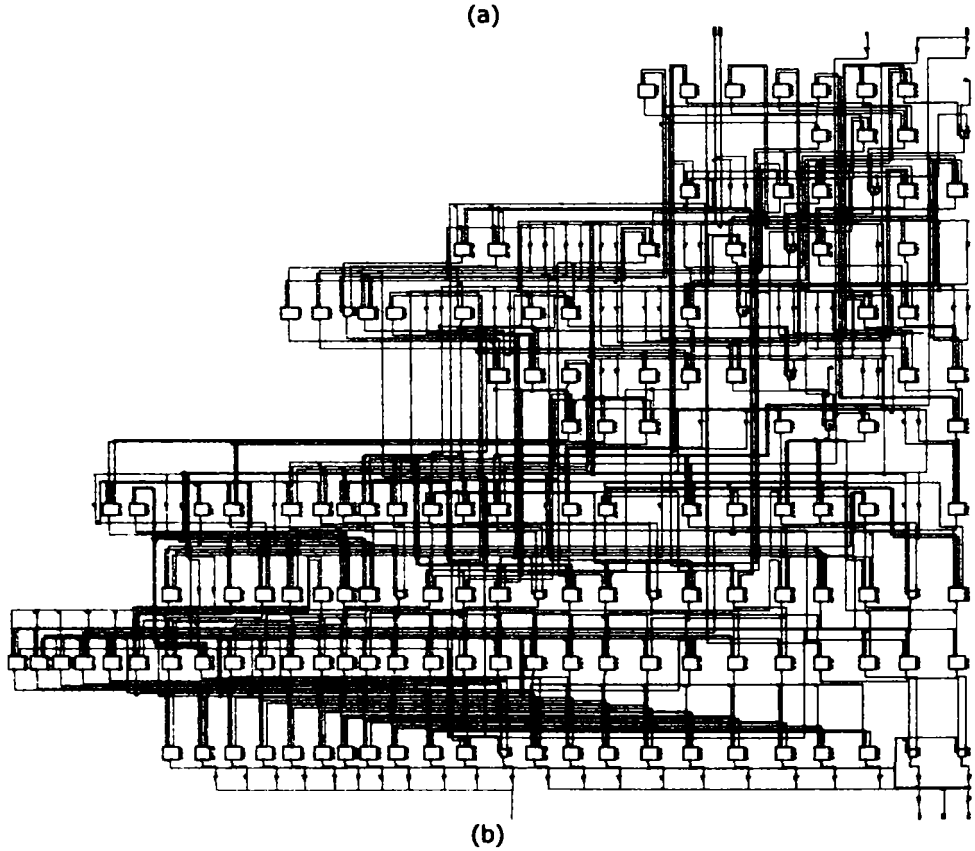

Figure 3.22 – Rounding Scheme for Dual Result Multiplier Based on Quach Algorithm[6]

These three schemes were adapted from the ones used for conventional floating point multiplication, which were presented in Section 3.2.6. A new scheme for dual result multipliers is proposed which can be used only for interval arithmetic. This scheme is based on the Yu-Zyner algorithm. However, the proposed algorithm is simpler, because it does not into account RNE, which is not need in interval arithmetic. The proposed algorithm proceeds as follows [6]:

1. The SUM and CARRY strings are separated into a high part consisting of 54 bits and a low part of 52 bits.
2. The lower part is used to compute C[52] and the S bit.
3. The high part is input to a line of half adders. The carry string (consisting of 53 bits) and the most significant 53 bits of the sum string are inputs to a compound adder. The results of the compound adder are Y0 and Y1 (Y0+1).
4. The LSB bit of the sum string (Lx) is added with C[52] for RZ, and with C[52] and S for RI. The sum bit is the ResultZ'[52] for RZ and ResultI'[52] for RI. These two bits will be the LSB of the two results in case no overflow occurred. The carry bits will be CZ[51] for RZ and CI[51] for RI. The two carry bits will determine which of the Y0 and Y1 will be selected for each rounding case.
5. A selection between Y0[-1] and Y1[-1] ( the overflow bits) is done for each rounding case, also based on the CZ[51] and CI[51]. The OvfZ signal for RZ and OvfI signal for RI are generated.
6. The most significant 52 bits of the result are selected in both cases between the normalized Y0 and Y1 based on the CZ[51] and CI[51]. The LSB of the result is selected in both cases based on the OvfZ and OvfI.

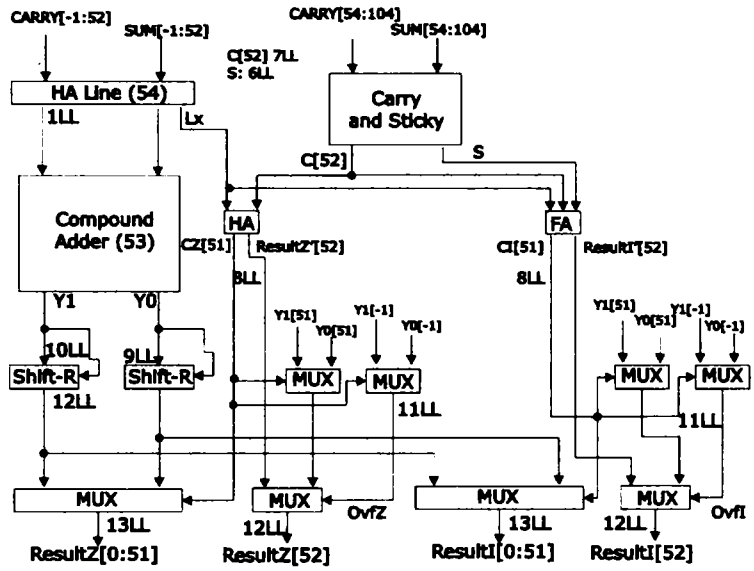



(a)

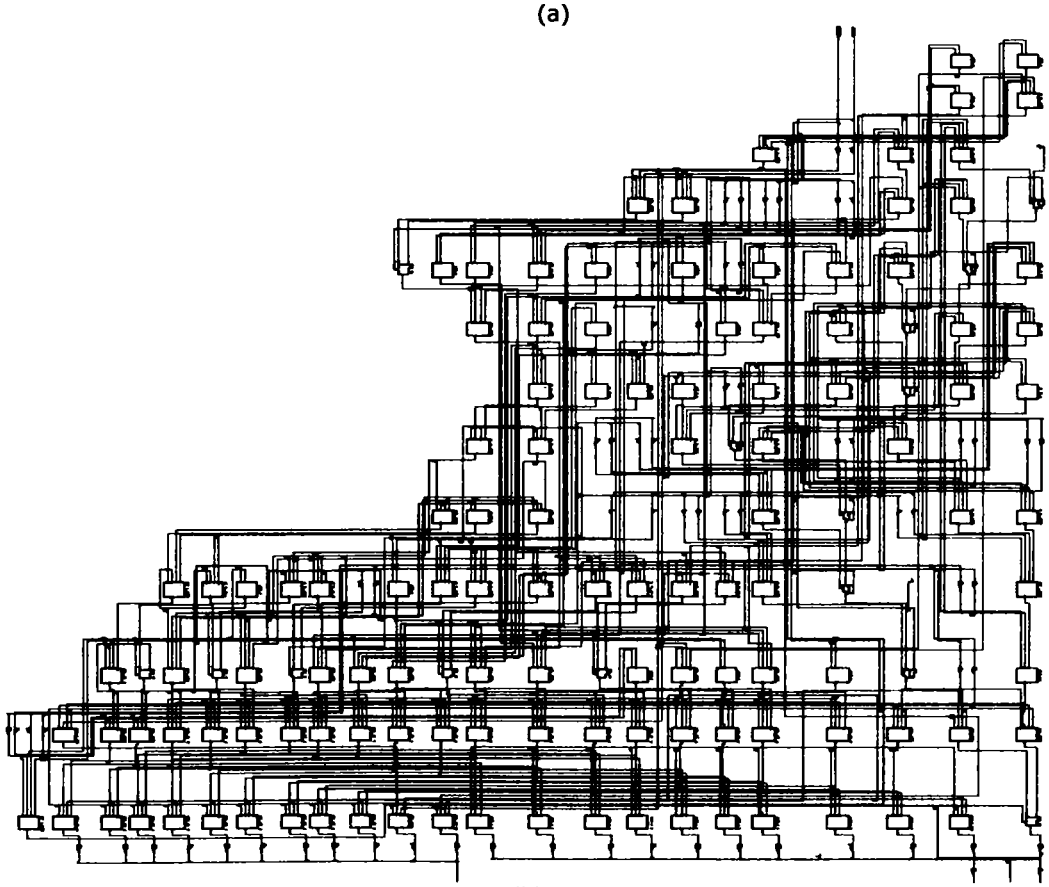


(b)

Figure 3.23 – Rounding Scheme for Dual Result Multiplier Based on Yu-Zyner Algorithm[6]
Technology Schematic Obtained with XST(b)




(a)



(b)

Fig 3.24 – Proposed Rounding Scheme for Dual Result Multipliers[6] (a) Technology Schematic Obtained with XST(b)

The hardware scheme is presented in Fig. 3.24. Furthermore, this algorithm uses a single compound adder and a single carry and sticky generator, which means a reduced cost. The drawback of this rounding scheme is that it doesn't implements RNE. Therefore, it cannot be used for conventional floating point multiplication, but only for interval arithmetic.

### 3.3.4 Interval Set Operations

The two floating point comparators can also be used for interval set operations, like interval hull, interval inclusion or interval intersection. These operations are of great importance for interval arithmetic (for example, the interval intersection is used in each iteration in the interval Newton's method). In Fig. 3.24, algorithms for interval hull and interval intersection are given [1].

| | |
|---|---|
| $Z_{lo} = \max(X_{lo}, Y_{lo})$ | $Z_{lo} = \min(X_{lo}, Y_{lo})$ |
| $Z_{hi} = \min(X_{hi}, Y_{hi})$ | $Z_{hi} = \max(X_{hi}, Y_{hi})$ |
| If $Z_{lo} < Z_{hi}$ then | $R = [Z_{lo}, Z_{hi}]$ |
| $R = [Z_{lo}, Z_{hi}]$ | |
| else $R = \varnothing$ | |
| a) | b) |

Figure 3.25 – Algorithm for Interval Intersection (a) and Interval Hull (b) [1]

In order to use the architecture proposed in Fig. 3.20 a series of multiplexers are introduced before the two floating point comparators. The latency penalty is minimal (only 1 LL). However, an increase in functionality for the proposed architecture is obtained [4].

## 3.4 Evaluation

### 3.4.1 Cost Analysis

The cost was estimated using gate count as metric. The gate count metric can provide useful information about the size of the proposed circuits, although this metric is not the most relevant for VLSI technology (as we have seen in 3.2.6, the wired tracks are also important for VLSI circuits). However, this metric is technology independent. Gate count was also used in [88]. As in [88], the basic gates considered are AND/NAND, OR/NOR, NOT, XOR and the 2-input multiplexer.

In table 3.11 the cost of the proposed architecture for IEEE double precision numbers is presented. The dual result multiplier has different configurations, depending on the partial products reduction tree and the final rounding and addition module. The encoding used was the Booth 2.

Table 3.12 presents the cost of conventional floating point multipliers which use for encoding Booth 2 algorithm. Different types of trees and rounding and addition modules are also used.

Table 3.11 - Gate Count for Proposed Architecture

| | Wallace Tree | Overturned Staircase | Balanced Delay | Binary Tree |
|---|---|---|---|---|
| Even-Seidel | 23286 | 25032 | 23868 | 24084 |
| Quach | 22030 | 24046 | 22612 | 22828 |
| Yu-Zyner | 21000 | 23016 | 21580 | 21798 |
| Proposed | 20938 | 22954 | 21518 | 21736 |

Table 3.12 - Gate Count for Conventional Floating Point Multiplier

| | Wallace Tree | Overturned Staircase | Balanced Delay | Binary Tree |
|---|---|---|---|---|
| Even-Seidel | 18800 | 20815 | 19238 | 19597 |
| Quach | 18630 | 20645 | 19068 | 19427 |
| Yu-Zyner | 18616 | 20631 | 19054 | 19413 |

As we can see from table 3.11, the multipliers which have as final rounding and addition modules the modified Yu-Zyner and the proposed solution have a significantly smaller gate count then the ones based on the Quach algorithm and the Even-Seidel algorithms. In table 3.13, is a presentation of the gate counts for these four solutions for final rounding and addition.

Table 3.13 - Gate Counts for Final Addition and Rounding Units

| Algorithm | Gate Count |
|---|---|
| Even-Seidel | 4168 |
| Quach | 2912 |
| Yu-Zyner | 1882 |
| Proposed | 1820 |

Another important aspect is that the proposed interval multiplier architecture requires from 12% up to 27% more gates for double precision numbers compared to the conventional floating point multipliers. The reasons for this increase are:

- The two floating point comparators
- The increase cost of the final rounding and addition module.
- A more complex exponent update module.

The comparison with a conventional floating point multiplier is relevant, because this unit is the backbone of all other implementations for interval multiplication, like the sign examining algorithms or the eight products [94][96][107].

### 3.4.2 Performance Estimates

The latency was estimated using the independent technology metric logic levels (LL). This metric was also used for latency estimation in [16][30][88][89]. In table 3.14 are presented the latency estimates for the proposed architecture, which has uses a Booth 2 encoder module and a Wallace tree as the partial product reduction tree. Table 3.15 presents the latency of a conventional floating point multiplier, with the same type of encoder module and partial product reduction tree.

The latency was estimated for different schemes used for final addition and rounding modules.

Table 3.14 Latency Estimates for Proposed Architecture

| Module | Even-Seidel | Quach | Yu-Zyner | Proposed |
|---|---|---|---|---|
| Encoder | 5 | 5 | 5 | 5 |
| Partial Product Tree | 14 | 14 | 14 | 14 |
| Rounding Scheme | 14 | 15 | 16 | 13 |
| Comparator | 9 | 9 | 9 | 9 |
| Overall | 42 | 43 | 44 | 41 |

Table 3.15 Latency Estimates for Conventional Floating Point Multiplication

| Module | Even-Seidel | Quach | Yu-Zyner |
|---|---|---|---|
| Encoder | 5 | 5 | 5 |
| Partial Product Tree | 14 | 14 | 14 |
| Rounding Scheme | 14 | 15 | 16 |
| Overall | 33 | 34 | 35 |

As it can be observed in Table 3.14 the proposed solution for rounding scheme has the best latency. This is due to a much simpler logic of the proposed scheme, because the proposed solution does not perform rounding to nearest even (which is the main drawback of the proposed solution).

The latency of the proposed architecture is higher compared to a conventional floating point multiplier due to the floating point comparator (which lies in the critical data path). If the proposed architecture would be used for a conventional floating point multiplication (using Even-Seidel, Quach or Yu-Zyner rounding schemes) the latency would be almost the same (for Quach and Yu-Zyner it would be equal, while using Even-Seidel a 2 LL increase in latency can be observed).

Based on the obtained latency, a four stage pipeline architecture for the proposed multiplier can be used, with a clock cycle suitable for 12 LL (without considering the delays of the pipeline registers or the clock skew). The conventional floating point multiplier would have a three stage pipeline. In this case, an interval multiplication would require 7 clock cycles. A sign-examining interval multiplication using a conventional floating point multiplier would require 4 clock cycles in the best case, and 8 clock cycles in the worst case – when both intervals contain zero (6 clock cycles for the 4 multiplications and 2 clock cycles for the 2 comparisons). Therefore, the proposed architecture presents a better worst case performance then the sign examining algorithm.

### 3.4.3 Synthesis Results

Seven multiplier designs were implemented in Xilinx Virtex-4 family FPGA technology using the Xilinx ISE 10.1 Webpack and synthesized with Xilinx Synthesis Tool (XST). All seven designs use the Booth radix 4 (Booth 2) algorithm in the encoding module and the Wallace tree as the partial product reduction tree. A 12-bit

compound adder was used for the compound addition of the most significant halves of the final partial products, while the carry generator was designed for 11 bits. Therefore, in case of the Yu-Zyner rounding scheme, which uses 1-bit smaller compound adder and carry generator, the cost of the entire rounding scheme may be smaller. The following seven multiplier results are:

- Interval multiplication unit using the proposed addition and rounding unit
- Interval multiplication unit using the modified Yu-Zyner addition and rounding scheme
- Interval multiplication unit using the modified Quach addition and rounding scheme
- Interval multiplication unit using the modified Even-Seidel addition and rounding scheme
- Floating point multiplication unit using the Yu-Zyner addition and rounding scheme
- Floating point multiplication unit using the Quach addition and rounding scheme
- Floating point multiplication unit using the Even-Seidel addition and rounding scheme

The latency and cost results for both interval designs and conventional floating point multipliers are presented in Table 3.16. Figure 3.27 presents the relative cost of the interval multipliers compared to the conventional floating point multiplication units.

Table 3.16 – Latencies and Cost for the Interval and Floating Point Multipliers

| Interval Multipliers | | | Floating Point Multipliers | | |
|---|---|---|---|---|---|
| Rounding Scheme | Latency (ns) | Cost (LUT-4) | Rounding Scheme | Latency (ns) | Cost (LUT-4) |
| Even-Seidel | 19.529 | 495 | Even-Seidel | 15.365 | 336 |
| Quach | 20.397 | 452 | Quach | 15.959 | 327 |
| Yu-Zyner | 20.991 | 402 | Yu-Zyner | 15.366 | 320 |
| Proposed | 20.398 | 373 | Proposed | - | - |

The results obtained by synthesis are similar to the estimations presented in sections 3.4.1. Regarding the cost, an increase is observed due the two floating point comparators and the additional cost in the modified rounding scheme. The cost of the interval multipliers for half precision ranges from 25% (in case of Yu-Zyner rounding scheme) to 45% (in case of Even-Seidel rounding scheme). The reason for this is that in case of half precision the weight of the encoder scheme (124 LUT-4) and the partial product reduction tree (118 LUT-4) is smaller than in case of double precision. Regarding the latency, a four pipeline stage for the interval multipliers can be designed (as the conventional floating point multipliers can be designed for three pipeline stages). An interval multiplication can be executed in seven clock cycles using the proposed algorithm.

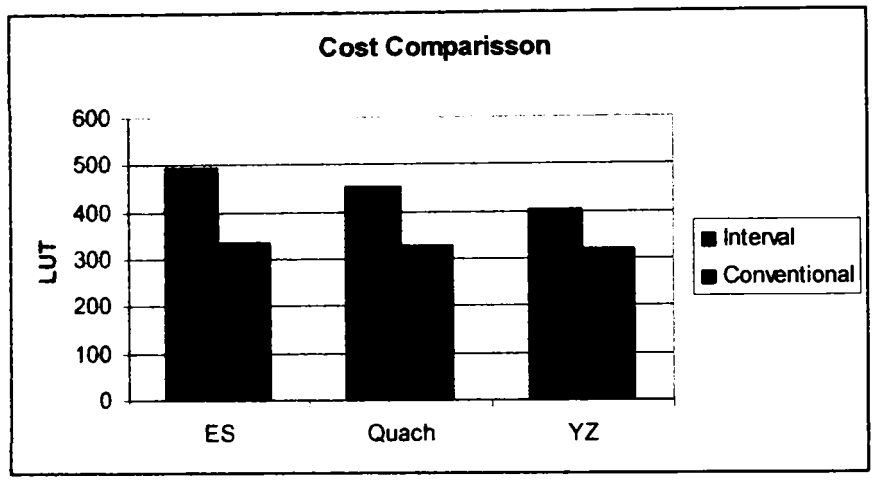



Figure 3.26 – Cost Comparison between Interval and Conventional Floating Point Multipliers

The results for the four interval rounding schemes are presented in Table 3.18 depicts the results (both latency and cost) obtained only for the four addition and rounding schemes used for the interval multipliers designs. Figure 3.28 depict cost*latency product (measured in ns*LUT-4) for the four interval rounding schemes.

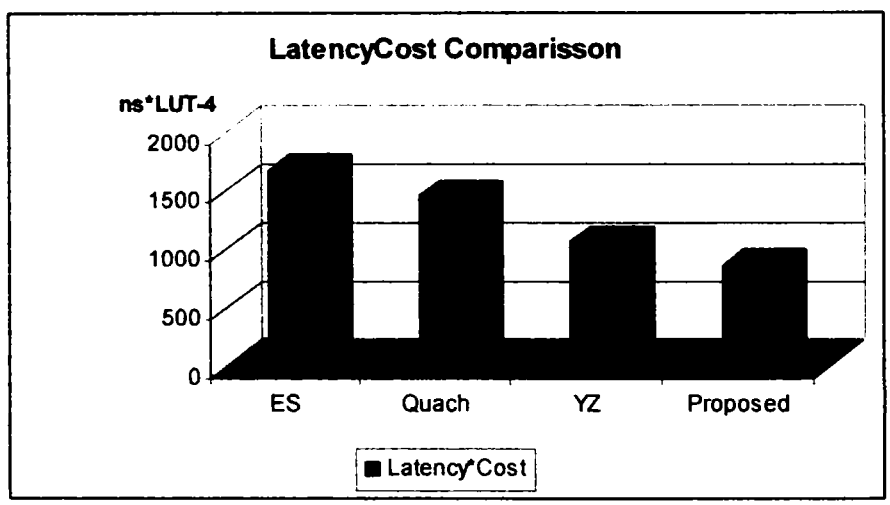



Figure 3.27 – Latency*Cost Comparison of the Four Interval Rounding Schemes

Table 3.17 – Latencies and Cost for the Interval Rounding Schemes

| Rounding Scheme | Latency (ns) | | | Cost (LUT-4) |
|---|---|---|---|---|
| | Logic | Route | Total | |
| Modified ES | 2.192 | 5.218 | 7.410 | 226 |
| Modified Quach | 2.342 | 6.796 | 9.138 | 159 |
| Modified YZ | 2.349 | 5.675 | 8.024 | 132 |
| Proposed | 1.911 | 5.265 | 7.176 | 120 |

As it can be observed from both Table 3.18 and Fig. 3.27, the proposed rounding scheme has the smallest latency and the smallest cost (as it was shown in sections 3.4.1 and 3.4.2). Very advantageous from the (latency*cost) perspective is the modified Yu-Zyner rounding scheme. The Even-Seidel also has a relative low latency (the second lowest), has a very high cost compared to the other three rounding schemes (almost double compared to the proposed one).

## 3.5 Summary

In this chapter, an overview of the most important algorithms and implementations for interval multiplication is presented in Section 3.1. Two techniques are of interest: the pipelined basic algorithm and the eight products algorithm. Emphasize is on these two because the proposed algorithm is a combination of these two.

In Section 3.2, a detailed description of the floating point multiplication is realized. We focused on the mantissa multiplication and rounding. Details of each major module of the mantissa multiplication are given: the encoding module, the partial product reduction tree and the final addition and rounding module. These modules constitute the building blocks of the proposed multiplier.

Section 3.3 is dedicated to the proposed solutions for interval multiplication. The main contributions regarding interval multiplications are presented in this section and are:

- An algorithm for interval multiplication, based on the basic pipeline algorithm and on the eight products algorithm
- A multiplier architecture for the proposed architecture and based on the dual result multiplier. Furthermore, the proposed architecture can be used for interval set operations.
- Modifications of the Even-Seidel, Yu-Zyner and Quach rounding schemes for dual result multipliers. Furthermore, a new rounding scheme for interval arithmetic is proposed.

Estimates for cost and performance were done and are presented in Section 3.4. The cost estimates show an increase form 12% up to 27% (double precision) and 25% up to 45% (half precision) of the proposed architecture compared to a conventional floating point multiplier. The main reasons for this increase are the two floating point comparators. However, this two floating point comparators can also be used for interval set operations, thus increasing the functionality of the proposed design. Performance estimates show an increase in the worst case of the proposed multiplier compared to an interval multiplier based on a sign examining algorithm. Furthermore, the proposed multiplier can be used for conventional floating point multiplication (with Even-Seidel, Quach and Yu-Zyner rounding schemes) with the same performance of a standard tree based floating point multiplier. Last, but not least, the proposed rounding scheme for interval arithmetic has an improved latency with respect to the three other rounding schemes, with a cost similar to the modified Yu-Zyner scheme (which has the smallest gate count of the three modified rounding schemes).

# 4. Floating Point Divide-Add Fused for Interval Newton's Method

## 4.1 Considerations on the Floating Point Divide-Add Fused

A detailed analysis on the floating point division and its impact on the overall performance in a floating point system were performed in [70] by Oberman, using the SPEC FPU benchmarks. The analysis showed that the division has a frequency of about 3% from all floating point operations (compared to the 55% for floating point addition and 39% for floating point multiplication). Hence, floating point division is a scarce operation, and therefore, the hardware designs regarding division should be oriented to lower cost rather than higher performance. Another analysis performed in [69][70] showed that the 29% of the results of floating point division are used as operands in addition. Thus, the percentage in conventional floating point systems of the divide-add fused (division followed by subtraction) would be less than 1%. Instead, multiply-add fused counts more than 12% from all floating point instruction. Thus, an implementation of the multiply-add fused is convenient even for general purpose processors. This is not the case for the divide-add fused, which is a very rare operation. Therefore, to the best of my knowledge, no hardware unit for divide-add fused has been implemented.

Also in [70], the authors conclude that a floating point divide-add fused would be convenient if the percentage of this operation would be similar to the one of floating point multiply-add fused. This could be the case in interval arithmetic, mainly because of the Newton's interval method. This algorithm is a powerful tool for nonlinear equation solving [24][34][82][102][104], with applicability in a wide range of fields such as chemical engineering, computer graphics, robotics and control theory, computer-aided design [47]. This algorithm is based on a division followed by subtraction, thus a divide-add fused unit would be convenient for this method.

## 4.2 Interval Newton's Method

### 4.2.1 Standard Interval Newton's Method

The standard interval Newton's method can be applied to functions which are continuous and monotonous on the specified interval. Given a function $f$ and a starting interval $X = [X_{lo}, X_{hi}]$, the algorithm relies on the following iteration [24][33][34][48][102][104]:

$$X_0 = X \tag{4.1}$$

$$X_{i+1} = \left( m(X_i) - \frac{f(m(X_i))}{f'(X_i)} \right) \cap X_i$$

Where $m(X_i)$ is one point which belongs to the interval $X_i$ (usually is considered the midpoint of the interval). The termination criteria for this algorithm is represented either the case of the $\varnothing$ interval (case where there is no solution in the $X = [X_{lo}, X_{hi}]$ for the equation $f(x) = 0$), either the obtained interval has the desired accuracy ($|X_{lo-i} - X_{hi-i}| < \varepsilon$)[102][104]. The graphic representation of the interval Newton's method is depicted in the Fig. 2.1.

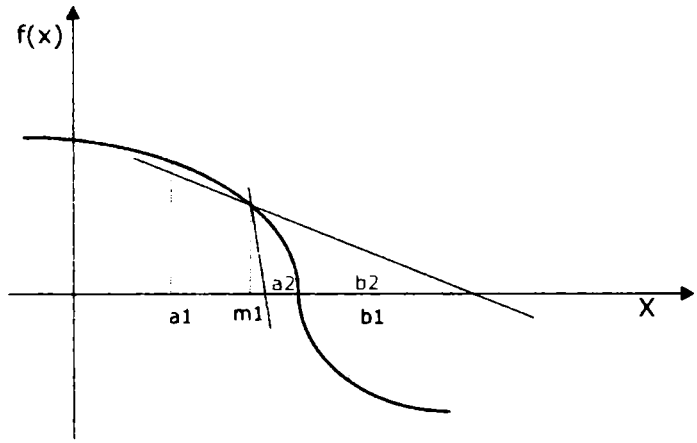



Figure 4.1 – Graphic Representation of Newton's Interval Method. [$a1$,$b1$] represent the initial interval, $m1$ is the midpoint, and [$a2$,$b2$] represent the result interval after the iteration (in this case $b1$=$b2$)

This algorithm makes use of the interval division. The interval division is defined as follows [53][95][94]:

$$X/Y = [X_{lo}, X_{hi}] / [Y_{lo}, Y_{hi}] = [X_{lo}, X_{hi}] * 1/[Y_{lo}, Y_{hi}], \quad (2.2)$$

Undefined for $0 \in [Y_{lo}, Y_{hi}]$

As a direct consequence of (2.2), the standard interval Newton's method cannot be applied for intervals where the function has a local minimum or maximum on the initial interval (or the derivative has a zero on the specified interval) [32][47]. Thus, this method can be used for functions which have maximum one root on the specified interval. In order to extend this algorithm to intervals where the function has a local minimum or maximum (thus, having more than one root), an analysis of the interval division by an interval containing zero must be performed.

### 4.2.2 Interval Division by a Zero Containing Interval

The interval division is defined in (2.2). It resembles the interval multiplication, thus, it would be expected to be as difficult as it, requiring ten

floating point operations in order to perform it [49]. As in the case of interval multiplication, sign examining for interval division can be performed [95]. Unlike the interval multiplication, due to $0 \in [Y_{lo}, Y_{hi}]$, the interval division consists of only six cases, each requiring only two floating point divisions, as presented in Table 4.1

Table 4.1 – Interval Division [49][95][94]

| | $[X_{lo}, X_{hi}]$ | $[Y_{lo}, Y_{hi}]$ | Result |
|---|---|---|---|
| 1 | $X_{lo} > 0$ | $Y_{lo} > 0$ | $\left[X_{lo}/Y_{hi}, X_{hi}/Y_{lo}\right]$ |
| 2 | $X_{lo} > 0$ | $Y_{hi} < 0$ | $\left[X_{hi}/Y_{hi}, X_{lo}/Y_{lo}\right]$ |
| 3 | $X_{lo} < 0 < X_{hi}$ | $Y_{lo} > 0$ | $\left[X_{lo}/Y_{lo}, X_{hi}/Y_{lo}\right]$ |
| 4 | $X_{lo} < 0 < X_{hi}$ | $Y_{hi} < 0$ | $\left[X_{hi}/Y_{hi}, X_{lo}/Y_{hi}\right]$ |
| 5 | $X_{hi} < 0$ | $Y_{lo} > 0$ | $\left[X_{lo}/Y_{lo}, X_{hi}/Y_{hi}\right]$ |
| 6 | $X_{hi} < 0$ | $Y_{hi} < 0$ | $\left[X_{hi}/Y_{lo}, X_{lo}/Y_{hi}\right]$ |

In order to extend the interval Newton's method to intervals where the function has a local minimum or maximum, the interval division by an interval containing zero must be considered. One such analysis was performed by Ulrich Kulisch [53][54] and constitutes a proposal for the future IEEE 1788 standard for interval arithmetic. This analysis is done according to the positioning of the zero within the interval.

**Case 1.** $Y = [0, 0] = 0$

This case can be regarded as follows: the result of the division $X/Y$ can be viewed as the solution of the following equation $0 * x = a, a \in X$. Two sub-cases do appear:

- $0 \in X$, which leads to the solution $(-\infty; +\infty)$ ( $(-\infty; +\infty)$ for $0 * x = 0$ and $\varnothing$ for $0 * x = a, a \neq 0$ )
- $0 \notin X$, which leads to the solution $\varnothing$

**Case 2.** $Y = [Y_{lo}, 0]$

In this case, the $Y = [Y_{lo}, 0]$ interval is replaced with the $Y' = [Y_{lo}, -\varepsilon]$. Depending on the dividend $X = [X_{lo}; X_{hi}]$ we have the following sub-cases:

- $X_{hi} < 0$. The result of the $X/Y$, division is $\left[X_{hi}/Y_{lo}, X_{lo}/-\varepsilon\right]$. Therefore, the result of the division $X/Y$ is $\lim_{\varepsilon\to 0}\left[X_{hi}/Y_{lo}, X_{lo}/-\varepsilon\right] = \left[X_{hi}/Y_{lo}, +\infty\right)$
- $X_{lo} > 0$. The result of the $X/Y$, division is $\left[X_{hi}/-\varepsilon, X_{lo}/Y_{lo}\right]$. Therefore, the result of the division $X/Y$ is $\lim_{\varepsilon\to 0}\left[X_{hi}/-\varepsilon, X_{lo}/Y_{lo}\right] = \left(-\infty, X_{lo}/Y_{lo}\right]$.

**Case 3.** $Y = [0, Y_{hi}]$

In this case, the $Y = [0, Y_{hi}]$ interval is replaced with the $Y' = [\varepsilon, Y_{hi}]$. Depending on the dividend $X = [X_{lo}; X_{hi}]$ we have the following sub-cases:

- $X_{hi} < 0$. The result of the $X/Y$, division is $\left[X_{lo}/\varepsilon, X_{hi}/Y_{hi}\right]$. Therefore, the result of the division $X/Y$ is $\lim_{\varepsilon\to 0}\left[X_{lo}/\varepsilon, X_{hi}/Y_{hi}\right] = \left(-\infty, X_{hi}/Y_{hi}\right]$
- $X_{lo} > 0$. The result of the $X/Y$, division is $\left[X_{lo}/Y_{hi}, X_{hi}/\varepsilon\right]$. Therefore, the result of the division $X/Y$ is $\lim_{\varepsilon\to 0}\left[X_{lo}/Y_{hi}, X_{hi}/\varepsilon\right] = \left[X_{lo}/Y_{hi}, +\infty\right)$.

**Case 4.** $Y = [Y_{lo}, Y_{hi}], Y_{lo} < 0 < Y_{hi}$

In this case, the $Y = [Y_{lo}, Y_{hi}]$ interval is replaced with the $Y' = [Y_{lo}, -\varepsilon] \cup [\varepsilon, Y_{hi}]$. In this case we use the inclusion-isotony property of interval arithmetic ($A \subset B \wedge C \subset D \Rightarrow AopC \subset BopD$)[54]. Depending on the dividend $X = [X_{lo}; X_{hi}]$ we have the following sub-cases:

- $X_{hi} < 0$. The result of the $X/Y$, division is $\left[X_{hi}/Y_{lo}, X_{lo}/-\varepsilon\right] \cup \left[X_{lo}/\varepsilon, X_{hi}/Y_{hi}\right]$. Therefore, the result of the division $X/Y$ is $\lim_{\varepsilon\to 0}\left[X_{hi}/Y_{lo}, X_{lo}/-\varepsilon\right] \cup \left[X_{lo}/\varepsilon, X_{hi}/Y_{hi}\right] = \left(-\infty, X_{hi}/Y_{hi}\right] \cup \left[X_{hi}/Y_{lo}, +\infty\right)$
- $X_{lo} > 0$. The result of the $X/Y$, division is $\left[X_{hi}/-\varepsilon, X_{lo}/Y_{lo}\right] \cup \left[X_{lo}/Y_{hi}, X_{hi}/-\varepsilon\right]$. Therefore, the result of the division $X/Y$ is $\lim_{\varepsilon\to 0}\left[X_{hi}/-\varepsilon, X_{lo}/Y_{lo}\right] \cup \left[X_{lo}/Y_{hi}, X_{hi}/-\varepsilon\right] = \left(-\infty, X_{lo}/Y_{lo}\right] \cup \left[X_{lo}/Y_{hi}, +\infty\right)$.

Table 4.2 - Interval Division by an Interval Containing Zero [54]

| | $[X_{lo}, X_{hi}]$ | $[Y_{lo}, Y_{hi}]$ | Result |
|---|---|---|---|
| 1 | $0 \in [X_{lo}, X_{hi}]$ | * | $[-\infty, +\infty]$ |
| 2 | * | $[0,0]$ | $\varnothing$ |
| 3 | $X_{lo} > 0$ | $[Y_{lo}, 0]$ | $\left(-\infty, X_{lo}/Y_{lo}\right]$ |
| 4 | $X_{lo} > 0$ | $[0, Y_{hi}]$ | $\left[X_{lo}/Y_{hi}, +\infty\right)$ |
| 5 | $X_{lo} > 0$ | $0 \in [Y_{lo}, Y_{hi}]$ | $\left(-\infty, X_{lo}/Y_{lo}\right] \cup \left[X_{lo}/Y_{hi}, +\infty\right)$ |
| 6 | $X_{hi} < 0$ | $[Y_{lo}, 0]$ | $\left[X_{hi}/Y_{lo}, +\infty\right)$ |
| 7 | $X_{hi} < 0$ | $[0, Y_{hi}]$ | $\left(-\infty, X_{hi}/Y_{hi}\right]$ |
| 8 | $X_{hi} < 0$ | $0 \in [Y_{lo}, Y_{hi}]$ | $\left(-\infty, X_{hi}/Y_{hi}\right] \cup \left[X_{hi}/Y_{lo}, +\infty\right)$ |

Table 4.2 presents all the cases for interval division when the divisor contains zero. This way, an extension of the standard interval Newton's method can be obtained for any continuous functions on the selected interval.

### 4.2.3 Extended Interval Newton's Method

In order to extend the interval Newton's method to function whose derivates have a zero on the selected interval, it must be taken into account the division by an interval which contains zero. Without losing from generality, we will consider the case of $0 \in f'(X)$, which corresponds to the 5th and 8th rows from the Table 2.3.

Starting from the iteration described in (2.1), after the division and the subtraction the following set is obtained [53]:

$$X'_{i+1} = (-\infty, X_{lo-i+1}] \cup [X_{hi-i+1}, +\infty) \tag{4.3}$$

After the intersection with the previous interval $X_i = [X_{lo-i}, X_{hi-i}]$, four cases might be obtained [53]:

$$X'_{i+1} \cap X_j = \begin{cases} \varnothing, \\ [X_{lo-i}, X_{lo-i+1}] \\ [X_{hi-i+1}, X_{hi-i}] \\ [X_{lo-i}, X_{lo-i+1}] \cup [X_{hi-i+1}, X_{hi-i}] \end{cases} \tag{4.4}$$

The first case corresponds to the no solution situation. The second corresponds to the case when the solutions are in the interval $[X_{lo-i}, X_{lo-i+1}]$, while the third corresponds to case when the solutions are found in the interval $[X_{hi-i+1}, X_{hi-i}]$. The fourth case represents the case when the equation has at least two solutions,

one in the $[X_{lo-i}, X_{lo-i+1}]$ and the other in $[X_{hi-i+1}, X_{hi-i}]$. The graphic representation of the fourth case is depicted in Fig. 2.2. Therefore, by using the extended interval Newton's method, equations which have multiple solutions on a specified interval can also be solved.

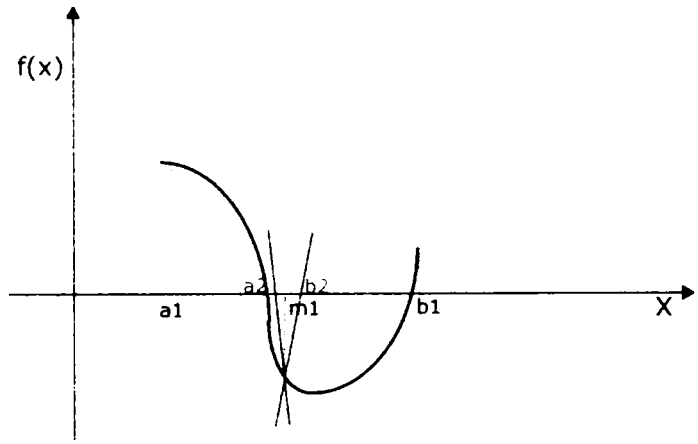



Figure 4.2 - Graphic Representation of the Interval Newton's Method when the Function Has a Local Minimum

Therefore, by employing division by an interval containing zero, the interval Newton's method can be used for every continuous function on the selected interval. This way, a very powerful numerical algorithm for solving nonlinear equations is developed.

### 4.2.4 Discussion

The interval Newton's method has been proven to provide accurate and, more important, guaranteed results [24][82]. This is quite in contrast to the conventional Newton's method, which is known that it fails in some conditions to converge to the root (having a false convergence). Furthermore, the extended Newton's algorithm can be used for functions with multiple roots, providing clear indications about the number of them. Another advantage compared to its classical counterpart is that it indicates from the first iteration the inexistence of any roots. Thus, computation time is saved from finding roots that do not exist.

Regarding the performance, this method has quadratic convergence [47]. Provided fast operations within the method (function evaluation, fast division, fast subtraction), the interval Newton's method becomes a numerical algorithm which can be used for high performance applications (like rendering in graphical computations [84]).

Furthermore, this algorithm can be extended for the nonlinear systems of equations. For this purpose, matrix operations are needed, and the Jacobian matrix is used in order to find the solution of the nonlinear systems [103][105].

For this algorithm, a dedicated operator – the interval Newton operator - has been defined for an interval $X$ based on a function $f$ [33]:

$$N(X,f)=\left(m(X)-\frac{f(m(X))}{f'(X)}\right) \tag{4.5}$$

This operator comprises of a function evaluation on an interval, on function evaluation in a point, an interval division and subtraction. The two function evaluations are dependent on the chosen function. However, the division followed by the subtraction remains as standard operation, no matter the function.

The interval Newton's method is the one of the most important, or maybe the most important, numerical method of the interval arithmetic. Every field of applications which require solving nonlinear equations or systems of nonlinear equations can benefit from interval Newton's method. Applications of this method have been developed in chemical engineering [47], computer graphics and visualization (ray tracing, surface intersections)[84], computer-aided design [32], control theory and robotics[24], etc.

## 4.3 Floating Point Multiply-Add Fused

### 4.3.1 Consideration on the Floating Point Multiply-Add Fused

The multiply-add fused floating point unit represents the basis for the proposed divide-add fused circuit due to the following reasons:

- One of the two involved operations is represented by the addition/subtraction
- The floating point division is similar to the floating point multiplication and has the same precedence compared to the addition/subtraction
- The rich experience in the design of the floating point multiply-add fused

Multiply-add fused (multiplication followed by an addition) $-F1+F2*F3$ - is one of the most frequent arithmetic operation [18][57][64][99]. It is the basic operation for dot product, matrix multiplication, convolution or polynomial evaluation, which are standard operations in different applications, like digital signal processing or computer graphics. Therefore, a hardware unit dedicated for multiply-add fused is necessary for a wide range of processors, like DSPs or graphical processors.

There are two reasons for using a dedicated hardware unit to perform the multiply-add fused operation, rather than using a multiplier and an adder [46][55]:

1. The performance of dedicated hardware multiply-add fused unit for the combined operation is greater compared with the solution based on a multiplier and an adder.
2. Instead on performing two rounding operations (one rounding for multiplication and one for addition), only one rounding operation is performed on a dedicated unit. This leads to a reduction of rounding errors and an increase of the accuracy.

Therefore, in many application specific processors or even general use desktop processors, like IBM PowerPC, the multiply-add fused had been

implemented [99]. The multiply-add fused can also be used for addition or for multiplication, but the performance of these two operations is lower compared to a floating point adder or a floating point multiplier.

### 4.3.2 Basic Algorithm

Given three floating point numbers, $F1 = (-1)^{S1} * 2^{E1-bias} * 1.M1$, $F2 = (-1)^{S2} * 2^{E2-bias} * 1.M2$, $F3 = (-1)^{S3} * 2^{E3-bias} * 1.M3$, the multiply-add fused operation $F1 + F2 * F3$ involves the following steps [46][99]:

1. **Addition of multiplication exponents (*E2+E3-bias*) and the subtraction from the result of the addition exponen**t (*E1-bias*). In this way, the result exponent can be determined (max (*E2+E3-bias, E1-bias*)) and the amount for the alignment shifting for the addition operand.
2. **Align the addition mantissa** (1.*M*1) based on the exponents difference obtained in step 1.
3. **Mantissa multiplication of the multiplication operands** (1.*M*2 and 1.*M*3). This usually is carried out using a tree multiplier.
4. **Addition of the aligned mantissa**.
5. **Result complementation**.
6. **Leading zero detection**.
7. **Normalization** of the result mantissa in case of overflow or in case of leading zeros.
8. **Rounding**

Steps 1 and 2 (exponents addition and subtraction followed by addend's mantissa alignment), respectively 3 (mantissas' multiplication) are done in parallel, while the other steps are performed sequentially. Due to the many steps in the critical path, a large latency results for this basic algorithm.

### 4.3.3. Enhancements of the Basic Algorithm

In order to increase the performance of the floating point multiply-add fused, several improvements are performed to this basic algorithm. Regarding the multiplication, a tree multiplier (which in case of normal multipliers consists of an encoding scheme, a partial product reduction tree and a final adder) is used [46]. However, in this case only the encoding module and the partial product reduction tree are used. Therefore, the product will be obtained in a redundant carry-sum form. The aligned addend will be added to this redundant form using a carry-save adder line, resulting thus two final sum and carry strings which will be added in large carry propagate adder.

Regarding the alignment step (step 2), a bidirectional shift (left or right shift) may be needed, based on the sign of the exponents' difference. In order to avoid the bidirectional shifting, the following procedure is used [46][56][99]:

1. The addend 1.*M*1 is initially considered to be positioned *m*+3 bits (where *m* is the number of bits of mantissa – in case of double precision format *m*=52) to the left of the product.
2. Two zero bits will be placed in front of the product, which will stand for the guard and round bits.
3. Rather than the $E1-(E2+E3-bias)$ difference (which will indicate the amount of shifting to the left or right), a new difference will computed: $m+3-(E1-(E2+E3-bias))$. In case that the new difference is negative, no shift will be computed.

By performing these three steps, the bidirectional rounding is avoided, only a right shift rounding being needed. However, even in the case of effective addition, normalization step is required.

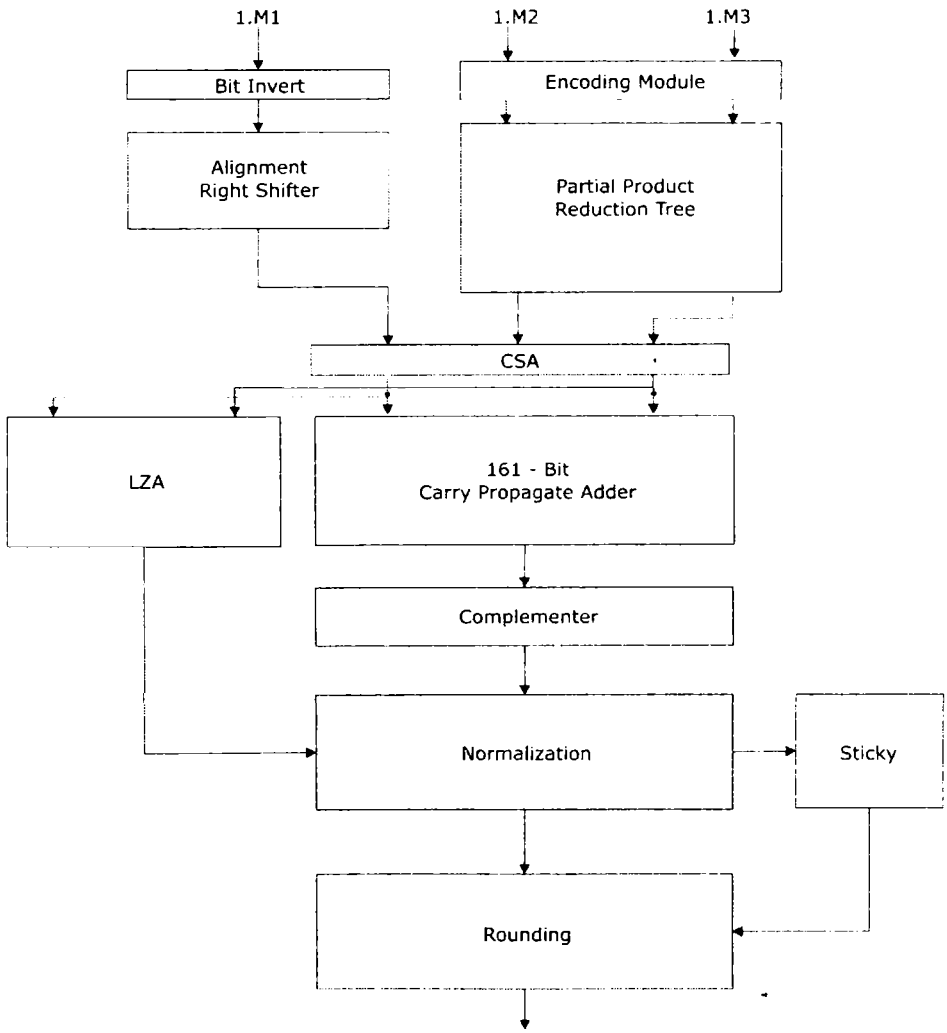



Figure 4.3 – General Architecture of the Mantissa Data Path in a Multiply-Add Fused [10][46][55]

The required size for the final carry-propagate adder is of $3*m+2$ bits [46](for double precision format 161 bits adder). As in the case of floating point addition, instead of a leading zero detector, a leading zero predictor is used, which runs in parallel with the final carry-propagate adder. However, unlike the floating point addition, rounding cannot be reduced to selection by using a compound adder, because normalization is done even in the case of effective addition. Therefore, rounding is performed after normalization.

Another improvement of the basic algorithm of the floating point multiply-add fused consists in the usage of a leading zero prediction instead of a leading zero detection [46][55]. This feature is inspired from the design of the floating point addition units. Like in the floating point addition, the leading zero prediction works in parallel with the carry propagate adder, providing the shifting amount needed for the normalization at the end of the addition.

The overall architecture of a floating point multiply-add fused unit is presented in Fig. 3.1. As it can be observed, the structure presents in the characteristic from both floating point multipliers (such as the encoding module based on a multiplication algorithm and the partial product reduction tree) and from floating point adders (such as the leading zero predictor or the alignment and normalization shifters).

### 4.3.4. High Performance Multiply-Add Fused Units

The basic algorithm for floating point multiply-add fused presented in Section 3.2 presents a high latency. The module which introduces the greatest latency is the large carry propagate adder (a three word length carry propagate adder), which for IEEE double precision format must be of size 161. Furthermore, the rounding step is not performed as a simple selection as in the case of floating point addition or multiplication, due to the normalization left shift which occurs even in the case of effective addition [55]. Several strategies for increasing the performance of the floating point multiply-add fused have been developed and are based either on a reduction of the carry-propagate adder size, either on reducing rounding to a simple selection.

One technique is used in [22] and is based on using a redundant adder – signed digit – for the three word addition. Two normalization shifters are used. One normalization shifter is used after the signed digit adder. After this first normalization shifter, a conversion from the signed digit redundant form to the conventional representation using a two word carry propagate adder is performed. After this carry propagate addition, another normalization shift is performed before rounding.

A more advanced design is proposed in [55] by Lang and Bruguera (Fig. 4.4). This strategy tries to reduce the rounding operation to a simple selection, in a similar fashion as in floating point adders or floating point multiplication, by reducing rounding to a simple selection a major reduction in latency occurs due to the following reasons: a single word length carry propagate adder is used (which works in parallel with the carry, round, guard and sticky bits computation circuits) and no carry propagate adder is used for rounding (instead a simple multiplexer circuit is used). However, as presented in the Section 4.3.2, because of normalization step, which in the case of the multiply-add fused is required even in the case of effective addition, performing rounding as a simple selection after the addition is rather difficult.

The main contribution of the floating point multiply-add fused proposed in [55] consists of performing normalization before the carry propagate addition. The obtained architecture of the [55] is presented in Fig. 3.3. The normalization is carried on the sum and carry strings which result after the carry-save adder line. However, normalization can be performed only after the leading zero anticipation. In order to decrease the delay, the carry save addition between the carry-sum redundant form of the product and the addend is performed in parallel with the leading zero prediction. Furthermore, two half adder lines are used, which also work in parallel with leading zero predictor. Also, parts of the final compound adder are placed after the half adder lines. The two lines of half adders and the compound adders' parts form two computational paths. The reason for the two computational paths is that one path computes for the case of a positive result, while the other computes for the case of a negative result.

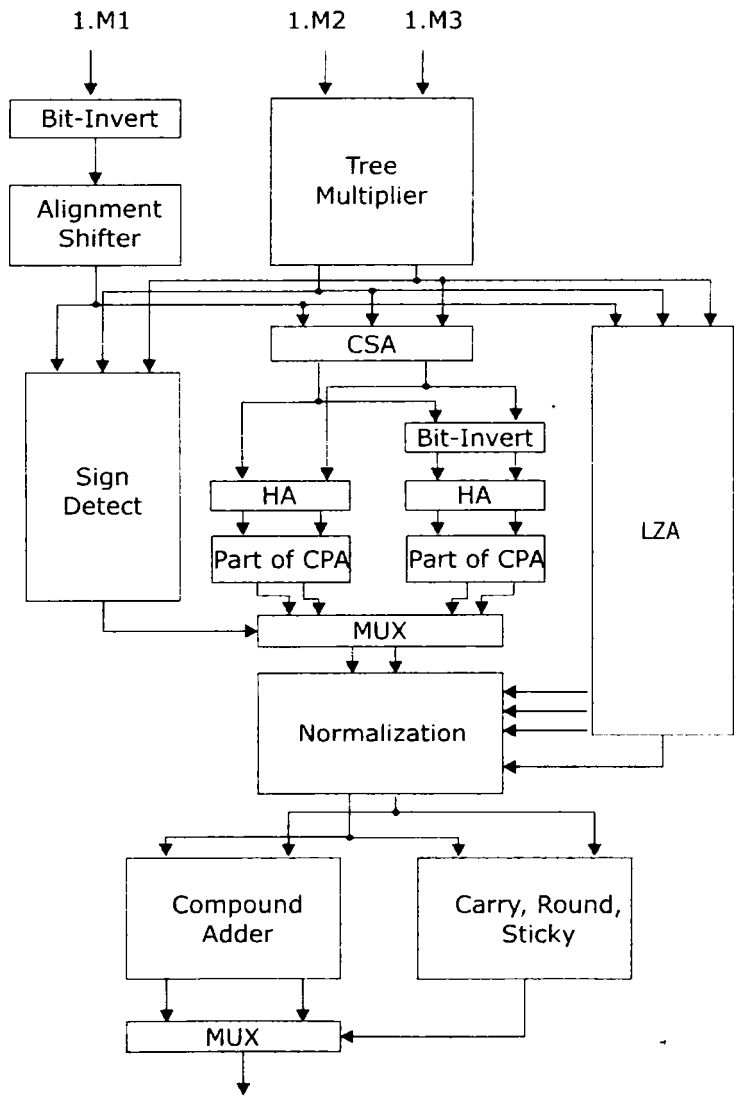



Figure 4.4 – The Floating Point Multiply-Add Fused as Proposed in [55]

Also, in parallel with the two computational path formed by the half adder line and parts of the compound adder, and the leading zero predictor, is placed a signed detection circuit which has the role of the sign detection of the final result. Based on the detected sign, the result is selected between the two computational paths. After the normalization left shifting, the final addition and rounding is performed similar to the one used in floating point addition and multiplication. The solution chosen by the authors in [55] consists of performing the rounding in a similar way to the Yu-Zyner floating point multiplication rounding algorithm [109]. Thus, for rounding a circuit for the carry, sticky computation, a small three bit carry propagate adder which computes the least, guard and round bits and one word length compound adder which runs in parallel with carry and sticky module are used. Based on this design is the one described in [57], which is dedicated to increasing the throughput of the operations performed on the multiply-add fused.

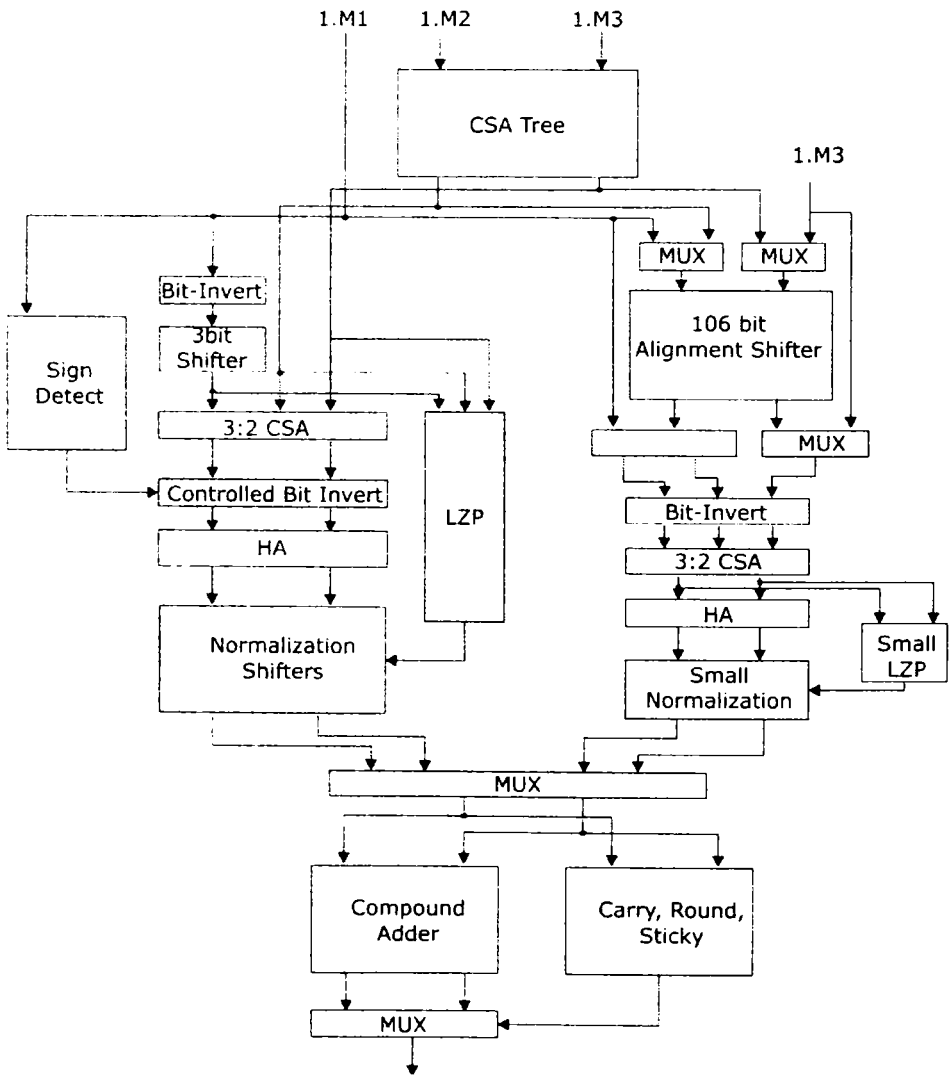


Figure 4.5 – The Double-Path Multiply-Add Fused Architecture [18]

Another approach for increasing the floating point multiply-add fused operation performance has its inspiration drawn from the floating point addition optimization techniques, and especially double path adders [18]. The approach to use two parallel and mutually exclusive paths in the mantissa data path of the floating point multiply-add fused has the main role of increasing the floating point addition (the case when the one of the multiplication operands is equal to one) performance. Figure 4.5 presents the general structure of the [18] multiply-add fused.

In order to obtain the double path structure for the floating point multiply-add fused, the alignment shifting is not performed in parallel with the multiplication. The two computational paths are called as in the classical double path floating point adders: the FAR path and the CLOSE path. On the CLOSE path are computed only effective subtractions when the exponent difference is 0, 1, -1 or 2 and the multiplication results in overflow. On the FAR path are computed the rest of the operations. The FAR path comprises of a small 3-bit alignment right shifter, the carry-save adder, the computational paths comprised of the half adders and parts of the compound adder (as in the design proposed in [55]), the normalization shifters and the leading zero prediction. The CLOSE path comprises of a large three word length alignment right shifter, the half adders computational path, a 3-bit normalization shifter and a small leading zero predictor. The selection between the two computational paths is realized before the final addition and rounding. Thus, this design requires a single compound adder, unlike the double path floating point adders, which require one compound adder for each computational path. The rounding algorithm used in this design is the same as in Lang-Bruguera multiply-add fused unit. This floating point multiply-add fused unit was designed in order to increase the performance of a floating point addition when executed on such unit, but without affecting the performance of the floating point multiply-add fused.

Regading the presented multiply-add fused architectures, the two designs based on combining rounding with the addition present the highest performance, due to the fact that only a single carry propagate addition is present in the critical path of the

## 4.4 Floating Point Division

In the case of the floating point multiply-add fused specific elements which are common to multiplication are found in the internal structure of the dedicated combined unit. These specific elements are the encoding module which implements one multiplication algorithm and the partial product reduction tree. It is expected that also for a dedicated floating point divide-add fused specific elements of the floating point division units to be present in the overall architecture. Therefore, a detailed analysis of the floating point division must be realized.

### 4.4.1 Basic Algorithm

The basic algorithm for the floating point division resembles greatly with the algorithm for the floating point multiplication. The division between two IEEE

floating point numbers ($F1 = (-1)^{s1} * 2^{E1-bias} * 1.M1$ and $F2 = (-1)^{s2} * 2^{E2-bias} * 1.M2$) is given by the following formula [51]:

$$s3 * 2^{E3-bias} * 1.M3 = F1 / F2 = (-1)^{s1 \oplus s2} * 2^{E1-E2+bias} * (1.M1 / 1.M2) \quad (4.6)$$

As it can be observed in the (4.6) the sign of the result is an exclusive-or between the two sign, the exponent is obtained by subtracting the two exponents and adding the bias, while the mantissa of the result is obtained by dividing the two mantissas.

Because the result has also to be represented in IEEE 754 format the following steps are also required [51]:

1. **Normalization of the mantissa** – because the mantissas of the two results are within $[1;2)$ interval, the result of their division is in the range $[0;2)$; if the result is in the range $(0,5;2)$ a normalization left shift with the exponent decrement are required.
2. **Rounding**, followed by post normalization.

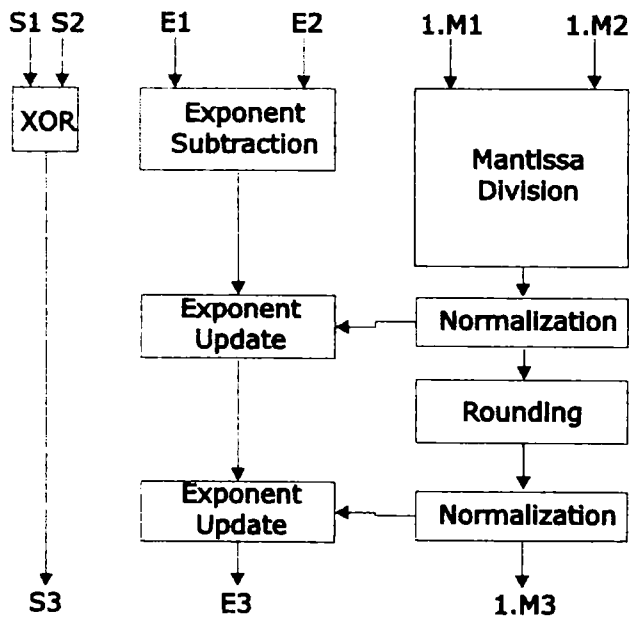



Figure 4.6 – Overall Architecture of a Floating Point Divider [51]

The basic architecture of a floating point multiplier is depicted in Fig 4.6. The largest delay module in this architecture is the mantissa divider. Furthermore, the rounding unit significantly contributes to an increase in the delay of the unit, because it requires a large carry propagate adder.

### 4.4.2 Mantissa Division

The mantissa division is the core operation for the floating point division. The mantissa division is basically an unsigned integer operation and is the most difficult from all basic integer arithmetic operations (addition/subtraction, multiplication and division). Several algorithms for division have been developed, which can be classified in four categories [29][60][71]: digit-recurrence, multiplicative methods, very high radix and table look-up. However, only the first two classes of division algorithms are being implemented in the current processors. Therefore, a detailed look for these two types of division algorithms is performed in the following sections.

#### 4.4.2.1 Digit-Recurrence Division

Digit recurrence division is a class of division algorithms which return a fixed number of quotient digits at each iteration. One class of digit recurrence division is the SRT (Sweeney, Robertson [83]and Toucher). The digit recurrence division between two numbers ( *Dividend* and *D – divisor* ) consists of finding the number *Q – quotient* which satisfies the following relation [52][67][100]:

$$Divident = Q * D + P \tag{4.7}$$

where $P$ represents the remainder of the division. The main feature of the digit recurrence algorithms is that a remainder is obtained after the division (unlike the multiplicative methods).

The digit recurrence algorithms consist of $k$ iterations, in which the following recurrence is followed [64]:

$$rP_0 = Dividend \tag{4.8}$$

$$P_{j+1} = rP_j - q_{j+1} * D \tag{4.9}$$

where $P_j$ represents the partial remainder at iteration $j$ ( $P_0$ is the first partial remainder), $r$ the radix in which the operation is performed ( $r = 2^b$ ), and $q_j$ represents the quotient digit. The quotient digit at iteration $j$ is obtained using a quotient selection function which depends on the divisor and on the partial remainder:

$$q_j = SEL(rP_{j-1}, D) \tag{4.10}$$

The number of iterations when dividing two $n$ bit numbers is given by the following relation:

$$k = \left\lceil \frac{n}{b} \right\rceil \tag{4.11}$$

The final partial remainder is computed from the final remainder [71]:

$$P = \begin{matrix} P_{k-1}, & if\ P_{k-1} \geq 0 \\ P_{k-1} + D, & if\ P_{k-1} < 0 \end{matrix} \tag{4.12}$$

For the computation of the square root the relation (4.3) becomes [21]:

$$P_{j+1} = rP_j - 2 * q_{j+1} * q_j - q_{j+1}^2 * r^{-i-1} \quad (4.13)$$

Therefore, digit recurrence algorithms can be used for both division and square root operations, many commercial floating point units present a combined divider and square root unit.

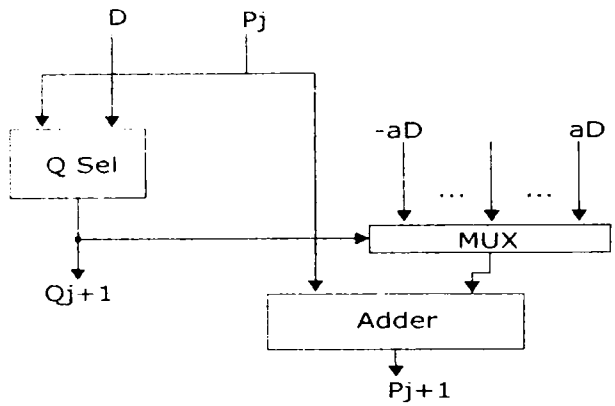



Figure 4.7 – Basic Block of a Digit Recurrence Divider [67]

As it can be observed from (4.9) and (4.11), the digit recurrence algorithms are based on the subtraction operation and their complexity is $O(n)$. Using (4.9) and (4.10), the basic block for a digit recurrence division (the block used for the computation of one digit of the quotient) is presented in Fig. 4.7. The divider may use only one such block and the division will be performed sequentially. In this case, the divider presents a reasonable cost, but it cannot be used efficiently in pipelined architectures (because the divider is occupied by only one division). At the opposite, a divider can be implemented by unrolling of the previous solution. In this case, the divider has $k$ blocks [10]. Therefore, the cost of this solution is much higher than the previous solution, but it has a far larger throughput. Also, combined solutions can be used, based on a trade-off between the area overhead and desired throughput. However, in many commercial implementations the first solution is used, mainly because the division is not a frequent floating point operation.

#### 4.4.2.1.1 Design Choices

Several design choices must be taken when designing a division unit, which implies a series of both performance and cost tradeoffs. These design choices are related to:

- **Radix.** A higher radix means less iteration, leading an increase in the performance [11]. However, this reduction in iterations comes at a cost [40]. On one hand the quotient digit selection becomes more complicated, either increasing the combinational logic needed, either the table lookup (the latency of the table is increased linearly with the radix, while the area of the table quadratically) [68]. This leads to an increase in the latency of the iteration, which may lead to the increase of the clock period. Furthermore, higher radices (for example 8 or 16) involve the generation of

hard multiples (like 3x, 5x, etc) which are difficult to obtain. As in multiplication, the chosen radices for division are 2 and more frequently 4.

- **Quotient Digit Set.** The quotient digit set depends on the radix $r$. The easiest way is to use a set consisting from $r$ values. This type of set is called the non-redundant digit set. However, using a non-redundant digit set means a very low performance from the division algorithm (for example, a radix-2 non-redundant digit set is equivalent to a non-restoring division). The redundant digit set is comprised of a set of digits $\{-a,\ldots,-1,0,1,\ldots,a\}$, where $r-1 \geq a \geq \frac{r}{2}$. For radix-2, the redundant digit is represented by the set $\{-1,0,1\}$ - this algorithm has been called SRT. For radix-4, there is a minimal redundant digit set ($\{-2,-1,0,1,2\}$) and a maximally redundant digit set ($\{-3,-2,-1,0,1,2,3\}$). The larger the redundancy, the easier quotient selection is. For radix-4 it has been proven by Oberman [68] that the quotient selection is about 20% faster for a maximally redundant digit set compared with a minimally redundant digit set. However, the cost for the increase in performance of the quotient selection is that it requires hard multiples, which are difficult to obtain. Furthermore, a more redundant digit set requires a more complex conversion to the non-redundant form which must be obtained. Thus, in the case of the selection of the quotient digit set, there must be a trade-off between the performance of the digit selection and the construction of needed multiples and the conversion to the non-redundant form [40].

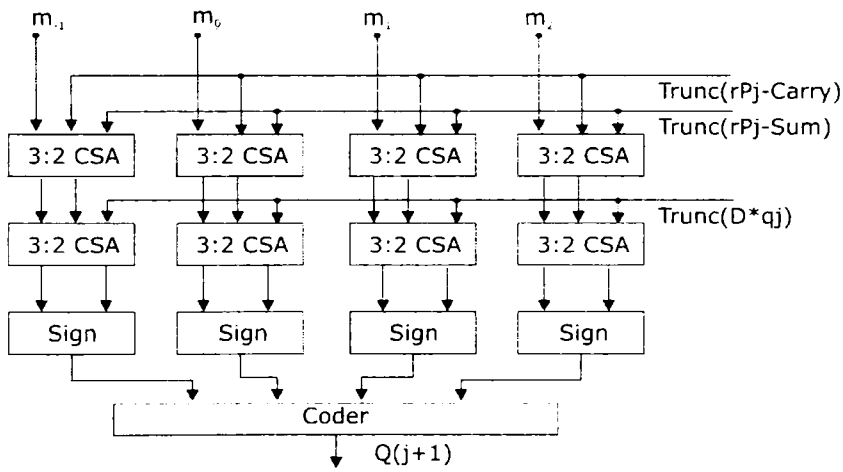



Figure 4.8 – Quotient Selection Scheme Based on Comparisons [8][20]

- **Quotient Digit Computation/Selection.** For radix-2, the quotient digit computation is relatively simply, employing only simple and fast combinational logic. For higher radices (radix-4), the combinational logic for quotient digit selection would become very complicated and with high latency. Therefore, for quotient digit selection two alternatives have been devised. The first one is using fast memory blocks in form of look-up tables (ROM or PLA) [64][68]. For radix-4 minimal redundant the table lookup

needs as inputs seven bits from the partial remainder and three bits of the divisor. The second alternative is represented by using a set of comparators [8][9][10][20][21] - Fig 4.8. These comparisons are made between a truncated partial remainder and a number of selection constants (which are preloaded at the first iteration depending on the divisor). The quotient digit is computed based on the results of these comparisons.

#### 4.4.2.1.2 Redundant Remainder Representation

One important is represented by the remainder computation (4.9) which requires a subtraction. Using a full carry propagate adder in order to perform this addition, the latency of the division becomes very large, mainly because this addition/subtraction is performed sequentially with the quotient digit selection. Thus, it is required to be avoided the full carry propagate addition for partial remainder computation. This can be achieved by using a redundant representation of the partial remainder, either a signed digit (borrow save) or a carry-save representation [67]. Thus, the partial remainder computation is performed very fast, because the carry propagation is thus avoided. For example, using a carry-save representation , the latency of this step is only 2 logic levels, which is a major improvement compared with latency of a large carry propagate adder (8 logic levels for IEEE double precision numbers) .

When using table lookup for quotient digit selection, the redundant representation of the partial remainder may prove a disadvantage. The reason for this is represented by the fact that instead of two entries to the table (the truncated partial remainder and truncated the divisor), three entries are needed, which will increase the table complexity and area [19]. A solution has been provided by using a small carry propagate adder (usually one or two bits greater than the number of bits needed for partial remainder) before the table lookup [71]. There is a tradeoff between the size of the carry propagate adder and the complexity of the table lookup, because smaller carry propagate adders means lower latency, but also larger truncation errors, which will lead to a more complex table lookup (Fig. 4.9).

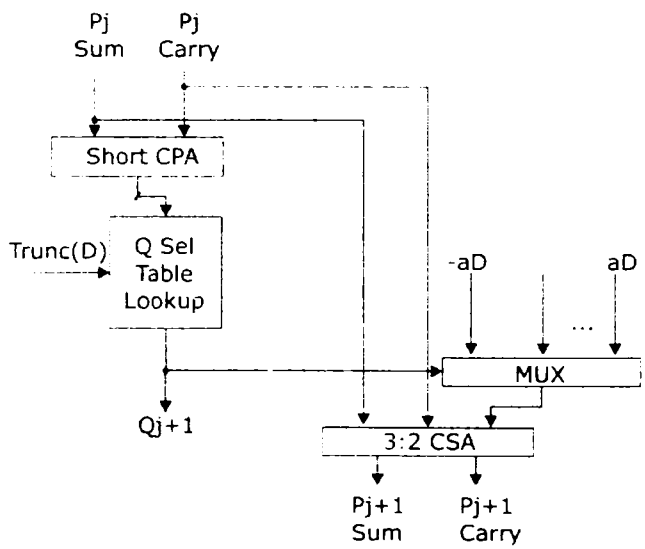


Figure 4.9 – SRT Stage with Remainder in Carry-Save Form and a Short Carry Propagate Adder for the Quotient Selection [64][71]

When using the comparators based method to select the quotient digits, the subtraction for comparisons is performed using carry-save adders, without the need of carry propagation [9][20]. A sign detector is used for determining the sign of the subtraction and a coder is used to analyze the obtained signs and to provide the quotient digit.

#### 4.4.2.1.3 Overlapped Architectures

One possibility to maintain the advantages of both higher radices, such the high number of quotient bits obtained at each iteration, and the low radices (low latency per stage, simple quotient selection, avoidance of hard multiples) is to design overlapped digit-recurrence architectures. By using the overlapping strategy, higher radices is obtained from cascading lower radices stages. Four types of overlapping strategies do exist, as described in [40]:

- Overlapped remainder formation
- Overlapped quotient selection
- Overlapped remainder and quotient formation
- Hybrid overlapping

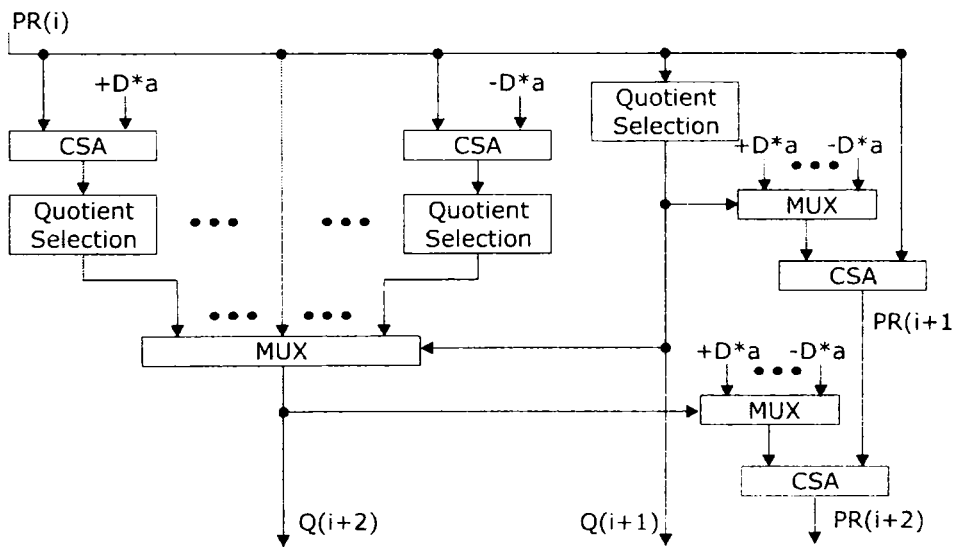



Figure 4.10 – Overlapped Quotient Selection [40]

Figure 4.10 depicts the structure of a SRT divider stage based on the overlapped quotient selection. The first quotient digit is obtained as in a normal SRT stage. In parallel with the first quotient digit selection several numbers of quotient digits (for each digit in the quotient digit set) are pre-computed. The first selected quotient digit will select the appropriate second quotient digit.

By employing the overlapping strategy, higher radix can be obtained from faster and simpler lower radix, such as radix 16 obtained from two overlapped radix-4 stages, or radix-4 from two overlapped radix-2 stages [9]. This strategy was employed in commercial processors, such the radix-16 divider used in ARM VFP11 obtained from two overlapped radix-4, or the radix-8 divider used by Sun obtained from overlapping three radix-2 stages.

#### 4.4.2.1.4 Quotient Conversion

One important feature of the digit recurrence division is that the quotient digits are represented in a redundant form. The redundancy factor can have a significant influence on the performance of the quotient digit selection; a higher redundancy factor leads to a simpler selection. However, the final result must be in a non-redundant form. Thus, a conversion from the redundant form into the non-redundant form must be performed.

The quotient digits are obtained from $q_1$ up to $q_k$, in $k$ iterations, where $q_0$ is the most significant digit and is obtained first, while $q_{k-1}$ is the least significant digit and is obtained in the last step. The conversion is the transformation of $Q$ into $Q'$, where the two strings are defined by the following relation:

$$\begin{aligned} &Q = \sum_{i=1}^{k} q_i * 2^{-i}, q_i \in \{-a, \ldots, -1, 0, 1, \ldots, a\}, \\ &Q' = -q'[0] + \sum_{i=1}^{k} q'_i * 2^{-i}, q'_i \in \{0, 1\} \end{aligned} \tag{4.11}$$

The conversion between the redundant form and the non-redundant form is made using the following iteration [26][27]:

$$q'[j] = q'[j-1] + r^{-j} q_j \tag{4.12}$$

This type of conversion can be very simple when the obtained quotient digits $q_j$ are positive. In this case, the addition means only adding the non-redundant representation of the quotient digit at the end at of the partial of the non-redundant quotient. However, the redundant quotient digits can be negative, which means that in order to convert these digits a carry propagate addition must be performed. The carry propagate adder leads to an increase area and latency for the conversion, which leads to an increase area and latency for the entire division.

A method for removing the carry propagate addition in the conversion process has been proposed by Ercegovac and Lang [27]. This method uses to strings $A[j]$ and $B[j]$, one expecting positive or zero quotient digits, while the other expecting negative digits.

The resulted quotient will have the following expression [26][27]:

$$Q'[j] = \begin{cases} A[j-1] + q[j] * r^{-j}, q[j] > 0 \\ A[j-1], q[j] = 0 \\ B[j-1] + (r - |q[j]|) * r^{-j}, q[j] < 0 \end{cases} \tag{4.13}$$

In order to obtain this expression for the quotient result, the $A[j]$ and $B[j]$ must be equal to:

$$\begin{aligned} A[j] &= Q'[j] \\ B[j] &= A[j] - r^{-j} \end{aligned} \tag{4.14}$$

The initial value of the $A[j]$ and $B[j]$ strings are:

$$\begin{aligned} A[1] &= \begin{cases} +q_1 * r^{-1}, q_1 \geq 0 \\ -|q_1| * r^{-1}, q_1 \geq 0 \end{cases} \\ B[1] &= \begin{cases} +(q_1 - 1) * r^{-1}, q_1 \geq 0 \\ -(|q_1| + 1) * r^{-1}, q_1 \geq 0 \end{cases} \end{aligned} \tag{4.15}$$

The recurrence for the two strings is given by the following relation [26][27]:

$$\begin{aligned} A[j] &= \begin{cases} A[j-1] + q_j * r^{-j}, q_j \geq 0 \\ B[j-1] + \left(r - |q_j|\right) * r^{-j}, q_j < 0 \end{cases} \\ B[j] &= \begin{cases} A[j-1] + \left(q_j - 1\right) * r^{-j}, q_j > 0 \\ B[j-1] + \left(r - |q_j| - 1\right) * r^{-j}, q_j \leq 0 \end{cases} \end{aligned} \tag{4.16}$$

Thus, obtaining the two strings requires only a concatenation of the decoded quotient digit at the end in each iteration. In this way the large carry propagate adder is thus avoided, which leads to smaller latency and area. The only thing needed to be computed is the $\left(r - |q_j|\right)$ difference, which requires a small carry propagate adder. For example, for radix 2, there is no need for such an adder, for radix 4 minimally redundant a two bit carry propagate adder is needed, while for radix 4 maximally redundant a three bit carry propagate adder is needed. Thus, as the radix and the redundancy factor increases, the more complex is the conversion.

Similar to on-fly quotient conversion is the on-fly rounding [28]. The main difference is represented by employing another string. This new string is used due to the fact that -1 ulp subtraction is needed when the final remainder is negative.

By these means, both the conversion of the quotient and the rounding in case of floating point operation is convenient to perform for digit-recurrence algorithm, which represents an advantage of this type of algorithms.

#### 4.4.2.2 Multiplicative Methods

Multiplicative methods imply using a multiplication as core operation, unlike the digit-recurrence methods which are based on addition/subtraction. A wide range of such algorithms, from which two categories are more important: Newton-Raphson algorithm and series expansion (or Goldschmidt) algorithms [26][60].

Newton-Raphson multiplicative algorithm for performing division is based on the Newton's method to approximate a root of the equation $f(x) = 0$ using the following iteration [59]:

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} \tag{4.17}$$

If we considered the function $f(x) = \frac{1}{x} - D$ which has the root $x = \frac{1}{D}$, it is possible to find the result of the division $q = \frac{Dividend}{D}$ by employing the following iteration[59]:

$$X_{i+1} = X_i(1 - 2DX_i) \tag{4.18}$$

The series expansion method relies on determining a string of numbers $X_i$, so the following relation to hold [59]:

$$\begin{aligned} &D * X_1 * X_2 * \ldots * X_n \to 1 \Rightarrow \\ &\Rightarrow Divident * X_1 * X_2 * \ldots * X_n \to q \end{aligned} \tag{4.19}$$

Having in mind that the divisor $D \in [1,2)$, it is possible to represent the divisor as $D = 1 + \varepsilon$, where $\varepsilon \in [0,1)$. In this case, the following string is considered:

$$X_0 = 1 - \varepsilon \quad X_i = 1 + \varepsilon^{2^i} \tag{4.20}$$

Therefore:

$$\begin{aligned} &D * X_0 * X_1 * X_2 \ldots * X_n = (1+\varepsilon)(1-\varepsilon)(1+\varepsilon^2)(1+\varepsilon^4)\ldots(1+\varepsilon^{2^n}) = \\ &= 1 - \varepsilon^{2^{n+1}} \to 1 \\ &Divident * X_0 * X_1 * X_2 \ldots * X_n \to q \end{aligned} \tag{4.21}$$

While the $X_i$ string can be computed using the following recursion:

$$X_{i+1} = 2 - D * X_0 * X_1 \ldots * X_i \tag{4.22}$$

Therefore, if we consider $Y_i = D * X_0 * X_1 \ldots * X_i$ the algorithm relies on the following recursions:

$$\begin{aligned} X_{i+1} &= 2 - Y_i \\ Y_{i+1} &= Y_i * X_{i+1} \\ Q_{i+1} &= Q_i * Y_{i+1} \end{aligned} \tag{4.23}$$

The two multiplication based division algorithm are very similar, due to the following reasons [59][60]:

- Both require two floating point multiplications and one subtraction
- Both have a quadratic convergence
- Both require an initial estimation, which is usually implemented as a table look-up; the number of performed steps depends heavily on this estimation

The main difference between the two multiplicative division algorithms is that the two floating point multiplications required in Newton-Raphson are dependent, while in the series expansion algorithm these two floating point algorithms are independent [60]. Therefore, the second algorithm may present a higher performance due to the possibility of pipelining the two multiplications [92].

In terms of required hardware, both algorithm rely on a slight modification of other floating point units, such as the floating point multiplier of the floating point multiply-add fused. Usually, this modifications increase slightly the latency of these floating point units, mainly due to the required multiplexers used in the critical path [26]. Another major disadvantage is represented by the rounding operation, which requires several more iterations and the computation of the remainder using a floating point multiplication and a floating point subtraction.

#### 4.4.2.3 Comparison between Digit Recurrence and Multiplicative Methods

In Table 4.3 a comparison between the two main classes of algorithms for floating point division is performed. The comparisons are made regarding convergence times, rounding and remainder computation and hardware requirements.

Table 4.3 – Comparison between Two Main Classes of Division Algorithm

| | Convergence | Rounding | Hardware Requirements |
|---|---|---|---|
| Multiplicative Methods | Quadratic | Difficult<br>Requires floating point multiplication and subtraction | Floating point multipliers or floating point MAF |
| Digit Recurrence Division | Linear | Easy<br>Remainder is obtained at the end of the iterations<br>Rounding is performed on-fly | Dedicated divider stages |

As presented in [92], 11 out 13 processor designs use digit recurrence algorithms. The vast majority of the digit recurrence algorithm implemented for division comes as a result that when implementing floating point division as shared units with other floating point operations (floating point multipliers and floating point multiply-add fused) a performance degradation of about 40% comes compared to the case when division is performed as a dedicated floating point unit using digit recurrence algorithms, as showed by Oberman based on an analysis made on SPEC FPU benchmarks [70][69]. The conclusion was expressed by Ercegovac and Lang, that for a dedicated floating point division unit (with no hardware sharing with other floating point units) the best choices are represented by digit-recurrence [26]. However, when the division is not implemented as a specific dedicated unit and hardware sharing with other floating point units is considered, the multiplying methods (which share the floating point multiply-add fused or the floating point multiplier) are the preferred ones.

## 4.5 Floating Point Divide-Add Fused

### 4.5.1 Basic Algorithm and Architecture

In order to perform a floating point divide-add fused between three floating point numbers Given three floating point numbers, $F1 = (-1)^{S1} * 2^{E1-bias} * 1.M1$, $F2 = (-1)^{S2} * 2^{E2-bias} * 1.M2$, $F3 = (-1)^{S3} * 2^{E3-bias} * 1.M3$, the divide-add fused operation $F1 + F2/F3$ involves the following steps [7][76]:

1. **Subtraction of division exponents** (*E2-E3+bias*) and the subtraction from the result of the addition exponent (*E1-bias*). In this way, the result exponent can be determined (max (*E2-E3+bias, E1-bias*)) and the amount for the alignment shifting for the addition operand.
2. **Align the addition mantissa** (1.*M*1) based on the exponents difference obtained in step 1.
3. **Mantissa division** of the division operands (1.*M*2 and 1.*M*3).
4. **Addition of the aligned mantissa** (the aligned addend with the quotient).
5. **Result complementation**.
6. **Leading zero detection**.
7. **Normalization** of the result mantissa in case of overflow or in case of leading zeros.
8. **Rounding** (for this operation the remainder is necessary)

The proposed divide-add fused algorithm is similar to floating point multiply-add fused algorithms [46][56][57]. Steps 1 and 2 (exponents addition and subtraction followed by addend's mantissa alignment), respectively 3 (mantissas' division) are done in parallel, while the other steps are performed sequentially. Furthermore, as in floating point addition and floating point multiply-add fused, the leading zero detection is replaced by a leading zero prediction which is done in parallel with the addition.

The overall architecture is depicted in Fig. 4.12. The floating point divide-add fused structure in its mantissa computation path contains the following blocks: addend complementation (in case of an effective subtraction), alignment shifter, the division module (which will have as results the quotient and the remainder), the mantissa adder (which works in parallel with the leading zero predictor), the normalization left shifter and the rounding unit.

This structure is very similar to the one of the floating point multiply-add fused [46][55], the major difference being that a divider is placed instead of a multiplication block. Furthermore, because the result of the division does not come in a convenient redundant form (such as the carry-save form which is the typical result of the partial products reduction tree of the tree multipliers), the 3:2 carry save adder, which is placed before the mantissa adder, is not required.

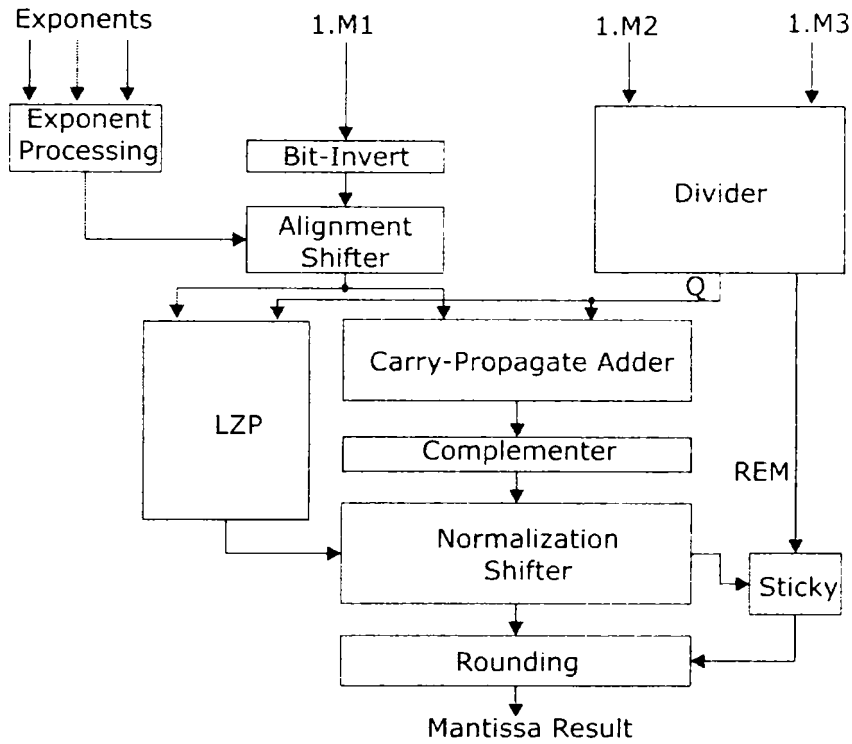



Figure 4.11 – Overall Architecture of the Mantissa Data Path of a Floating Point Divide-Add Fused [7][76]

The biggest influence in both the performance and the area overhead in the proposed scheme is represented by the number of quotient bits needed. The number of quotient bits influences the following modules:

- The latency/area of the divider; greater number of quotient bits means greater division latency, as the digit-recurrence division has a linear complexity with the number of quotient digits
- The alignment shift amount; the number of logic levels if the barrel right shifter used for alignment is influenced by the number of quotient bits
- The mantissa adder size
- The normalization left shifter

Therefore, a very analysis must be performed in order to use the minimum number of quotient bits. However, this number has to be high enough in order to attain a reasonable precision for the operation.

### 4.5.3 Number of Required Quotient Bits

The number of quotient bits represents the main issue in the design of the floating point divide-add fused, because it affects both the performance of the unit and the area overhead. However, the number of bits has to be large enough in order to make possible a correct IEEE rounding.

In order to determine the number of quotient bits needed we will consider different cases, depending on the difference between exponents ($d$ – relative to the number of bits in the mantissa $m$) and the effective operation which will be executed (addition or subtraction) [7].

*Case I.* $d \geq m+3$

This case corresponds to the one presented in Fig. 4.13 a. The result significant will be equal to the addend (or incremented addend in case of rounding towards infinity). The maximum needed left shift of the addend is equal to $m+3$. For rounding towards nearest even, two zero bits have to be inserted (for the round and guard digits) – similar to the floating point multiply-add fused [46]. The sticky bit corresponds to the condition quotient different to zero (which is always true), thus the sticky bit is always one in this case. For rounding towards zero (truncation), no zeros have to be inserted, the result being equal to the addend. In case of rounding for infinity, no zeros have to be inserted and the result is equal to the incremented addend, as the sticky bit is always one (for rounding towards zero and infinity the condition $d \geq m+3$ is transformed into $d \geq m+1$) Therefore, for this case, we don't need any quotient bits.

*Case II.* $1 < d < m+3$

This second case corresponds to the case when the addend must be left-shifted for alignment and the addition/subtraction of the quotient must be performed (Fig 4.13.b). In this case, in case of an effective subtraction the maximum number of leading zeros is one (as in the case of floating point addition when the exponents' difference is greater than one [31]). As in floating point digit recurrence division, the remainder will be used for the computation of the sticky bit. The number of quotient digits needed in this case depends on the rounding mode and is:

- $m-d+3$ for rounding towards nearest even ($m-d+2$ in order to align the quotient and two quotient digits for the round and guard bit, in case of no cancellation and no overflow, $m-d+1$ in case of overflow and $m-d+3$ in case of leading zero)
- $m-d+1$ for rounding towards zero (truncation)
- $m-d+1$ for rounding towards infinity

*Case III.* $1 < d \leq -2$

In this case also the addend must be added to the obtained quotient (Fig. 3). The main feature of this case is that in case of an effective subtraction, a massive cancellation of the result can occur, thus, leading to a large number of leading zeros (this case is similar to the CLOSE path in double path adders [31]). Unlike the CLOSE path in the double path adders (which occurs when the exponent difference is -1, 0 or 1), in the case of floating point divide-add fused also the case of is considered because the most significant bit of the quotient can be equal to zero (un-normalized quotient – Fig. 4.13.c). The number of quotient digits needed in this case depends on the amount of the leading zeros. The maximum number of leading zeros, considering only the addend and the most significant part of the quotient, is equal to $m$ (the case when the addend is equal to the most significant part of the quotient). Furthermore, after the most significant $m$ positions of the quotient, the next bits of the quotient can be equal to zero (a series of $k$ zeros). In this case, the number of quotient bits needed, if the rounding mode is rounding towards nearest

even, is equal to $m+k+m+3$ ($m+1$ - leading zeros resulted after the subtraction, $k$ - the length of the series of zero quotient bits, $m$ - the required length for mantissa, 2 – guard and round bits for rounding towards nearest even). For every rounding mode, the remainder is used for the computation of the sticky bit, as in floating point division [28].

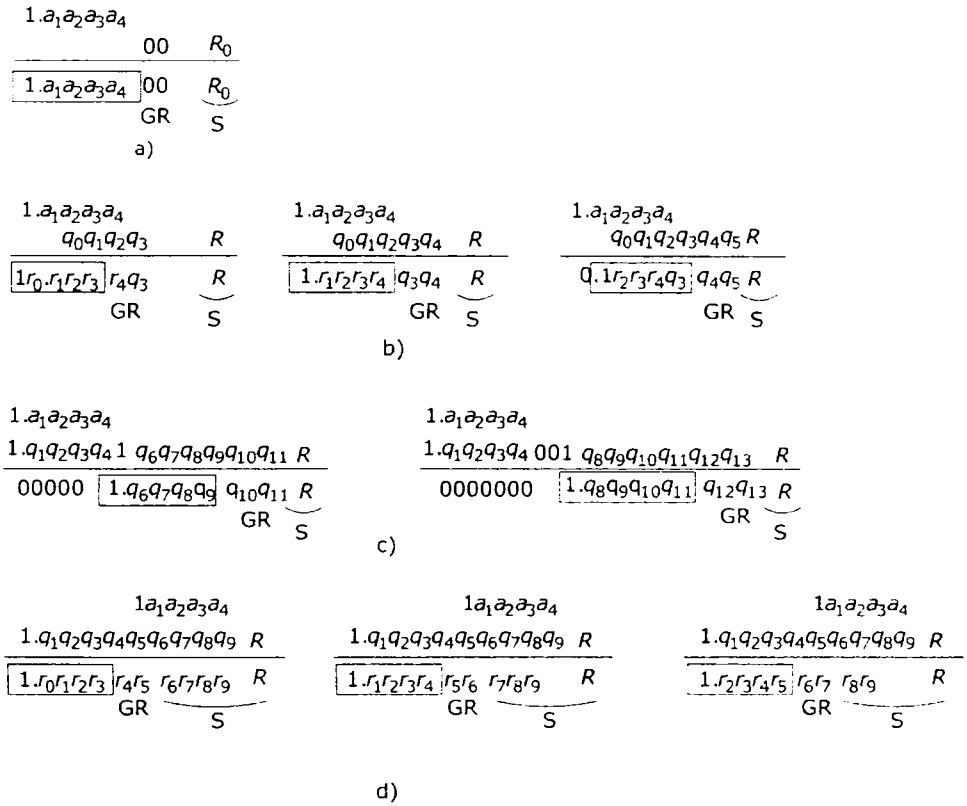



Figure 4.12 – The Four Cases for Divide-Add Fused a) $d \geq m+3$ b) $1 < d < m+3$ c) $1 < d \leq -2$ d) $d < -2$ (in this trivial example $m$=5)

*Case IV.* $d < -2$

In this case the addend is shifted to the right compared to the quotient (Fig 4.13.d). Also, in this case, the maximum number of leading zeros is 1 (this case is similar to the FAR path in the double path adder). The truncated result is given by the most significant $m-1$ bits of the addition of the result in case of an overflow, $m$ bits for no overflow and no leading zero, and $m+1$ bits for the leading zero case. For rounding towards nearest even, two more bits are needed, while for rounding towards zero and rounding towards infinity there is no need for additional bits. The rest of the addition result bits are used for the sticky bit computation. In this case, the number of the quotient bits is equal to $m+|d|-1$, for overflow, $m+|d|$, when no overflow and no leading zero occurs, and $m+|d|+1$ when the leading zero occurs.

As it can be observed from this analysis, the third and the fourth case represent problematic cases, due to the high number of quotients bits required.

### 4.5.3 Implementations

As it was presented in the previous section, the number of quotient digits can vary from case to case. In cases III and IV, the number of required quotient digits can be very high. This will lead to lower performances, as the latency of the division will increase, the mantissa carry-propagate adder will be larger and the two shifters will have an increased number of levels. Better performance can be obtained, but with a loss in precision as penalty. Several implementations are presented in this section, the evaluation of the performance, cost and precision being evaluated in the next section.

#### 4.5.3.1 Pro-Accuracy Implementation

The first proposed implementation uses a number of $2m+3$ quotient bits. $2m+3$ is the minimum number of quotient bits for which in the third case in the case of an effective subtraction, there is no loss in precision when the most significant bits of the quotient are equal to the addend and the first quotient bit after the result is one. This proposed floating point divide-add fused resembles with a floating point multiply-add fused [46][55]. The alignment shifter, the adder, leading zero predictor, the normalization shifter and the rounding unit has the same size as in the case of a basic floating point multiply-add fused - $3m+5$.

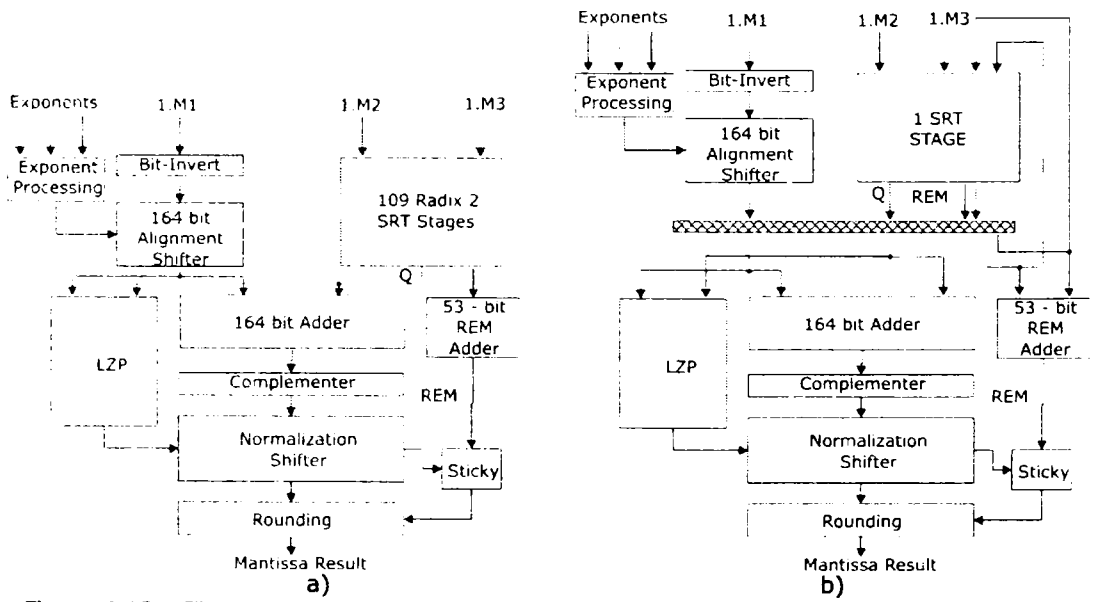



Figure 4.13 – The Pro-Accuracy Architecture a)Divider Unfolded b)Sequential Generation of Quotient Bits ($m$=53 – IEEE double precision format)

The proposed structure is presented Fig 4.14. The first proposed structure (Fig. 5.3-a) presents an unfolded divide unit, which is formed from $p = \lceil 2m/b \rceil$ successive digit recurrence blocks, where $r = 2^b$ is the working radix [64]. This structure presents a high throughput, but the area overhead is considerable high. In the second proposed architecture (Fig 4.14-b) the quotient bits are

obtained sequentially. This architecture presents a lower cost than the first architecture, but has a lower throughput.

Regarding the obtained accuracy, the problematic cases do appear for the third and the fourth case presented in the previous section. Thus, in the third case, the problems appear in the event of an effective subtraction, when the massive cancellation can appear and the number of leading zeros is greater than $m+1$ (the last favorable case is represented by an un-normalized quotient equal in its most significant bits to the addend – which is shifted by one position to the right compared to the quotient). This corresponds to the case when the most significant part formed of $m+1$ bits is equal to the addend, while a series of $k$ zero bits in the quotient follows after the most significant leading m bits. In this case, the proposed implementation will consider the following rounding rules (which are not always IEEE compliant):

- If the remainder is equal to zero, then the result is exact. The obtained result will be formed by the least significant $2m-m-k+3$ bits (which follow after the leading zero) followed by zeros until the completion of the mantissa.
- If the remainder is different than zero, then the result will not be IEEE compliant. For rounding to zero (truncation) the result will be formed by the least significant $2m-m-k+2$ bits (which follow after the leading zeros) followed by zeros until the completion of the mantissa. Rounding towards nearest even will be considered as a truncation (as the considered guard bit and round bit will be equal to zero). In the case of rounding towards infinity, a plus 1 addition is performed to the least significant quotient bit (the sticky bit is indicated by the non-zero remainder).

Therefore, the maximum number of leading zero for which the result is correct (IEEE rounded), in the event of a non-zero remainder, is equal to $m$. For this case, all four IEEE rounding modes can be performed correctly ($m$ bits remain for the mantissa, 1 bit is for guard bit, 1 bit for the round bit, while the remainder is determine the sticky bit [28]).

In the fourth case of division, loss in precision does appear when the addend is right-shifted compared to the quotient more than $m+1$ positions. In this fourth case, the most significant $m$ bits represent the non-rounded result, followed by two bits (the guard and round bits in case of rounding towards nearest even). The sticky will be computed from the next $m$ bits, the addend bits which were not added, and the remainder. In case of a non-zero remainder, there is a probability of a possible carry one which would result from the non-calculated quotient bits added to the least significant bits of the addend. This carry could propagate to the most significant bits of the result. Therefore, the result may not be correct. If for rounding towards zero (truncation) and rounding towards nearest even, this carry was not considered, for rounding towards infinity this carry will be considered and will be added to the lowest significant bit of the first m bits of the result. Furthermore, a plus one addition is also performed if the sticky bit is not equal to zero.

#### 4.5.3.2 Pro-Accuracy Implementation

The second proposed implementation uses a number of $m+3$ quotient bits. This number of quotient bits is the same as in a standard floating point division

[69]. The alignment shifter, the adder, leading zero predictor, the normalization shifter and the rounding unit have a much smaller size compared to the previous implementation - $2m+5$. All the features of the floating point divide-add fused presented in the previous section remain to this implementation, but their size is smaller, which means that lower latency and lower cost is associated to this proposed floating point divide-add fused.

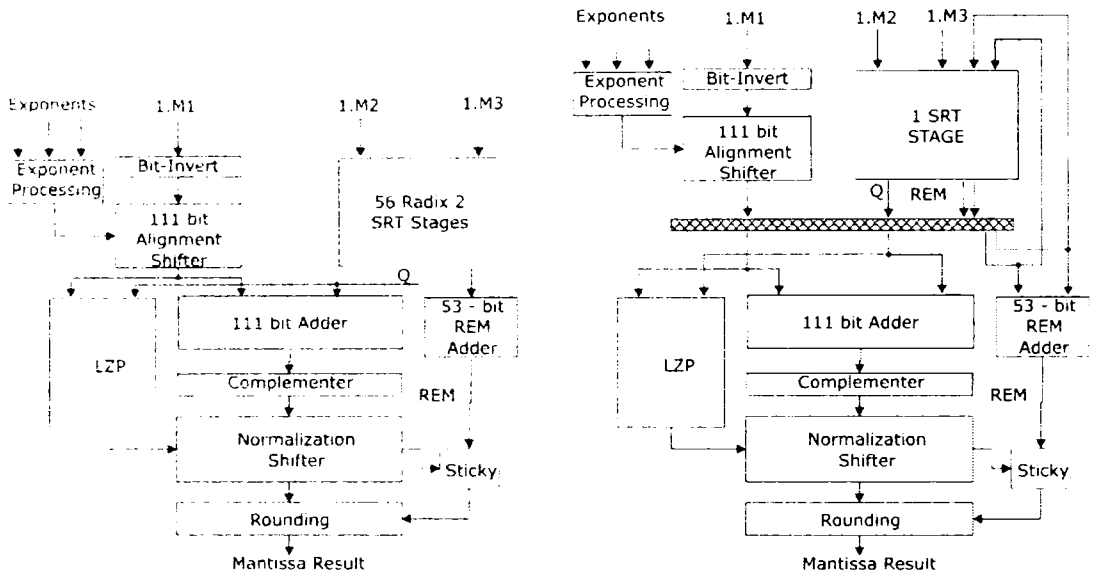



Figure 4.14 – The Pro-Performance Architecture a)Divider unfolded b)Sequential Generation of Quotient Bits ($m$=53 – IEEE double precision format)

The proposed structure is presented Fig 4.15. As in the case of the floating point divide-add fused, two architectures were designed: an architecture which has an unfolded divider (a) and architecture which presents a sequential divider (b).

Regarding the accuracy, this floating point divide-add fused presents more problematic cases (when it cannot be performed an IEEE compliant rounding operation) compare to the one presented in 5.4.1. In the third case, in the event of effective subtraction and a massive cancellation of the result, the loss in precision is inevitable. The three types of rounding are performed as in the previous implementation when the number of leading zeros where greater than $m+1$. Loss precision is also inevitable for the fourth case. Also, in this case, the rounding is performed as in the previous implementation, when the addend is right-shifted with more than $m+2$ positions.

### 4.5.4 Variable Latency

As presented in Section 4.5.2, the number of quotient bits varies from case to case. Thus, in the first case no quotient bits are needed, in the second case a rather small number of quotient bits are needed, in the third case, a large number of quotient bits are needed in case of an effective subtraction and massive cancellation, while in the fourth case also a large number of quotient bits is required. Therefore, devising a variable latency would ensure a higher performance

when small number of quotient bits is required and a high accuracy when a large number of bits are required.

The variable latency floating point divide-add fused is best suited to the sequential floating point divide-add fused depicted in Fig 4.14.-b and Fig 4.15-b. While in these designs, the feedback path of the division unit is used for generation of the maximum number of quotient bits, in the variable latency design, the feedback path of the division unit is used only for a limited number of times, depending on the case. Considering the design in Fig 4.14 - b, the feedback path of divider is used as follows:

- One time for the Case I (although no quotient digit is required)
- $p = \left\lceil q/b*f \right\rceil$ for the Case II where $q$ is the number of required quotient bits, $r = 2^b$ is the radix and $f$ the number of unfolded stages,
- $p = \left\lceil q/b*f \right\rceil$ for the Case III and effective addition
- $p = \left\lceil q/b*f \right\rceil$ for the Case IV and $d > -(m+1)$
- $\left\lceil 2m+3/b*f \right\rceil$ for Case III and effective subtraction and Case IV and $d < -(m+1)$

The decision for the number of division loops is taken based on the effective addition and the exponents' operation result. In the Case IV, more quotient bits can be generated than the maximum, but this leads to an increase carry propagate adder. In Case III, effective subtraction, fewer quotient digits may be needed, in the event of a small number of leading zeros. However, the number of leading zeros is determined by the leading zero predictor, after the quotient bits have been generated [26]. The variable latency algorithm does not require changes to the divide-add fused unit, but only to the feedback control block.

### 4.5.5 Interval Divide-Add Fused

By implementing rounding towards infinity and rounding towards zero (which can be easily extended to rounding towards negative infinity and rounding towards positive infinity), interval arithmetic operations can be performed on the proposed floating point divide-add fused. Furthermore, in the event of an inexact result (due to few bits in the quotient), the performed rounding considered for both rounding towards zero and rounding towards infinity will covered the less favorable cases. Therefore, the principles of the interval arithmetic (the obtained result to contain the correct result) were respected [47]. The two operations are performed according to the Table 4.4.

Table 4.4 - The Divide-Add Fused of Three Intervals $X \pm Y/Z$, for $0 \notin Z$ [47]

| Operation | $[Y_{lo};Y_{hi}]$ | $[Z_{lo};Z_{hi}]$ | Result |
|---|---|---|---|
| Addition | $Y_{lo} > 0$ | $Z_{lo} > 0$ | $X_{lo} + {}^{Y_{lo}}/_{Z_{hi}}, X_{hi} + {}^{Y_{hi}}/_{Z_{lo}}$ |
| Addition | $Y_{lo} > 0$ | $Z_{hi} < 0$ | $X_{lo} + {}^{Y_{hi}}/_{Z_{hi}}, X_{hi} + {}^{Y_{lo}}/_{Z_{lo}}$ |
| Addition | $Y_{hi} < 0$ | $Z_{lo} > 0$ | $X_{lo} + {}^{Y_{lo}}/_{Z_{lo}}, X_{hi} + {}^{Y_{hi}}/_{Z_{hi}}$ |
| Addition | $Y_{hi} < 0$ | $Z_{hi} < 0$ | $X_{lo} + {}^{Y_{hi}}/_{Z_{lo}}, X_{hi} + {}^{Y_{lo}}/_{Z_{hi}}$ |
| Addition | $Y_{lo} < 0 < Y_{hi}$ | $Z_{lo} > 0$ | $X_{lo} + {}^{Y_{lo}}/_{Z_{lo}}, X_{hi} + {}^{Y_{hi}}/_{Z_{lo}}$ |
| Addition | $Y_{lo} < 0 < Y_{hi}$ | $Z_{hi} < 0$ | $X_{lo} + {}^{Y_{hi}}/_{Z_{hi}}, X_{hi} + {}^{Y_{lo}}/_{Z_{hi}}$ |
| Subtraction | $Y_{lo} > 0$ | $Z_{lo} > 0$ | $X_{lo} - {}^{Y_{hi}}/_{Z_{lo}}, X_{hi} - {}^{Y_{lo}}/_{Z_{hi}}$ |
| Subtraction | $Y_{lo} > 0$ | $Z_{hi} < 0$ | $X_{lo} - {}^{Y_{lo}}/_{Z_{lo}}, X_{hi} - {}^{Y_{hi}}/_{Z_{hi}}$ |
| Subtraction | $Y_{hi} < 0$ | $Z_{lo} > 0$ | $X_{lo} - {}^{Y_{hi}}/_{Z_{hi}}, X_{hi} - {}^{Y_{lo}}/_{Z_{lo}}$ |
| Subtraction | $Y_{hi} < 0$ | $Z_{hi} < 0$ | $X_{lo} - {}^{Y_{lo}}/_{Z_{hi}}, X_{hi} - {}^{Y_{hi}}/_{Z_{lo}}$ |
| Subtraction | $Y_{lo} < 0 < Y_{hi}$ | $Z_{lo} > 0$ | $X_{lo} - {}^{Y_{hi}}/_{Z_{lo}}, X_{hi} - {}^{Y_{lo}}/_{Z_{lo}}$ |
| Subtraction | $Y_{lo} < 0 < Y_{hi}$ | $Z_{hi} < 0$ | $X_{lo} - {}^{Y_{lo}}/_{Z_{hi}}, X_{hi} - {}^{Y_{hi}}/_{Z_{hi}}$ |

As it can be observed from the above table, interval divide-add fused requires only two floating point operations. This is due to the fact that both interval addition/subtraction and interval division require only two floating point operations. The two operations can be performed either sequentially using only one floating point divide-add fused, either in parallel, using two such units.

## 4.6 Evaluation

### 4.6.1 Accuracy

In order to determine the accuracy of the two proposed designs, a round-off error analysis is performed. The round-off error analysis will consider only rounding towards nearest even, however, it can be easily extended to the other two rounding modes (rounding towards infinity and rounding towards zero). These round-off errors will be compared to the round-off error introduced by the rounding towards nearest even. This method was applied to the analysis of the round-off errors produced by different on-fly rounding algorithms by Ercegovac and Lang in [28].

Considering the size of mantissa to be $m$ (including the hidden one), the round-off error introduced by a correct rounding towards nearest even is equal to $\frac{1}{2} * 2^{1-m}$ [28].

Using the cases presented in Section 4.5.3, for the pro-accuracy implementation, the round-off error is equal to:

- Case I
  In this case the rounding is performed correctly; therefore the rounding error is equal to the one of the rounding towards nearest even: $\frac{1}{2} * 2^{1-m}$.

- Case II
  Also in this particular case the rounding is performed correctly; therefore the rounding error is equal to the one of the rounding towards nearest even: $\frac{1}{2} * 2^{1-m}$.

- Case III
  In this case, the problem of leading zeros can appear. When the number of leading zeros is smaller then $m$, then there are a sufficient number of bits in order to perform a correct rounding. When the number of leading zero is between $m$ and $m+3$, an incorrect rounding bits (sticky, round and guard) computation does appear. In this case the rounding error is equal to $2^{1-m}$, because a truncation may be performed. When the number of leading zeros is greater than $m+3$, the result will consist of $2m+3-l$ bits of the subtraction result in the most significant positions (where $l$ is the number of leading zeros), while the rest will be equal to 0. The rounding error in this case will be equal to $2^{1-m+(l-m-3)}$. If $l > 2m+3$ than the result will be equal to zero.

- Case IV
  In this case two situations do appear. In the first situation, the addend is rightshifted by maximum $m+3$ positions compared to the quotient. In this case, the computation of the rounding bits will be correct and the result will be correctly rounded. In the second case, the addend is rightshifted more than $m+3$ positions compared to the quotient. In the worst case, a carry propagation to the guard and round bit or even to the most significant $m$ bits can appear. In this case the result will not be rounded correctly and the maximum error is $2^{1-m}$ (similar to the error of a truncation).

Regarding the pro-performance implementation, the analysis will be performed in the same way:

- Case I
  In this case the rounding is performed correctly; therefore the rounding error is equal to the one of the rounding towards nearest even: $\frac{1}{2} * 2^{1-m}$.

- Case II

Also in this particular case the rounding is performed correctly; therefore the rounding error is equal to the one of the rounding towards nearest even: $\frac{1}{2} * 2^{1-m}$.

- Case III
  In this case, the problem of leading zeros can appear. When the number of leading zeros is equal to zero, then there are a sufficient number of bits in order to perform a correct rounding. When the number of leading zero is between 1 and 3, an incorrect rounding bits (sticky, round and guard) computation may appear. In this case the rounding error is equal to $2^{1-m}$, because a truncation may be performed. When the number of leading zeros is greater than 3, the result will consist of $m+3-l$ bits of the subtraction result in the most significant positions (where $l$ is the number of leading zeros), while the rest will be equal to 0. The rounding error in this case will be equal to $2^{1-m+(l-3)}$. If $l > m+3$ than the result will be equal to zero.

- Case IV
  In this case a carry propagation to the guard and round bit or even to the most significant $m$ bits can appear. Therefore, the result will not be rounded correctly and the maximum error is $2^{1-m}$ (similar to the error of a truncation).

These two implementations are compared to the accuracy obtained by perfuming the combined operation using a floating point divider and a floating point adder. The round-off error for the result of the floating point divider is equal to $\frac{1}{2} * 2^{1-m}$ [28]. Considering the floating point addition, we obtain the following cases:

- The addend is leftshifted by at least two positions compared to the quotient. In this case the least significant bits of the quotient are used for the computation of the rounding bits (guard, round, sticky). In the worst case, the value of the sticky bit may not be the correct one (due to the fact that in the rounding operations performed in division, quotient bits which could influence the value of the sticky bit were "truncated"). Therefore, in this case the round-off error is $2^{1-m}$. This case corresponds to the first two cases in the analysis performed for floating point divide-add fused.

- The addend is rightshifted by at least two positions compared to the quotient. This case is equivalent to the fourth case in the analysis of the floating point divide-add fused and the round-off error is the same: $2^{1-m}$.

- The difference between exponents is 0 or 1. In the case of an effective subtraction, leading zeros may appear. The result will be comprised of the most significant $m-l$ bits (which were leftshifted during the normalization step) followed by zeros. The maximum round-off error in this case is $2^{1-m+l}$. If $l > m$, than the result will be equal to zero.

Table 4.5 - Maximum round-off errors for divide-add fused

| Implementation | Max Error Addend Leftshifted | Max Error Addend Rightshifted | Max Error Leading Zeros | Min leading zeros for a zero result |
|---|---|---|---|---|
| FP Divider+ FP Adder | $2^{1-m}$ | $2^{1-m}$ | $2^{1-m+l}$, $l > 0$ | $m$ |
| Pro-performance DAF | $\frac{1}{2} * 2^{1-m}$ | $2^{1-m}$ | $2^{1-m+(l-3)}$ $l > 3$ | $m + 3$ |
| Pro-accuracy DAF | $\frac{1}{2} * 2^{1-m}$ | $2^{1-m}$, $d > m$ $\frac{1}{2} * 2^{1-m}$, $d \leq m$ | $2^{1-m+(l-m-3)}$, $l > m + 3$ | $2m + 3$ |

The results of the analysis are summarized in table 4.5. The table presents the maximum error in case when addend is leftshifted compared to the quotient, the maximum error when the addend is rightshifted compared to the quotient, maximum error in case of leading zeros and the minimum number of leading zeros for a zero result.

The analysis shows that even for the pro-performance divide-add fused unit which uses the same number of quotient bits as the solution based on an adder and divider, an increase in accuracy is obtained. Therefore, in terms of accuracy, an implementation of the divide-add fused unit is favorable.

### 4.6.2 Synthesis Results

In order to obtain, two IEEE half precision divide-add fused units have been implemented in VHDL and synthesized using Xilinx Synthesis Tool:

- A pro-accuracy divide-add fused with unrolled SRT stages
- A pro-accuracy divide-add fused with one SRT stage

These four implementations were also compared to a floating point divider obtained using unrolled 14 SRT radix-2 stages and a AMD double path floating point adder.

When implementing the floating point divider and the floating point divide-add fused units, the following modules were used:

- The implemented divider stage was a SRT radix-2, used in the 167 MHz Sun UltraSparc processor [77]. The reason for implementation was its simplicity. However, any other type of SRT division stage can also be used, depending on the desired performance, cost and power consumption (Fig 4. 15).
- The carry propagate adders used were implemented using Bret-Kung carry lookahead computational chains [15].
- The alignment right shifter and the normalization left shifter were implemented as barrel shifters.

Regarding the obtained cost, the results are presented in table 4.6 (for the unfolded and one SRT stage dividers). Fig 4.16 presents a comparison between the

two implemented divide-add fused versus divider plus a double path adder (both unfolded and one SRT stage).

Table 4.6 – Cost of Proposed
Divide-Add Fused Architectures (in LUT-4)

| Architecture | Divider Unrolled |
|---|---|
| Pro-Accuracy | 1904 |
| Pro-Performance | 862 |
| FP Divider+ AMD FP Adder | 776+446 1222 |

As it can be observed from tabel 4.7 and Fig 4.16, the cost of the pro-accuracy implementation using unrolled SRT stages is higher compared to an implementation consisting of a floating point divider and a double path adder. Regarding the cost of the pro-performance implementation a decrease compared to the other two solutions can be observed.

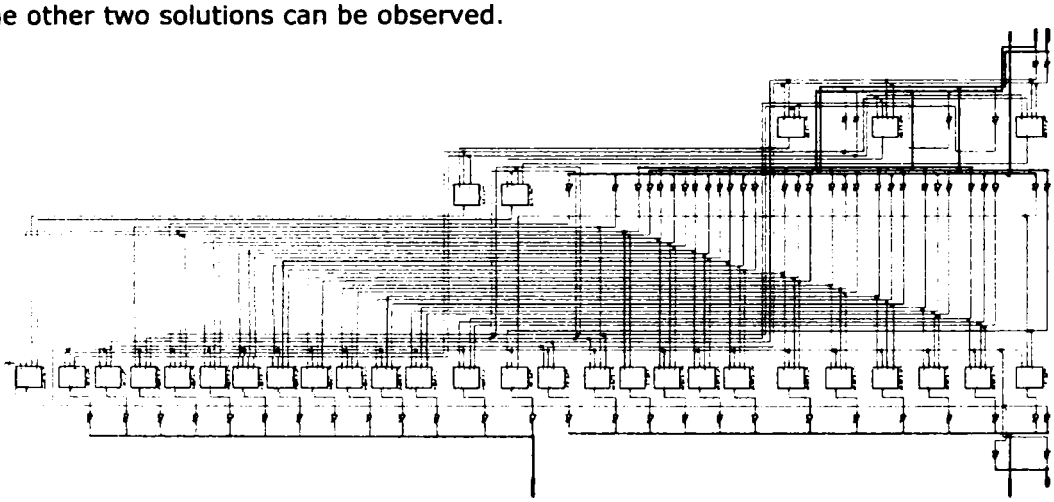


Figure 4.15 – Technology Schematic of the Used SRT Radix-2 Stage Obtained with XST

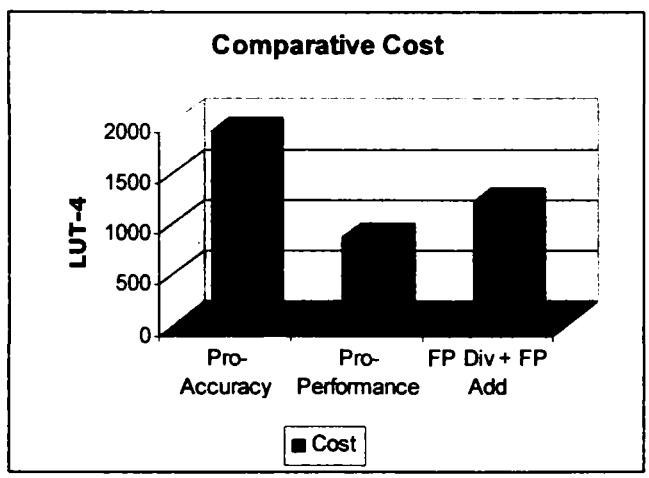



Figure 4.16 – Comparative Cost of the Three Implemented Designs

Regarding the latency as an indicator of performance, the considered latency was for the unrolled SRT stages designs for both the proposed divide-add fused implementations and the floating point division. The results are presented in table 4.7 and Fig 4.19.

Table 4.7 - Latency of Proposed Divide-Add Fused Architectures (in ns)

| Architecture | Divider Unrolled |
|---|---|
| Pro-Accuracy | 112,657 |
| Pro-Performance | 63,546 |
| FP Divider+ AMD FP Adder | 61,746+15,83 77,576 |

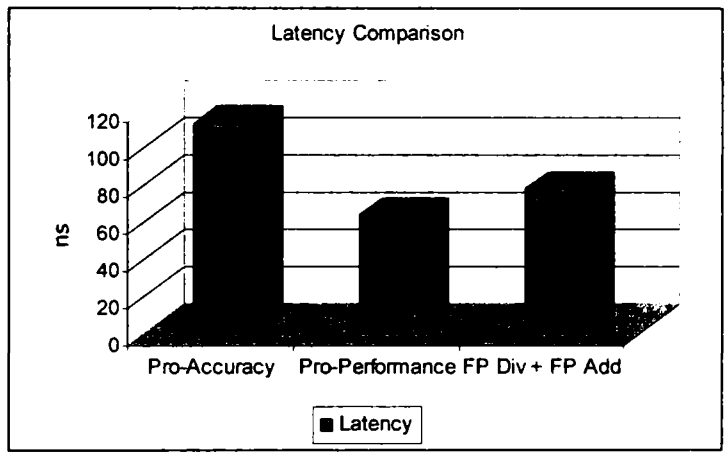



Figure 4.19 - Comparative Latency of the Three Implemented Designs

Regarding the latency, the results show that the pro-performance implementation has a smaller latency compared to the combined floating point divider and floating point adder, while the pro-accuracy implementation have a higher latency, mainly because the number of quotient bits generated is almost double. However, this latency analysis does not consider the penalties imposed by the write back operation into the register file imposed by the storage of the floating point division results.

Therefore, regarding the three considered aspects of the floating point division followed by a floating point addition/subtraction - accuracy, cost and latency - the implementation of a dedicated divide-add fused unit represents an advantage compared to using a floating point adder and a floating point divider.

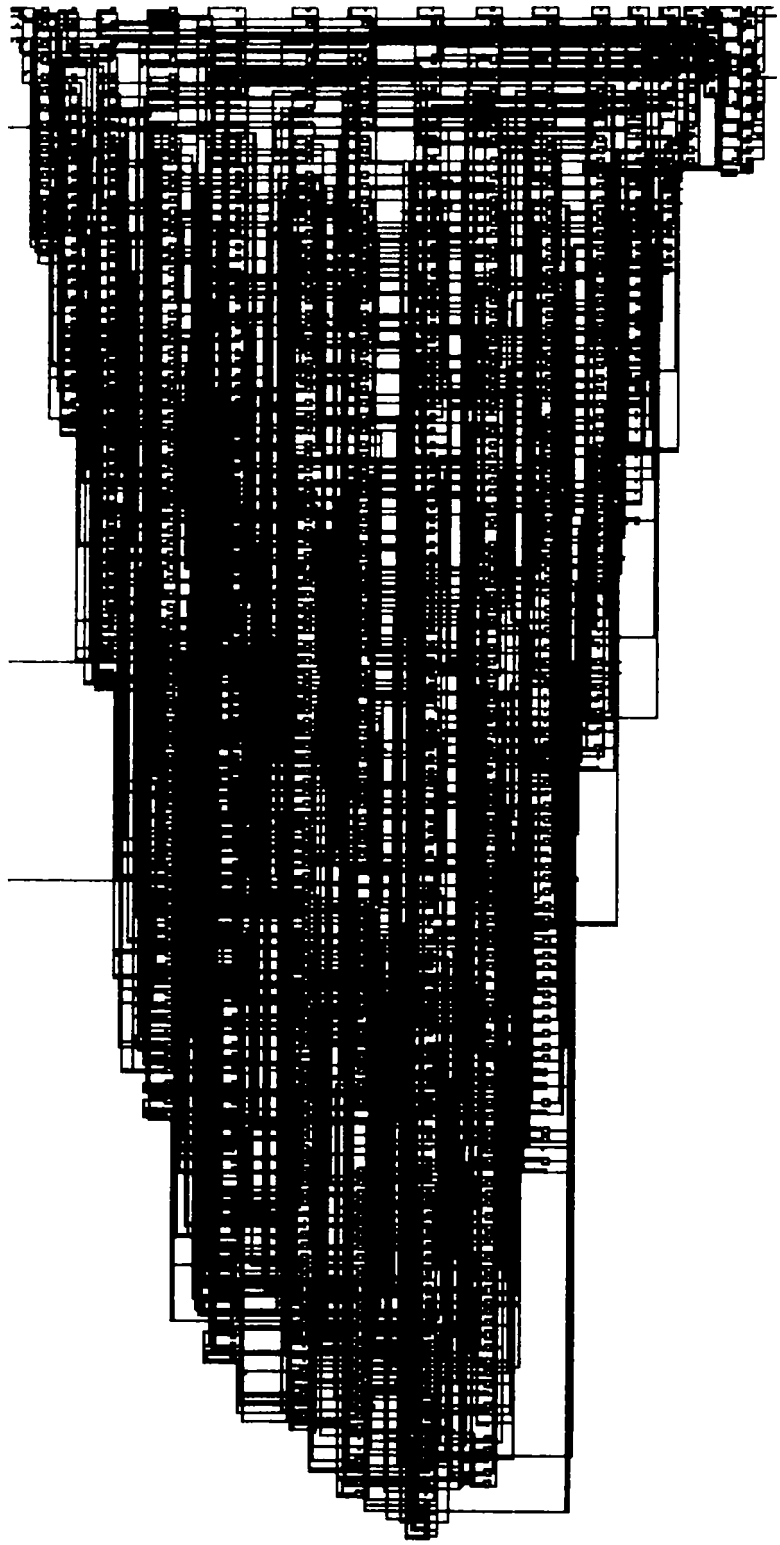


Figure 4.17 – Technology Schematic of Mantissa Datapath for Pro-Performance Divide-Add Fused Obtained with XST

## 4.7 Summary

This chapter is dedicated to the floating point divide-add fused unit, a dedicated arithmetic unit for increasing the performance of a specific interval arithmetic algorithm: the interval Newton's method. To the best of my knowledge, in this thesis is presented the first such unit. In the first section, the reasons for not being implemented such a dedicated hardware unit are presented, which are related to the fact that division followed by addition/subtraction is a very rare operation.

The second section of this chapter presents the interval Newton's method. This represents a powerful interval algorithm for nonlinear equations and systems of equation solving, with applicability in a wide range of fields of applications. This method has several advantages compared to other equation solving numerical algorithms, such as: guaranteed convergence to the solution, quadratic convergence, and no solution situation indicated in the first iteration. The basic arithmetic operations for this method are represented by the division followed by subtraction. Therefore, for this case, a dedicated divide-add fused would bring certain advantages.

The next two sections are dedicated to the floating point multiply-add fused and floating point division. Due to the fact that multiply-add fused has one of its operation the addition and the floating point multiplication is similar to floating point division, it is expected that the architecture of the divide-add fused unit to be similar to the one of the multiply-add fused. Therefore, a detailed presentation of the multiply-add fused is realized in the third section of this chapter. The basic algorithm, enhancements of this algorithm and high performance multiply-add fused architectures are presented. The fourth section is dedicated to the floating point division. An overview of the two most used division algorithm classes is realized. The first analyzed division algorithms class is represented by digit recurrence. These algorithms have as their basic operation the addition. The main design choices, such as radix, quotient digit set, quotient digit computation, partial remainder representation or on-fly conversion and rounding, are presented. The second class of division algorithms analyzed is represented by the multiplicative methods, such as Newton-Raphson and series expansion. A comparison between these two types of division algorithms is realized.

The final two sections present the proposed floating point divide-add fused unit. The algorithm and the according architecture of this dedicated hardware united are detailed. They are similar to the ones used for multiply-add fused, with several changes: instead of the encoding module and the partial product reduction tree a digit-recurrence divider is used. The main issue in designing a floating-point divide-add fused unit is represented by the number of quotient bits needed. A detailed analysis is performed. A large number of quotient bits leads to lower performance and higher cost, but provides a better accuracy of the result. Therefore, a tradeoff must be made between the desired performance and the needed accuracy. Two architectures are proposed, one suitable for better accuracy, while the second is suitable for better performance. Also, variable latency architecture can be used for the two proposed architectures. The last chapter presents the evaluation results. An analysis of the accuracy and comparison with an architecture composed of a division and addition unit is realized. Furthermore, a cost and latency analysis based on synthesis results is realized.

# 5. Conclusions

## 5.1 Context and Relevance

Interval arithmetic has been the focus of the research community for more than four decades. In this years, a wide range of interval applications have been developed, which have proven more reliable and in some cases with higher performance compared to the ones based on conventional floating point arithmetic. However, lack of appropriate hardware support made these interval algorithms and methods to be slow and inefficient. This thesis tackles the problem of designing dedicated floating point units for interval arithmetic. This way, basic incentives for an efficient implementation of interval algorithms have been created.

The importance and the need for hardware units for interval arithmetic are proven by the last initiatives of the IEEE Standards Association – Microprocessor Standards Committee:

- The first initiative is represented by the revision and the extension of the IEEE 754 standard for floating point arithmetic which is expected to be released by the end of 2008.
- The second initiative is represented by the formation of the Work Group 1788 for developing an IEEE standard for interval arithmetic [116]. The future standard – IEEE 1788 standard for interval arithmetic – is expected to be completed by the end of 2012.

Three dedicated floating point unit have been designed and implemented. The first two are dedicated for the most frequent operations in any arithmetic system: addition and multiplication. This way, the basis for any interval arithmetic system has been laid.

The third designed and implemented module is the floating point divide-add fused. This unit comes as a dedicated floating point hardware circuit for an interval arithmetic specific algorithm: the interval Newton's algorithm. The interval Newton's method represents one of the most important interval algorithms, with a wide range of applications. The proposed unit comes in the context of designing more and more dedicated hardware for both System-on-Chip and accelerators implemented in FPGA based circuits [119].

Furthermore, all three units can be used for performing conventional floating point operations. This goal represents a necessity for interval arithmetic units, as combined interval and conventional floating point applications might be used. Furthermore, several crucial interval operators (such as midpoint and width of an interval) rely on conventional floating point arithmetic.

## 5.2 Summary

This thesis has three main chapters, each of them dedicated to the corresponding floating point hardware unit.

The first hardware arithmetic unit is dedicated to the interval addition, treated in Chapter 2. The first section of this chapter presents the two solutions for the interval addition, which are based on the usage of conventional floating point adders. The following subsection is dedicated to the conventional floating point addition. The basic algorithm and the enhancements for this basic algorithm are detailed. These enhancements of the basic floating point addition algorithm represent the backbone of the single path floating point adder. A detailed presentation of the double path adder follows. Several high performance floating point adders based on the double adder architecture are described, including both academic and commercial designs. The third section of the addition dedicated chapter presents then proposed addition unit, which is based on the double path adders. The proposed adder exploits the parallel structure of it, by trying to perform the two operations required for an interval addition/subtraction simultaneously. This represents a novel approach in the design of double path adders. In the last section of this chapter, performance and cost evaluations were performed for the proposed adders compared to other commercial and academic double path adder based solutions. These evaluations showed that the proposed adder presents the best cost/performance tradeoff with respect to other interval adders. Furthermore, the proposed adder can be used for increasing the performance of the conventional addition, due to its increased throughput.

The third chapter presents the interval multiplication. Unlike addition, interval multiplication represents a difficult operation. Therefore, a number of algorithms have been developed in order to improve the performance of this crucial operation. These algorithms are presented in the first section of this chapter. Further, an overview of the conventional floating point multiplication is given in the second section. Tree multipliers are presented, different tree topologies being analyzed. Finally, three rounding schemes for floating point multiplication are presented: the Even-Seidel, Quach and Yu-Zyner. The third section of Chapter 3 is dedicated to the proposed interval multiplier. Algorithm for interval multiplication and its corresponding architecture are presented. The architecture is based on the dual result multiplier and two floating point comparators. The three rounding schemes presented in the previous section are modified for dual result multipliers. A new rounding scheme is proposed. In the last section, performance and cost estimates are realized. These estimates show an improvement in the worst case performance for the proposed interval multiplication architecture. Furthermore, the proposed rounding scheme has the lowest latency and lowest cost.

The fourth chapter is dedicated to the floating point divide-add fused hardware unit. Due to the fact that divide-add fused have never been implemented yet, to the best of my knowledge, the first sections of this chapter presents why such a dedicated hardware unit has never been yet designed and the reasons for which in the context of interval arithmetic, such a unit may prove and advantage. The reasons for such a unit are related to the accelerating the interval Newton's method, which is presented in detail in this chapter. The following sections present the floating point multiply-add fused units and the floating point division, which represent the inspiration point for such a hardware unit. The fifth section of this chapter is dedicated to the design and implementation of the floating point divide-add fused unit. The algorithm and its architecture are inspired from the ones used for floating point multiply-add fused. The main difference is represented by the usage of a divider stage instead of the encoding module and the partial product reduction tree. The main issues regarding the implementation of this dedicated unit are analyzed. These two issues are related to the number of quotient bits needed

and the rounding in floating point divide-add fused. An analysis of these two problems is performed. Last, but not least, performance and cost evaluations are performed.

## 5.3 Contributions

This section will present the contributions of this thesis linked to the three research directions pointed in Section 5.1. The motivations for these three directions have been provided in Section 1.3 of the introductory chapter.

- **Interval Addition Unit**

Regarding the most frequent operation in any arithmetic system – addition/subtraction – the major contribution is represented by the design ***of a new adder architecture***. The proposed adder architecture is based on the double path adder structure and exploits the parallelism of this structure, by performing the two floating point additions required for the interval operation. The main advantages of the proposed adder, as they resulted from the cost and performance estimates, are:

- Higher performance compared to the interval adder based on a single double path adder
- Lower cost compared to the interval adder based on two floating point adders
- High throughput compared to other interval adder architectures
- Best cost/performance cost tradeoff when dealing with high number of interval additions.

The limitations of the proposed interval adder are:

- Higher cost compared to the interval adder based on a single double path adder
- Lower performance compared to the interval adder based on two floating point adder
- Low performance when dealing with isolated additions, as the proposed adder's main characteristic is represented by the throughput

The proposed adder can also be used for increasing the throughput of the conventional floating point addition, due to the parallel structure of it. Thus, as estimates results show, the proposed adder present the highest performance and the highest cots-performance tradeoff when dealing with series of large numbers of additions. Due to higher latency compared to other floating point addition implementations, the proposed adder is not suitable for single additions.

- **Interval Multiplication Unit**

Regarding the interval multiplication, the following contributions have been proposed in this thesis:

- Interval multiplication algorithm based on the pipelined interval multiplication proposed by Kulisch and the eight product multiplication algorithm
- Novel interval multiplier architecture based on a dual result multiplier and two floating point comparators

- Dual result multiplier based on tree based floating point multipliers
- Novel addition and rounding units for dual result multipliers based on the Quach rounding scheme and on the Yu-Zyner rounding schemes for conventional floating point multiplication
- Novel addition and rounding units for dual result multiplier for usage exclusively for interval arithmetic

The advantages of the proposed interval multiplication unit based on the dual result multiplier are:

- Higher worst case performance compared to other interval multiplication algorithm
- Increased functionality, as the proposed unit can also be used for conventional floating point multiplication and interval set operations based on comparisons

The disadvantage of the proposed unit relies in a lower medium performance compared to other interval multipliers.

Regarding the proposed addition and rounding multiplication scheme for interval arithmetic, the main advantages are lower latency and lower cost compared to the rounding units based on Quach rounding scheme and on Yu-Zyner rounding scheme.

- **Floating Point Divide-Add Fused**

The floating point divide-add fused has never been implemented, to the best of my knowledge. Therefore, a motivation had to be provided for implementing this dedicated combined operations unit. This motivation has been provided by the one of the most powerful interval arithmetic algorithms: interval Newton's method. Thus, the contributions regarding the floating point divide-add fused are:

- Algorithm for floating point divide-add fused inspired from the multiply-add fused algorithm
- Architecture for divide-add fused inspired from the basic multiply-add fused architecture based on digit-recurrence dividers
- A detailed analysis of the number of quotient bits required for performing the divide-add fused architecture
- Variable latency architecture for divide-add fused units

This way, a specific divide-add fused floating point unit for interval Newton's method have been proposed and implemented. Therefore, performance and precision increases for one of the most powerful interval arithmetic algorithm has been provided.

## 5.4 Future Work

This thesis has addressed immediate problems regarding the design of the floating point hardware units for interval arithmetic. However, as the future IEEE 1788 standard for interval arithmetic will be developed, several modification of the proposed design will have to be done in order to be fully compliant to the standard. The proposed units didn't address specific problems closely related to an arithmetic

standards such as the special quantities (such as infinity, zero, NaN used in the IEEE 754) or the exceptions and their handling.

The second research direction is represented by the combining the divide-add fused architectures with other division algorithms classes, such as the multiplication methods (Newton-Raphson, series expansion) and very high radix algorithms. This way, a wide variety of divide-add fused floating point units with different performance/cost/power characteristics should be provided. Furthermore, higher performance divide-add fused architectures inspired from multiply-add fused structure are considered for future implementations.

The third research direction will be the design of interval function evaluators in order to further increase the performance of the interval Newton's method and other interval algorithms which make extensive use of function evaluators. Thus, hardware implementations for functions such exponentiation, logarithmic, square root or function describing surfaces, curves, etc. This way, dedicated hardware accelerators for specific applications which used interval arithmetic algorithms, such as the computer graphics, will be considered for implementation.

# Appendix A
# VHDL Source Code and Technology Schematics for ISCAS'85 Benchmark Circuits

In order to increase the confidence of the obtained synthesis results, five ISCAS'85 benchmark circuits were modeled in VHDL and synthesized with Xilinx Synthesis Tool (XST) for Xilinx Virtex-4 FPGA. These circuits are described in [39], while their Verilog HDL source codes can be found at http://www.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html [118].

## A1. C17 Benchmark Circuit

```
entity C17 is
   Port ( in_g1 : in  STD_LOGIC;
          in_g2 : in  STD_LOGIC;
          in_g3 : in  STD_LOGIC;
          in_g4 : in  STD_LOGIC;
          in_g5 : in  STD_LOGIC;
          out_g1 : out  STD_LOGIC;
          out_g2 : out  STD_LOGIC);
end C17;

architecture Behavioral of C17 is
signal g11,g12,g21,g22:std_logic;
begin
          Nand1_1:  g11<=in_g1 nand in_g3;
          Nand1_2:  g12<=in_g3 nand in_g4;
          Nand2_1:  g21<=in_g2 nand g12;
          Nand2_2:  g22<=in_g5 nand g12;
          Nand3_1:  out_g1<=g11 nand g21;
          Nand3_2:  out_g2<=g21 nand g22;
end Behavioral;
```

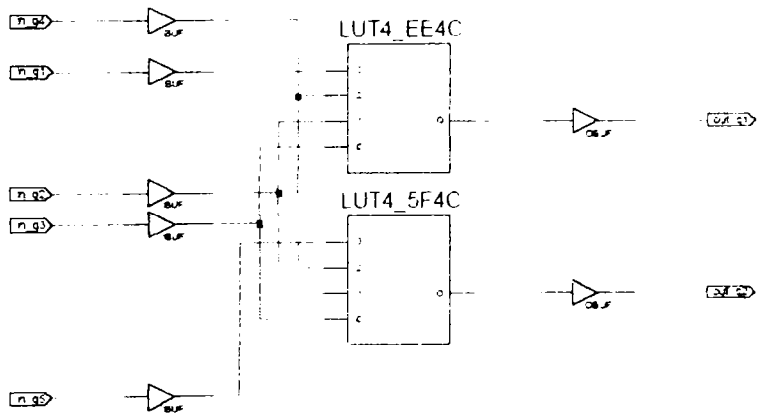



Figure A1 – Technology Schematic for C17 Obtained with XST

## A2. C432 Benchmark Circuit

```
entity C432 is
   Port ( e : in  STD_LOGIC_VECTOR (8 downto 0);
          a : in  STD_LOGIC_VECTOR (8 downto 0);
          b : in  STD_LOGIC_VECTOR (8 downto 0);
          c : in  STD_LOGIC_VECTOR (8 downto 0);
          pa : out  STD_LOGIC;
          pb : out  STD_LOGIC;
          pc : out  STD_LOGIC;
          Chan : out  STD_LOGIC_VECTOR (3 downto 0));
end C432;

architecture Struct of C432 is
component m1
          port (a:in std_logic_vector (8 downto 0);
                                e:in std_logic_vector (8 downto 0);
                                pa:out std_logic;
                                x1:out std_logic_vector (8 downto 0));
end component;
component m2
          port (x1 : in  STD_LOGIC_VECTOR (8 downto 0);
        b : in  STD_LOGIC_VECTOR (8 downto 0);
        e : in  STD_LOGIC_VECTOR (8 downto 0);
        pb : out  STD_LOGIC;
        x2 : out  STD_LOGIC_VECTOR (8 downto 0));
end component;
component m3
          Port ( x1 : in  STD_LOGIC_VECTOR (8 downto 0);
         x2 : in  STD_LOGIC_VECTOR (8 downto 0);
         c : in  STD_LOGIC_VECTOR (8 downto 0);
         e : in  STD_LOGIC_VECTOR (8 downto 0);
         pc : out  STD_LOGIC);
end component;
component m4
           Port ( e : in  STD_LOGIC_VECTOR (8 downto 0);
          a : in  STD_LOGIC_VECTOR (8 downto 0);
          pa : in  STD_LOGIC;
          b : in  STD_LOGIC_VECTOR (8 downto 0);
          pb : in  STD_LOGIC;
          c : in  STD_LOGIC_VECTOR (8 downto 0);
          pc : in  STD_LOGIC;
          i : out  STD_LOGIC_VECTOR (8 downto 0));
end component;
component m5
          Port ( i : in  STD_LOGIC_VECTOR (8 downto 0);
         chan : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

signal pa_int, pb_int,pc_int:std_logic;
signal x1,x2,i:std_logic_vector (8 downto 0);
begin
          M1_logic: m1 port map (a, e, pa_int, x1);
          M2_logic: m2 port map (x1, b, e, pb_int, x2);
          M3_logic: m3 port map (x1, x2, c, e, pc_int);
          M4_logic: m4 port map (e, a, pa_int, b, pb_int, c, pc_int, i);
          M5_logic: m5 port map (i, chan);
```

```
        pa<=pa_int;
        pb<=pb_int;
        pc<=pc_int;
end Struct;

entity m1 is
   Port ( a : in  STD_LOGIC_VECTOR (8 downto 0);
        e : in  STD_LOGIC_VECTOR (8 downto 0);
        pa : out  STD_LOGIC;
        x1 : out  STD_LOGIC_VECTOR (8 downto 0));
end m1;

architecture Behavioral of m1 is
signal g1:std_logic_vector (8 downto 0);
signal pa_gen:std_logic;
begin
        G1_logic:
        for i in 8 downto 0 generate
                g1(i)<= (not a(i)) nand e(i);
        end generate;

        PA_logic:
        pa_gen<=not (g1(0) and g1(1) and g1(2) and g1(3) and g1(4)
                        and g1(5) and g1(6) and g1(7) and g1(8));
        pa<=pa_gen;

        X1_logic:
        for i in 8 downto 0 generate
                x1(i)<=pa_gen xor g1(i);
        end generate;
end Behavioral;

entity m2 is
   Port ( x1 : in  STD_LOGIC_VECTOR (8 downto 0);
        b : in  STD_LOGIC_VECTOR (8 downto 0);
        e : in  STD_LOGIC_VECTOR (8 downto 0);
        pb : out  STD_LOGIC;
        x2 : out  STD_LOGIC_VECTOR (8 downto 0));
end m2;
architecture Behavioral of m2 is
signal g1, g2: std_logic_vector(8 downto 0);
signal pb_gen: std_logic;
begin
        G1G2_logic:
        for i in 8 downto 0 generate
                g1(i)<=(not e(i)) nor b(i);
                g2(i)<=g1(i) nand x1(i);
        end generate;
        PB_logic:
                pb_gen<= not (g2(0) and g2(1) and g2(2)
                                and g2(3) and g2(4) and g2(5)
                                and g2(6) and g2(7) and g2(8));
                pb<=pb_gen;
        x2_logic:
        for i in 0 to 8 generate
                x2(i)<=pb_gen xor g2(i);
        end generate;
end Behavioral;
```

```
entity m3 is
   Port ( x1 : in  STD_LOGIC_VECTOR (8 downto 0);
          x2 : in  STD_LOGIC_VECTOR (8 downto 0);
          c : in  STD_LOGIC_VECTOR (8 downto 0);
          e : in  STD_LOGIC_VECTOR (8 downto 0);
          pc : out  STD_LOGIC);
end m3;

architecture Behavioral of m3 is
signal g1,rc:std_logic_vector(8 downto 0);
begin
          g1_logic:
          for i in 8 downto 0 generate
                    g1(i)<= (not e(i)) nor c(i);
          end generate;
          RC_logic:
          for i in 8 downto 0 generate
                    rc(i)<= not (g1(i) and x1(i) and x2(i));
          end generate;
          PC_logic:
                    pc<=not (rc(0) and rc(1) and rc(2) and rc(3)
                              and rc(4) and rc(5) and rc(6) and rc(7)
                              and rc(8));
end Behavioral;

entity m4 is
   Port ( e : in  STD_LOGIC_VECTOR (8 downto 0);
          a : in  STD_LOGIC_VECTOR (8 downto 0);
          pa : in  STD_LOGIC;
          b : in  STD_LOGIC_VECTOR (8 downto 0);
          pb : in  STD_LOGIC;
          c : in  STD_LOGIC_VECTOR (8 downto 0);
          pc : in  STD_LOGIC;
          i : out  STD_LOGIC_VECTOR (8 downto 0));
end m4;
architecture Behavioral of m4 is
signal ga,gb,gc:std_logic_vector(8 downto 0);
begin
          i_logic:
          for j in 8 downto 0 generate
                    GA_logic: ga(j)<= a(j) nand pa;
                    GB_logic: gb(j)<= b(j) nand pb;
                    GC_logic: gc(j)<= c(j) nand pc;
                    I_generation:     i(j)<= not (e(j) and ga(j) and gb(j) and gc(j));
          end generate;
end Behavioral;

entity m5 is
   Port ( i : in  STD_LOGIC_VECTOR (8 downto 0);
          chan : out  STD_LOGIC_VECTOR (3 downto 0));
end m5;

architecture Behavioral of m5 is
signal i7_0, i6_5, i5_2, i6_3, i6_1: std_logic;
begin
          chan3_logic:
                    i7_0<=i(7),and i(6) and i(5) and i(4)
                                        and i(3) and i(2) and i(1) and i(0);
```

```
                chan(3)<= not ( (not i(8)) nor i7_0);
        chan2_logic:
                i6_5<= not i(5) nand i(6);
                chan(2)<=not (i(7) and i(6) and i(4) and i6_5);
        chan1_logic:
                i5_2<= not (i(5) and i(4) and (not i(2)));
                chan(1)<=not (i(7) and i(6) and i5_2 and i6_3);
        chan9_logic:
                i6_3<= not (i(6) and i(5) and i(4) and i(3));
                i6_1<= not (i(6) and i(5) and i(2) and (not i(1)));
                chan(0)<= not (i(7) and i6_5 and i6_3 and i6_1);
end Behavioral;
```

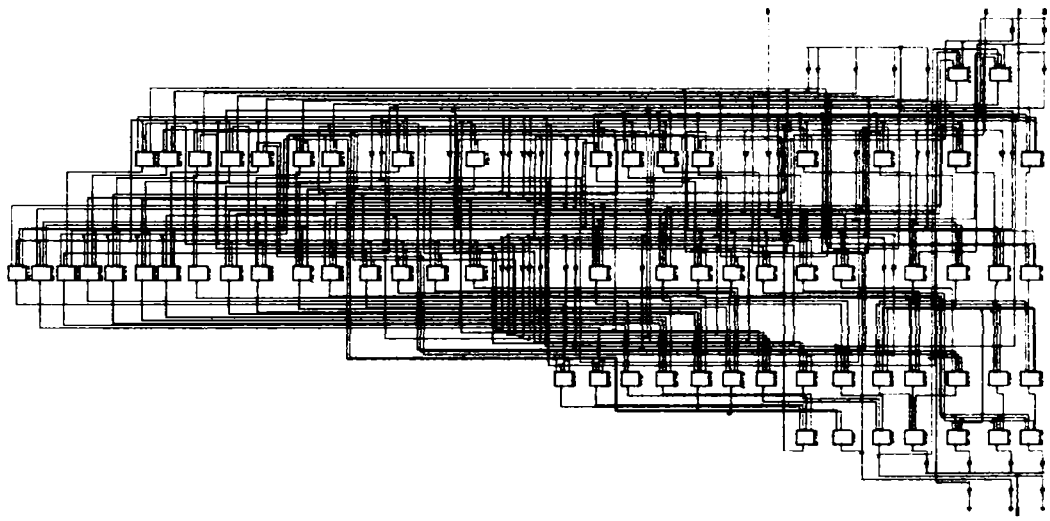


Figure A2 – Technology Schematic of C432 Benchmark Circuit Obtained with XST

## A3. C499 Benchmark Circuit

```
entity c499 is
   Port ( r : in  STD_LOGIC;
          ic : in  STD_LOGIC_VECTOR (7 downto 0);
          id : in  STD_LOGIC_VECTOR (31 downto 0);
          od : out  STD_LOGIC_VECTOR (31 downto 0));
end c499;

architecture Struct of c499 is
component m1
          Port ( r : in  STD_LOGIC;
          ic : in  STD_LOGIC_VECTOR (7 downto 0);
          id : in  STD_LOGIC_VECTOR (31 downto 0);
          s : out  STD_LOGIC_VECTOR (7 downto 0));
end component;
component m2
          Port ( s : in  STD_LOGIC_VECTOR (7 downto 0);
          id : in  STD_LOGIC_VECTOR (31 downto 0);
          od : out  STD_LOGIC_VECTOR (31 downto 0));
end component;
signal s:std_logic_vector (7 downto 0);
begin
```

```
        M1_logic: m1 port map(r, ic, id, s);
        M2_logic: m2 port map(s, id, od);
end Struct;

entity m1 is
   Port ( r : in  STD_LOGIC;
        ic : in  STD_LOGIC_VECTOR (7 downto 0);
        id : in  STD_LOGIC_VECTOR (31 downto 0);
        s : out  STD_LOGIC_VECTOR (7 downto 0));
end m1;

architecture Behavioral of m1 is
begin
   s(0)<=(id(0) xor id(4) xor id(8) xor id(12)) xor (id(16) xor id(17) xor id(18) xor id(19))
                  xor (id(20) xor id(21) xor id(22) xor id(23)) xor (r and ic(0));
   s(1)<=(id(1) xor id(5) xor id(9) xor id(13)) xor (id(24) xor id(25) xor id(26) xor id(27)) xor
                  (id(28) xor id(29) xor id(30) xor id(31)) xor (r and ic(1));
   s(2)<=(id(2) xor id(5) xor id(10) xor id(14)) xor  (id(16) xor id(17) xor id(18) xor id(19))
                  xor (id(24) xor id(25) xor id(26) xor id(27)) xor (r and ic(2));
   s(3)<=(id(3) xor id(7) xor id(11) xor id(15)) xor (id(20) xor id(21) xor id(22) xor id(23))
                  xor (id(28) xor id(29) xor id(30) xor id(31)) xor (r and ic(3));
   s(4)<=(id(16) xor id(20) xor id(24) xor id(28)) xor (id(0) xor id(1) xor id(2) xor id(3))
                  xor (id(4) xor id(5) xor id(6) xor id(7)) xor (r and ic(4));
   s(5)<=(id(17) xor id(21) xor id(25) xor id(29)) xor (id(8) xor id(9) xor id(10) xor id(11))
                  xor (id(12) xor id(13) xor id(14) xor id(15)) xor (r and ic(5));
   s(6)<=(id(18) xor id(22) xor id(26) xor id(30)) xor (id(0) xor id(1) xor id(2) xor id(3))
                  xor (id(8) xor id(9) xor id(10) xor id(11)) xor (r and ic(6));
   s(7)<=(id(19) xor id(23) xor id(27) xor id(31)) xor (id(4) xor id(5) xor id(6) xor id(7)) xor
                  (id(12) xor id(13) xor id(14) xor id(15)) xor (r and ic(7));
end Behavioral;

entity m2 is
   Port ( s : in  STD_LOGIC_VECTOR (7 downto 0);
        id : in  STD_LOGIC_VECTOR (31 downto 0);
        od : out  STD_LOGIC_VECTOR (31 downto 0));
end m2;

architecture Behavioral of m2 is
begin
        od(0)<= ((s(0) and (not s(1)) and (not s(2)) and (not s(3)))and
                        (s(4) and (not s(5)) and s(6) and (not s(7)))) xor id(0);
        od(1)<= (((not s(0)) and s(1) and (not s(2)) and (not s(3)))and
                        (s(4) and (not s(5)) and s(6) and (not s(7)))) xor id(1);
        od(2)<= (((not s(0)) and (not s(1)) and s(2) and (not s(3)))  and
                        (s(4) and (not s(5)) and s(6) and (not s(7)))) xor id(2);
        od(3)<= (((not s(0)) and (not s(1)) and (not s(2)) and s(3))  and
                        (s(4) and (not s(5)) and s(6) and (not s(7)))) xor id(3);
        od(4)<= ((s(0) and (not s(1)) and (not s(2)) and (not s(3)))  and
                        (s(4) and (not s(5)) and (not s(6)) and s(7))) xor id(4);
        od(5)<= (((not s(0)) and s(1) and (not s(2)) and (not s(3)))  and
                        (s(4) and (not s(5)) and (not s(6)) and s(7))) xor id(5);
        od(6)<= (((not s(0)) and (not s(1)) and s(2) and (not s(3)))  and
                        (s(4) and (not s(5)) and (not s(6)) and s(7))) xor id(6);
        od(7)<= (((not s(0)) and (not s(1)) and (not s(2)) and s(3))  and
                        (s(4) and (not s(5)) and (not s(6)) and s(7))) xor id(7);
        od(8)<= ((s(0) and (not s(1)) and (not s(2)) and (not s(3)))  and
                        ((not s(4)) and s(5) and s(6) and (not s(7)))) xor id(8);
```

```
        od(9)<= (((not s(0)) and s(1) and (not s(2)) and (not s(3)))  and
                        ((not s(4)) and s(5) and s(6) and (not s(7)))) xor id(9);
        od(10)<= (((not s(0)) and (not s(1)) and s(2) and (not s(3)))and
                        ((not s(4)) and s(5) and s(6) and (not s(7)))) xor id(10);
        od(11)<= (((not s(0)) and (not s(1)) and (not s(2)) and s(3))and
                        ((not s(4)) and s(5) and s(6) and (not s(7)))) xor id(11);
        od(12)<= ((s(0) and (not s(1)) and (not s(2)) and (not s(3)))and
                        ((not s(4)) and s(5) and (not s(6)) and s(7))) xor id(12);
        od(13)<= (((not s(0)) and s(1) and (not s(2)) and (not s(3))) and
                        ((not s(4)) and s(5) and (not s(6)) and s(7))) xor id(13);
        od(14)<= (((not s(0)) and (not s(1)) and s(2) and (not s(3))) and
                        ((not s(4)) and s(5) and (not s(6)) and s(7))) xor id(14);
        od(15)<= (((not s(0)) and (not s(1)) and (not s(2)) and s(3))and
                        ((not s(4)) and s(5) and (not s(6)) and s(7))) xor id(15);
        od(16)<= ((s(4) and (not s(5)) and (not s(6)) and (not s(7)))and
                        (s(0) and (not s(1)) and s(2) and (not s(3)))) xor id(16);
        od(17)<= (((not s(4)) and s(5) and (not s(6)) and (not s(7)))and
                        (s(0) and (not s(1)) and s(2) and (not s(3)))) xor id(17);
        od(18)<= (((not s(4)) and (not s(5)) and s(6) and (not s(7)))and
                        (s(0) and (not s(1)) and s(2) and (not s(3)))) xor id(18);
        od(19)<= (((not s(4)) and (not s(5)) and (not s(6)) and s(7)) and
                        (s(0) and (not s(1)) and s(2) and (not s(3)))) xor id(19);
        od(20)<= ((s(4) and (not s(5)) and (not s(6)) and (not s(7)))and
                        (s(0) and (not s(1)) and (not s(2)) and s(3))) xor id(20);
        od(21)<= (((not s(4)) and s(5) and (not s(6)) and (not s(7)))
                        and (s(0) and (not s(1)) and (not s(2)) and s(3))) xor id(21);
        od(22)<= (((not s(4)) and (not s(5)) and s(6) and (not s(7)))
                        and (s(0) and (not s(1)) and (not s(2)) and s(3))) xor id(22);
        od(23)<= (((not s(4)) and (not s(5)) and (not s(6)) and s(7))
                        and (s(0) and (not s(1)) and (not s(2)) and s(3))) xor id(23);
        od(24)<= ((s(4) and (not s(5)) and (not s(6)) and (not s(7)))
                        and ((not s(0)) and s(1) and s(2) and (not s(3)))) xor id(24);
        od(25)<= (((not s(4)) and s(5) and (not s(6)) and (not s(7)))
                        and ((not s(0)) and s(1) and s(2) and (not s(3)))) xor id(25);
        od(26)<= (((not s(4)) and (not s(5)) and s(6) and (not s(7)))
                        and ((not s(0)) and s(1) and s(2) and (not s(3)))) xor id(26);
        od(27)<= (((not s(4)) and (not s(5)) and (not s(6)) and s(7))
                        and ((not s(0)) and s(1) and s(2) and (not s(3)))) xor id(27);
        od(28)<= ((s(4) and (not s(5)) and (not s(6)) and (not s(7)))
                        and ((not s(0)) and s(1) and (not s(2)) and s(3))) xor id(28);
        od(29)<= (((not s(4)) and s(5) and (not s(6)) and (not s(7)))
                        and ((not s(0)) and s(1) and (not s(2)) and s(3))) xor id(29);
        od(30)<= (((not s(4)) and (not s(5)) and s(6) and (not s(7)))
                        and ((not s(0)) and s(1) and (not s(2)) and s(3))) xor id(30);
        od(31)<= (((not s(4)) and (not s(5)) and (not s(6)) and s(7))
                        and ((not s(0)) and s(1) and (not s(2)) and s(3))) xor id(31);

end Behavioral;
```

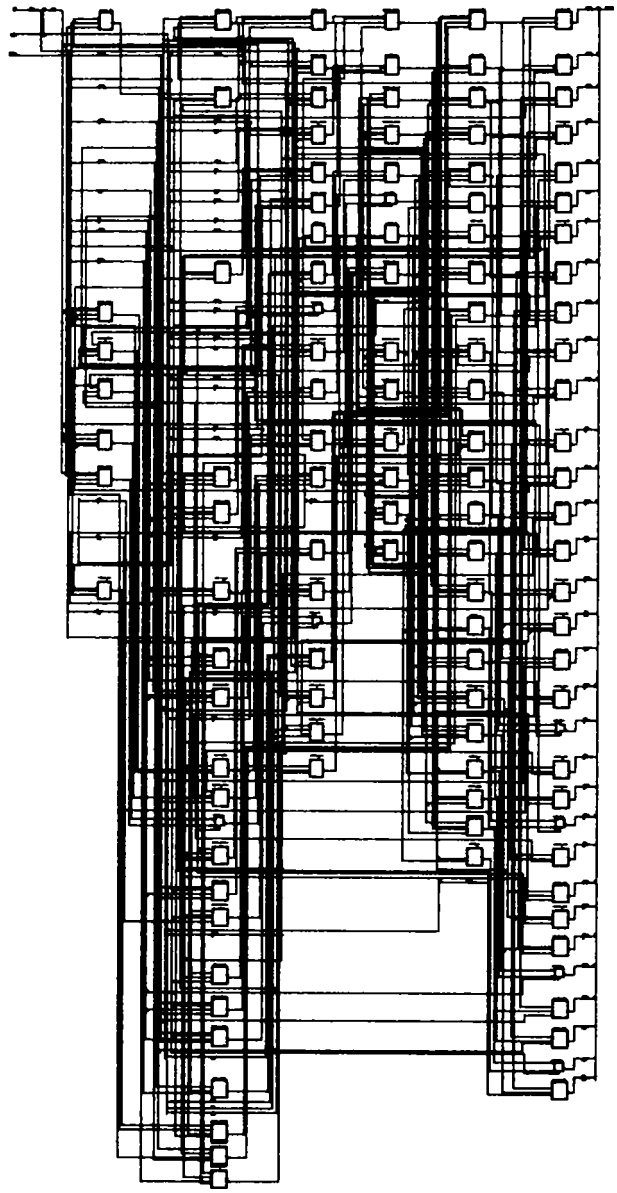


Figure A3 – Technology Schematic of C499 Benchmark Circuit Obtained with XST

## A4 C6288 Benchmark Circuit

```
entity C6288 is
   Port ( a : in  STD_LOGIC_VECTOR (15 downto 0);
          b : in  STD_LOGIC_VECTOR (15 downto 0);
          p : out  STD_LOGIC_VECTOR (31 downto 0));
end C6288;

architecture Struct of C6288 is

type matrix is array(15 downto 0) of std_logic_vector(15 downto 0);
```

```
component HA
        Port ( a : in  STD_LOGIC;
       b : in  STD_LOGIC;
       s : out  STD_LOGIC;
          cout:out STD_LOGIC);
end component;

component FA
        Port ( a : in  STD_LOGIC;
       b : in  STD_LOGIC;
       cin : in  STD_LOGIC;
       s : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end component;

signal line0, line1, line2, line3, line4, line5, line6, line7: std_logic_vector (15 downto 0);
signal line8, line9, line10, line11, line12, line13, line14, line15: std_logic_vector (15 downto
0);
signal s,c:matrix;

begin
        line_generation:
        for i in 0 to 15 generate
                line0(i)<=a(i) and b(0);
                line1(i)<=a(i) and b(1);
                line2(i)<=a(i) and b(2);
                line3(i)<=a(i) and b(3);
                line4(i)<=a(i) and b(4);
                line5(i)<=a(i) and b(5);
                line6(i)<=a(i) and b(6);
                line7(i)<=a(i) and b(7);
                line8(i)<=a(i) and b(8);
                line9(i)<=a(i) and b(9);
                line10(i)<=a(i) and b(10);
                line11(i)<=a(i) and b(11);
                line12(i)<=a(i) and b(12);
                line13(i)<=a(i) and b(13);
                line14(i)<=a(i) and b(14);
                line15(i)<=a(i) and b(15);
        end generate;

        matrix_generation:
        for i in 1 to 15 generate
                HA_line0:
                        HA port map (line0(i), line1(i-1), s(0)(i-1), c(0)(i));
                s(0)(15)<=line1(15);
                c(0)(0)<='0';

                FA_line1:
                        FA port map(line2(i-1), s(0)(i), c(0)(i), s(1)(i-1), c(1)(i));
                s(1)(15)<=line2(15);
                c(1)(0)<='0';

                FA_line2:
                        FA port map(line3(i-1), s(1)(i), c(1)(i), s(2)(i-1), c(2)(i));
                s(2)(15)<=line3(15);
                c(2)(0)<='0';
```

```
FA_line3:
        FA port map(line4(i-1), s(2)(i), c(2)(i), s(3)(i-1), c(3)(i));
s(3)(15)<=line4(15);
c(3)(0)<='0';

FA_line4:
        FA port map(line5(i-1), s(3)(i), c(3)(i), s(4)(i-1), c(4)(i));
s(4)(15)<=line5(15);
c(4)(0)<='0';

FA_line5:
        FA port map(line6(i-1), s(4)(i), c(4)(i), s(5)(i-1), c(5)(i));
s(5)(15)<=line6(15);
c(5)(0)<='0';

FA_line6:
        FA port map(line7(i-1), s(5)(i), c(5)(i), s(6)(i-1), c(6)(i));
s(6)(15)<=line7(15);
c(6)(0)<='0';

FA_line7:
        FA port map(line8(i-1), s(6)(i), c(6)(i), s(7)(i-1), c(7)(i));
s(7)(15)<=line8(15);
c(7)(0)<='0';

FA_line8:
        FA port map(line9(i-1), s(7)(i), c(7)(i), s(8)(i-1), c(8)(i));
s(8)(15)<=line9(15);
c(8)(0)<='0';

FA_line9:
        FA port map(line10(i-1), s(8)(i), c(8)(i), s(9)(i-1), c(9)(i));
s(9)(15)<=line10(15);
c(9)(0)<='0';

FA_line10:
        FA port map(line11(i-1), s(9)(i), c(9)(i), s(10)(i-1), c(10)(i));
s(10)(15)<=line7(15);
c(10)(0)<='0';

FA_line11:
        FA port map(line12(i-1), s(10)(i), c(10)(i), s(11)(i-1), c(11)(i));
s(11)(15)<=line8(15);
c(11)(0)<='0';

FA_line12:
        FA port map(line13(i-1), s(11)(i), c(11)(i), s(12)(i-1), c(12)(i));
s(12)(15)<=line13(15);
c(12)(0)<='0';

FA_line13:
        FA port map(line14(i-1), s(12)(i), c(12)(i), s(13)(i-1), c(13)(i));
s(13)(15)<=line14(15);
c(13)(0)<='0';

FA_line14:
        FA port map(line15(i-1), s(13)(i), c(13)(i), s(14)(i-1), c(14)(i));
s(14)(15)<=line15(15);
```

```
                c(14)(0)<='0';

                FA_line15:
                        FA port map(c(15)(i-1), s(14)(i), c(14)(i), s(15)(i-1), c(15)(i));
                c(15)(0)<='0';
                s(15)(15)<=c(15)(15);

        end generate;

        p(0)<=line0(0);
        p(1)<=s(0)(0);
        p(2)<=s(1)(0);
        p(3)<=s(2)(0);
        p(4)<=s(3)(0);
        p(5)<=s(4)(0);
        p(6)<=s(5)(0);
        p(7)<=s(6)(0);
        p(8)<=s(7)(0);
        p(9)<=s(8)(0);
        p(10)<=s(9)(0);
        p(11)<=s(10)(0);
        p(12)<=s(11)(0);
        p(13)<=s(12)(0);
        p(14)<=s(13)(0);
        p(15)<=s(14)(0);

        p_generation:
                for i in 0 to 15 generate
                        p(16+i)<=s(15)(i);
                end generate;
end Struct;

entity FA is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          cin : in  STD_LOGIC;
          s : out  STD_LOGIC;
          cout : out  STD_LOGIC);
end FA;

architecture Behavioral of FA is
begin
         s<=a xor b xor cin;
         cout<=(a and b) or (a and cin) or (b and cin);
end Behavioral;

entity HA is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          s : out  STD_LOGIC;
          cout:out STD_LOGIC);
end HA;

architecture Behavioral of HA is
begin
         s<=a xor b;
         cout<=a and b;
end Behavioral;
```

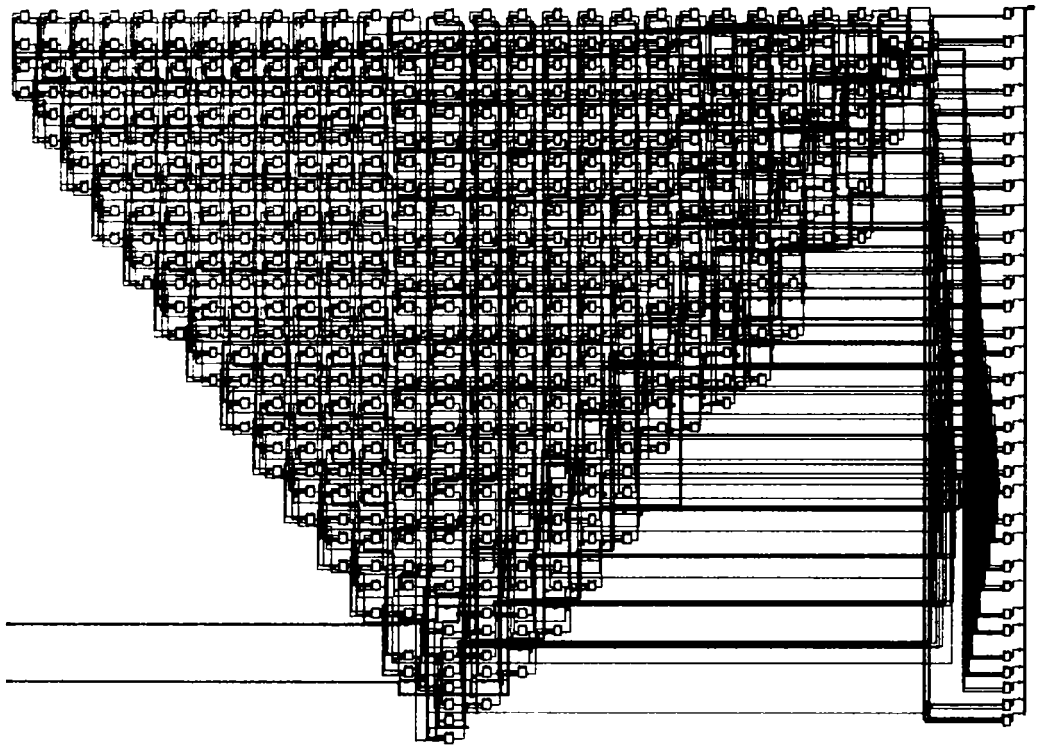


Figure A4 – Technology Schematic of C6288 Benchmark Circuit Obtained with XST

## A5 74181 Benchmark Circuit

```
entity C74181 is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          s : in  STD_LOGIC_VECTOR (3 downto 0);
          cin : in  STD_LOGIC;
          m : in  STD_LOGIC;
          eq : out  STD_LOGIC;
          f : out  STD_LOGIC_VECTOR (3 downto 0);
          x : out  STD_LOGIC;
          y : out  STD_LOGIC;
          cout : out  STD_LOGIC);
end C74181;
architecture Struct of C74181 is
component m1
           Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          s : in  STD_LOGIC_VECTOR (1 downto 0);
          e : out  STD_LOGIC_VECTOR (3 downto 0));
end component;
component m2
           Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          s : in  STD_LOGIC_VECTOR (1 downto 0);
          d : out  STD_LOGIC_VECTOR (3 downto 0));
end component;
component CLA
```

```
        Port ( d : in  STD_LOGIC_VECTOR (3 downto 0);
       e : in  STD_LOGIC_VECTOR (3 downto 0);
       cin : in  STD_LOGIC;
       c : out  STD_LOGIC_VECTOR (3 downto 0);
       cout : out  STD_LOGIC;
       x : out  STD_LOGIC;
       y : out  STD_LOGIC);
end component;
signal d,e,carry, f_buf:std_logic_vector(3 downto 0);
signal s1, s2:std_logic_vector(1 downto 0);
begin
        s1(0)<=s(2);
        s1(1)<=s(3);
        s2(0)<=s(0);
        s2(1)<=s(1);
        M1_logic: m1 port map(a,b,s1,e);
        M2_logic: m2 port map(a,b,s2,d);
        CLA_logic: CLA port map(d,e,cin, carry, cout, x, y);
        f_generation:
        for i in 0 to 3 generate
                f_buf(i)<=( m or carry(i)) xor e(i) xor d(i);
                f(i)<=f_buf(i);
        end generate;
        eq<=f_buf(0) and f_buf(1) and f_buf(2) and f_buf(3);
end Struct;

entity CLA is
   Port ( d : in  STD_LOGIC_VECTOR (3 downto 0);
       e : in  STD_LOGIC_VECTOR (3 downto 0);
       cin : in  STD_LOGIC;
       c : out  STD_LOGIC_VECTOR (3 downto 0);
       cout : out  STD_LOGIC;
       x : out  STD_LOGIC;
       y : out  STD_LOGIC);
end CLA;
architecture Behavioral of CLA is
signal c0g0,c0g01,c0g012,c0g0123:std_logic;
signal p0g1,p0g12,p0g123,p1g2,p1g23,p2g3:std_logic;
signal y_buf:std_logic;
begin
        p_g_products:
                c0g0<=cin and e(0);
                c0g01<=cin and e(0) and e(1);
                c0g012<=cin and e(0) and e(1) and e(2);
                c0g0123<=cin and e(0) and e(1) and e(3);
                p0g1<=d(0) and e(1);
                p0g12<=d(0) and e(1) and e(2);
                p0g123<=d(0) and e(1) and e(2) and e(3);
                p1g2<=d(1) and e(2);
                p1g23<=d(1) and e(2) and e(3);
                p2g3<=d(2) and e(3);
        x<=(e(0) nand e(1)) nand (e(2) nand e(3));
        carry:
                c(0)<=not cin;
                c(1)<=d(0) nor c0g0;
                c(2)<=not (d(1) or p0g1 or c0g01);
                c(3)<=not (d(2) or p1g2 or p0g12 or c0g012);
        y_buf<=not (d(3) or p2g3 or p1g23 or p0g123);
```

```
        cout<=y_buf nand (not c0g0123);
        y<=y_buf;
end Behavioral;

entity m1 is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          s : in  STD_LOGIC_VECTOR (1 downto 0);
          e : out  STD_LOGIC_VECTOR (3 downto 0));
end m1;
architecture Behavioral of m1 is
begin
        result_generation:
        for i in 0 to 3 generate
                e(i)<=(a(i) and b(i) and s(1)) nor (a(i) and (not b(i)) and s(0));
        end generate;
end Behavioral;

entity m2 is
   Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
          b : in  STD_LOGIC_VECTOR (3 downto 0);
          s : in  STD_LOGIC_VECTOR (1 downto 0);
          d : out  STD_LOGIC_VECTOR (3 downto 0));
end m2;
architecture Behavioral of m2 is
begin
        d_generation:
        for i in 0 to 3 generate
                d(i)<=((not b(i) and s(1)) nor (b(i) and s(0))) nor a(i);
        end generate;
end Behavioral;
```

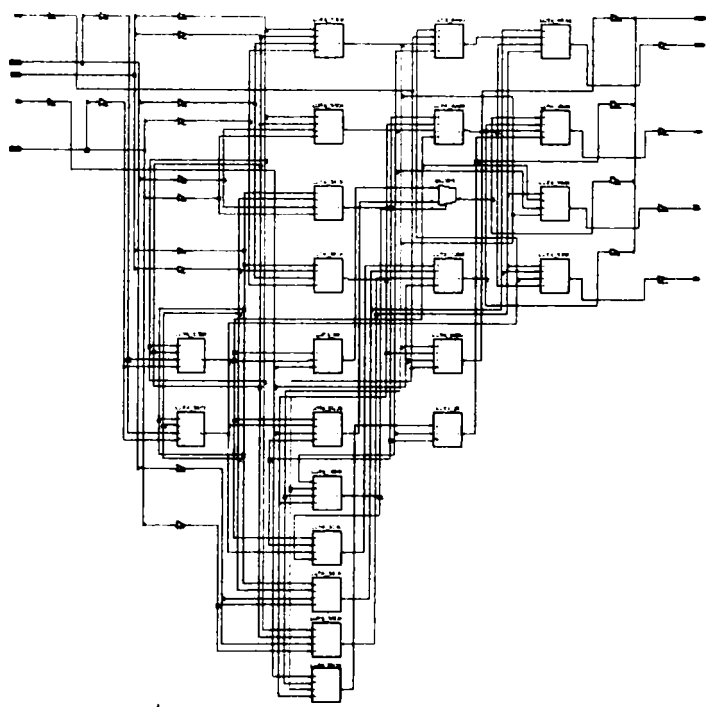


Figure A5 – Technology Schematic of 74181 Benchmark Circuits Obtained with XST

# Appendix B
# VHDL Descriptions for Basic Modules Used in the IEEE Half Precision FP Synthesizable Designs

## B1. Compound Adder

```
entity CompoundAdder12 is
   Port ( a : in  STD_LOGIC_VECTOR (11 downto 0);
          b : in  STD_LOGIC_VECTOR (11 downto 0);
          s : out  STD_LOGIC_VECTOR (11 downto 0);
          s1 : out  STD_LOGIC_VECTOR (11 downto 0));
end CompoundAdder12;

architecture Behavioral of CompoundAdder12 is

component CLA_cell
          Port ( g_ij : in  STD_LOGIC;
          p_ij : in  STD_LOGIC;
          g_jk : in  STD_LOGIC;
          p_jk : in  STD_LOGIC;
          g_ik : out  STD_LOGIC;
          p_ik : out  STD_LOGIC);
end component;

signal p_level1, g_level1:std_logic_vector(11 downto 0);
signal p_level2, g_level2:std_logic_vector(11 downto 0);
signal p_level3, g_level3:std_logic_vector(11 downto 0);
signal p_level4, g_level4:std_logic_vector(11 downto 0);
signal p_level5, g_level5:std_logic_vector(11 downto 0);
signal c0,c1:std_logic_vector(11 downto 0);

begin
          level1:
          for i in 0 to 11 generate
                    p_level1(i)<=a(i) xor b(i);
                    g_level1(i)<=a(i) and b(i);
                    s(i)<=p_level1(i) xor c0(i);
                    s1(i)<=p_level1(i) xor c1(i);
          end generate;

          level2_p:
          for i in 0 to 5 generate
                    CLA_block2_0: CLA_cell port map(g_level1(2*i), p_level1(2*i),
                                                    g_level1(2*i+1), p_level1(2*i+1),
                                                    g_level2(2*i+1), p_level2(2*i+1));
                    g_level2(2*i)<=g_level(2*i);
                    p_level2(2*i)<=g_level1(2*i);
          end generate;

          level3_p:
```

```
        for i in 0 to 2 generate
                CLA_block3_0:CLA_cell port map(g_level2(4*i+1), p_level2(4*i+1),
                                               g_level2(4*i+3), p_level2(4*i+3),
                                               g_level3(4*i+3), p_level3(4*i+3));
        CLA_block3_1:CLA_cell port map(g_level2(4*i+1), p_level2(4*i+1),
                                               g_level2(4*i+2), p_level2(4*i+2),
                                               g_level3(4*i+2), p_level3(4*i+2));
                g_level3(4*i)<=g_level2(4*i);
                p_level3(4*i)<=p_level2(4*i);
                g_level3(4*i+1)<=g_level2(4*i+1);
                p_level3(4*i+1)<=p_level2(4*i+1);
        end generate;

        level4_p:
        for i in 0 to 3 generate
                CLA_block4: CLA_cell port map(g_level3(3), p_level3(3),
                                               g_level3(4+i), p_level3(4+i),
                                               g_level4(4+i), p_level4(4+i));
                g_level4(i)<=g_level3(i);
                p_level4(i)<=p_level3(i);
                g_level4(8+i)<=g_level3(8+i);
                p_level4(8+i)<=p_level3(8+i);
        end generate;

        level5_p:
        for i in 0 to 3 generate
                CLA_block5: CLA_cell port map(g_level4(7), p_level4(7),
                                               g_level4(8+i), p_level4(8+i),
                                               g_level5(8+i), p_level5(8+i));
                g_level5(i)<=g_level4(i);
                p_level5(i)<=p_level4(i);
                g_level5(4+i)<=g_level4(4+i);
                p_level5(4+i)<=p_level4(4+i);
        end generate;

        carry_formation:
        for i in 0 to 10 generate
                c0(i+1)<=g_level5(i);
                c1(i+1)<=g_level5(i) or p_level5(i);
        end generate;
        c0(0)<='0';
        c1(0)<='1';
end Behavioral;

entity CLA_cell is
   Port ( g_ij : in  STD_LOGIC;
          p_ij : in  STD_LOGIC;
          g_jk : in  STD_LOGIC;
          p_jk : in  STD_LOGIC;
          g_ik : out  STD_LOGIC;
          p_ik : out  STD_LOGIC);
end CLA_cell;

architecture Behavioral of CLA_cell is
begin
        p_ik<=p_ij and p_jk;
        g_ik<=g_jk or (p_jk and g_ij);
end Behavioral;
```

## B2. Leading Zero Predictor

```
entity LZP is
   Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
          b : in  STD_LOGIC_VECTOR (10 downto 0);
          sub : in  STD_LOGIC;
          norm : out  STD_LOGIC_VECTOR (3 downto 0));
end LZP;

architecture Behavioral of LZP is
component encoding_cell
          Port ( a : in  STD_LOGIC;
                 b : in  STD_LOGIC;
                 s_sup: in std_logic;
                 g_sup: in std_logic;
                 e_inf: in std_logic;
                 s : out  STD_LOGIC;
                g : out  STD_LOGIC;
                e : out  STD_LOGIC;
                 f : out std_logic);
end component;
component lzd
          Port ( f : in  STD_LOGIC_VECTOR (10 downto 0);
          z : out  STD_LOGIC_VECTOR (3 downto 0));
end component;

signal f:std_logic_vector(10 downto 0);
signal g,s,e:std_logic_vector (11 downto 0);
signal z:std_logic_vector(3 downto 0);
begin
          g(0)<='0';
          s(0)<='0';
          e(11)<='1';
          encoding_modules:
          for i in 0 to 10 generate
                                    enc_cell: encoding_cell port map(a(i),b(i), s(i),g(i),e(i+1),
                                    s(i+1),g(i+1),e(i),f(i));
          end generate;
          leading_zero_detection:
                    lzd port map(f, z);
          result:
          for i in 0 to 3 generate
                    norm(i)<=sub and z(i);
          end generate;
end Behavioral;

entity LZD is
   Port ( f : in  STD_LOGIC_VECTOR (10 downto 0);
          z : out  STD_LOGIC_VECTOR (3 downto 0));
end LZD;

architecture Behavioral of LZD is
signal v0:std_logic_vector(2 downto 0);
signal p0_0,p0_1,p0_2:std_logic_vector(1 downto 0);
signal v1:std_logic;
signal p1_0:std_logic_vector(2 downto 0);
signal p2_0:std_logic_vector(3 downto 0);
```

```
begin
   first_level_4_bits:
        v0(2)<=f(10) or f(9) or f(8) or f(7);
        p0_2(1)<=f(10) nor f(9);
        p0_2(0)<=(f(10) nor (not f(9))) or (f(10) nor f(8));
        v0(1)<=f(6) or f(5) or f(4) or f(3);
        p0_1(1)<=f(6) nor f(5);
        p0_1(0)<=(f(6) nor (not f(5))) or (f(6) nor f(4));
        v0(0)<=f(2) or f(1) or f(0);
        p0_0(1)<=f(2) nor f(1);
        p0_0(0)<=(f(2) nor (not f(1))) or (f(2) nor f(0));

    second_level_8_bits:
        v1<=v0(2) or v0(1);
        p1_0(2)<=not(v0(2));
        p1_0(1)<=       p0_2(1) when v0(2)='1' else
                        p0_1(1) when v0(2)='0' else
                        '0';
        p1_0(0)<=       p0_2(0) when v0(2)='1' else
                        p0_1(0) when v0(2)='0' else
                        '0';

    third_level_11_bits:
        p2_0(3)<=not v1;
        p2_0(2)<=       p1_0(2) when v1='1' else
                        (not v0(0)) when v1='0' else
                        '0';
        p2_0(1)<=       p1_0(1) when v1='1' else
                        (v0(0) and p0_0(1)) when v1='0' else
                        '0';
        p2_0(0)<=       p1_0(0) when v1='1' else
                        (v0(0) and p0_0(0)) when v1='0' else      '0';
        Z<=p2_0;
end Behavioral;
```

## B3. Booth Radix-4 Encoding Module

```
entity Booth_encoder is
   Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
        b : in  STD_LOGIC_VECTOR (10 downto 0);
        pp0 : out  STD_LOGIC_VECTOR (14 downto 0);
        pp1 : out  STD_LOGIC_VECTOR (15 downto 0);
        pp2 : out  STD_LOGIC_VECTOR (15 downto 0);
        pp3 : out  STD_LOGIC_VECTOR (15 downto 0);
        pp4 : out  STD_LOGIC_VECTOR (14 downto 0);
        pp5 : out  STD_LOGIC_VECTOR (12 downto 0));
end Booth_encoder;

architecture Behavioral of Booth_encoder is
component booth_encoding_cell
         Port ( x_i0 : in  STD_LOGIC;
        x_i1 : in  STD_LOGIC;
        x_i2 : in  STD_LOGIC;
        sel1 : out  STD_LOGIC;
        sel2 : out  STD_LOGIC;
        s : out  STD_LOGIC);
```

```
end component;
component booth_line
        Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
       sel1 : in  STD_LOGIC;
       sel2 : in  STD_LOGIC;
       a_booth : out  STD_LOGIC_VECTOR (11 downto 0));
end component;

signal zero, one:std_logic;
signal line0, line1, line2, line3, line4, line5:std_logic_vector(11 downto 0);
signal sel1_0, sel1_1, sel1_2, sel1_3, sel1_4, sel1_5:std_logic;
signal sel2_0, sel2_1, sel2_2, sel2_3, sel2_4, sel2_5:std_logic;
signal s0, s1, s2, s3, s4, s5:std_logic;

begin
        zero<='0';
        one<='1';

        booth_encoding_cell_0:
                booth_encoding_cell port map(zero, a(0), a(1), sel1_0, sel2_0, s0);
        booth_line_0:
                booth_line port map(b, sel1_0, sel2_0, line0);
        partial_product_0:
                for i in 0 to 11 generate
                        pp0(i)<=line0(i);
                end generate;
        pp0(12)<=s0;
        pp0(13)<=s0;
        pp0(14)<=not s0;

        booth_encoding_cell_1:
                booth_encoding_cell port map(a(1), a(2), a(3), sel1_1, sel2_1, s1);
        booth_line_1:
                booth_line port map(b, sel1_1, sel2_1, line1);
        partial_product_1:
                for i in 0 to 11 generate
                        pp1(i+2)<=line1(i) xor s1;
                end generate;
        pp1(14)<=not s1;
        pp1(15)<=one;
        pp1(0)<=s0;
        pp1(1)<=zero;

        booth_encoding_cell_2:
                booth_encoding_cell port map(a(3), a(4), a(5), sel1_2, sel2_2, s2);
        booth_line_2:
                booth_line port map(b, sel1_2, sel2_2, line2);
        partial_product_2:
                for i in 0 to 11 generate
                        pp2(i+2)<=line2(i) xor s2;
                end generate;
        pp2(14)<=not s2;
        pp2(15)<=one;
        pp2(0)<=s1;
        pp2(1)<=zero;

        booth_encoding_cell_3:
                booth_encoding_cell port map(a(5), a(6), a(7), sel1_3, sel2_3, s3);
```

```
        booth_line_3:
                booth_line port map(b, sel1_3, sel2_3, line3);

        partial_product_3:
                for i in 0 to 11 generate
                        pp3(i+2)<=line3(i) xor s3;
                end generate;
        pp3(14)<=not s3;
        pp3(15)<=one;
        pp3(0)<=s2;
        pp3(1)<=zero;

        booth_encoding_cell_4:
                booth_encoding_cell port map(a(7), a(8), a(9), sel1_4, sel2_4, s4);
        booth_line_4:
                booth_line port map(b, sel1_4, sel2_4, line4);
        partial_product_4:
                for i in 0 to 11 generate
                        pp4(i+2)<=line4(i) xor s4;
                end generate;
        pp4(14)<= not s4;
        pp4(0)<=s3;
        pp4(1)<=zero;

        booth_encoding_cell_5:
              booth_encoding_cell port map(a(9), a(10), zero, sel1_5, sel2_5, s5);
        booth_line_5:
                booth_line port map(b, sel1_5, sel2_5, line5);
        partial_product_5:
                for i in 0 to 10 generate
                        pp5(i+2)<=line5(i) xor s5;
                end generate;
        pp5(0)<=s4;
        pp5(1)<=zero;
end Behavioral;

entity Booth_encoding_cell is
   Port ( x_i0 : in  STD_LOGIC;
          x_i1 : in  STD_LOGIC;
          x_i2 : in  STD_LOGIC;
          sel1 : out  STD_LOGIC;
          sel2 : out  STD_LOGIC;
          s : out  STD_LOGIC);
end Booth_encoding_cell;

architecture Behavioral of Booth_encoding_cell is
begin
         sel1<=x_i0 xor x_i1;
         sel2<=(x_i0 xor x_i1) nor (x_i1 xnor x_i2);
         s<=x_i2;
end Behavioral;

entity booth_line is
   Port ( a : in  STD_LOGIC_VECTOR (10 downto 0);
          sel1 : in  STD_LOGIC;
          sel2 : in  STD_LOGIC;
          a_booth : out  STD_LOGIC_VECTOR (11 downto 0));
end booth_line;
```

```
architecture Behavioral of booth_line is
signal a_prep, a2_prep:std_logic_vector(11 downto 0);
begin
        multiplicand_prep:
                a_prep(10 downto 0)<=a(i);
                a2_prep(11 downto 1)<=a(i);
                a_prep(11)<='0';
                a2_prep(0)<='0';
        line_encoding:
                for i in 0 to 11 generate
                        a_booth(i)<=(a_prep(i) and sel1) or (a2_prep(i) and sel2);
                end generate;
end Behavioral;
```

## B4. Wallace Tree

```
entity wallace_tree is
   Port ( pp0 : in  STD_LOGIC_VECTOR (14 downto 0);
        pp1 : in  STD_LOGIC_VECTOR (15 downto 0);
        pp2 : in  STD_LOGIC_VECTOR (15 downto 0);
        pp3 : in  STD_LOGIC_VECTOR (15 downto 0);
        pp4 : in  STD_LOGIC_VECTOR (14 downto 0);
        pp5 : in  STD_LOGIC_VECTOR (12 downto 0);
        sum : out  STD_LOGIC_VECTOR (21 downto 0);
        carry : out  STD_LOGIC_VECTOR (21 downto 0));
end wallace_tree;

architecture Behavioral of wallace_tree is
component fa_cell
            Port ( a : in  STD_LOGIC;
        b : in  STD_LOGIC;
        cin : in  STD_LOGIC;
        s : out  STD_LOGIC;
        c : out  STD_LOGIC);
end component;

signal pp0_0, pp1_0, pp2_0:std_logic_vector(17 downto 0);
signal pp3_0, pp4_0, pp5_0:std_logic_vector(16 downto 0);
signal s1, c1:std_logic_vector(17 downto 0);
signal s2, c2:std_logic_vector(16 downto 0);
signal pp0_1, pp1_1, pp2_1:std_logic_vector(20 downto 0);
signal s3, c3:std_logic_vector(20 downto 0);
signal pp0_2, pp1_2, pp2_2:std_logic_vector(21 downto 0);
signal s4,c4:std_logic_vector(21 downto 0);
begin
          pp0_0(14 downto 0)<=pp0;
          pp0_0(15)<='0';
          pp0_0(16)<='0';
          pp0_0(17)<='0';
          pp1_0(15 downto 0)<=pp1;
          pp1_0(16)<='0';
          pp1_0(17)<='0';
          pp2_0(17 downto 2)<=pp2;
          pp2_0(0)<='0';
          pp2_0(1)<='0';
          fa_line0_1:
          for i in 0 to 17 generate
```

```
            full_adder0:fa_cell port map (pp2_0(i),pp1_0(i),pp0_0(i),s1(i),c1(i));
        end generate;

        pp3_0(15 downto 0)<=pp3;
        pp3_0(16)<='0';
        pp4_0(16 downto 2)<=pp4;
        pp4_0(0)<='0';
        pp4_0(1)<='0';
        pp5_0(16 downto 4)<=pp5;
        pp5_0(0)<='0';
        pp5_0(1)<='0';
        pp5_0(2)<='0';
        pp5_0(3)<='0';
        fa_line1_1:
        for i in 0 to 16 generate
            full_adder1:fa_cell port map(pp5_0(i),pp4_0(i),pp3_0(i),s2(i),c2(i));
        end generate;

        pp0_1(17 downto 0)<=s1;
        pp0_1(18)<='0';
        pp0_1(19)<='0';
        pp0_1(20)<='0';
        pp1_1(18 downto 1)<=c1;
        pp1_1(0)<='0';
        pp1_1(19)<='0';
        pp1_1(20)<='0';
        pp2_1(20 downto 4)<=s2;
        pp2_1(0)<='0';
        pp2_1(1)<='0';
        pp2_1(2)<='0';
        pp2_1(3)<='0';
        fa_adder0_2:
        for i in 0 to 20 generate
            full_adder2:fa_cell port map(pp2_1(i),pp1_1(i),pp0_1(i),s3(i),c3(i));
        end generate;

        pp0_2(20 downto 0)<=s3;
        pp0_2(21)<='0';
        pp1_2(21 downto 1)<=c3;
        pp1_2(0)<='0';
        pp2_2(21 downto 5)<=c2;
        pp2_2(0)<='0';
        pp2_2(1)<='0';
        pp2_2(2)<='0';
        pp2_2(3)<='0';
        pp2_2(4)<='0';
        fa_adder0_3:
        for i in 0 to 21 generate
          full_adder3:fa_cell port map(pp2_2(i),pp1_2(i),pp0_2(i),s4(i),c4(i));
        end generate;

        sum<=s4;
        carry(21 downto 1)<=c4(20 downto 0);
        carry(0)<='0';
end Behavioral;
```

## B5. SRT Radix 2 Stage

```
entity SRT_stage is
   Port ( pr_s : in  STD_LOGIC_VECTOR (14 downto 0);
          pr_c : in  STD_LOGIC_VECTOR (15 downto 0);
          d : in  STD_LOGIC_VECTOR (13 downto 0);
          pr_s_next : out  STD_LOGIC_VECTOR (14 downto 0);
          pr_c_next : out  STD_LOGIC_VECTOR (15 downto 0);
          q : out  STD_LOGIC_VECTOR (1 downto 0));
end SRT_stage;

architecture Behavioral of SRT_stage is
component selection_logic
          Port ( pr_s : in  STD_LOGIC_VECTOR (3 downto 0);
          pr_c : in  STD_LOGIC_VECTOR (3 downto 0);
          q : out  STD_LOGIC_VECTOR (1 downto 0));
end component;

signal d_neg, d_pos, d_zero, d_add:std_logic_vector(14 downto 0);
signal q_par:std_logic_vector(1 downto 0);
signal cin:std_logic;
signal pr_s_sel, pr_c_sel:std_logic_vector(3 downto 0);
signal pr_c_add, pr_s_res:std_logic_vector(14 downto 0);
signal pr_c_res:std_logic_vector(15 downto 0);
begin
          d_pos(13 downto 0)<=d;
          d_pos(14)<='0';
          d_zero<="000000000000000";
          negate:
                    for i in 0 to 14 generate
                              d_neg(i)<=not d_pos(i);
                    end generate;
          multiplexing:
                    d_add<=d_neg when q_par="01" else
                              d_pos when q_par="10" else
                              d_zero;
                    cin<='1' when q_par="01" else '0';

          pr_s_sel<=pr_s(13 downto 10);
          pr_c_sel<=pr_c(13 downto 10);
          quotient_sel:
                    selection_logic port map(pr_s_sel, pr_c_sel, q_par);
          pr_c_add(14 downto 1)<=pr_c(14 downto 1);
          pr_c_add(0)<=cin;
          carry_save_addition:
          for i in 0 to 14 generate
                    pr_s_res(i)<=pr_c_add(i) xor pr_s(i) xor d_add(i);
                    pr_c_res(i+1)<=(pr_c_add(i) and pr_s(i)) or (pr_c_add(i) and d_add(i)) or
                                                                        (pr_s(i) and d_add(i));
          end generate;
          pr_s_next<=pr_s_res;
          pr_c_next(15 downto 1)<=pr_c_res(15 downto 1);
          pr_c_next(0)<='0';
          q<=q_par;
end Behavioral;


entity selection_logic is
   Port ( pr_s : in  STD_LOGIC_VECTOR (3 downto 0);
          pr_c : in  STD_LOGIC_VECTOR (3 downto 0);
```

```
        q : out  STD_LOGIC_VECTOR (1 downto 0));
end selection_logic;

architecture Behavioral of selection_logic is

signal sum:std_logic_vector(3 downto 0);
signal all_0, all_1, zero:std_logic;

begin
        sum<=pr_s + pr_c;

        all_0<= not(sum(0) or sum(1) or sum(2) or sum(3));
        all_1<=sum(0) and sum(1) and sum(2) and sum(3);
        zero<=not(pr_s(3) or pr_c(3) or pr_s(2) or pr_c(2) or
                        pr_s(1) or pr_c(1) or pr_s(0) or pr_c(0));

        q<="00" when ((all_1 or (zero and all_0))='1') else
             "01" when ((not sum(3) and not (zero and all_0))='1') else
              "10" when ((sum(3) and (not all_1))='1') else "11";
end Behavioral;

entity quotient_formation_block is
        generic(n:integer:=8);
        port(     q_digit:in std_logic_vector(1 downto 0);
                        q_n:in std_logic_vector(n-1 downto 0);
                        qm_n:in std_logic_vector(n-1 downto 0);
                        q_n1:out std_logic_vector(n downto 0);
                        qm_n1:out std_logic_vector (n downto 0));
end quotient_formation_block;

architecture Behavioral of quotient_formation_block is
begin
        q_n1(n downto 1)<=qm_n when (q_digit="10") else
                                q_n;
        q_n1(0)<='1' when (q_digit="01" or q_digit="10") else
                    '0';
        qm_n1(n downto 1)<=q_n when (q_digit="01") else
                                    qm_n;
        qm_n1(0)<='1' when (q_digit="00") else '0';
end Behavioral;
```

# REFERENCES


[1] A. Akkas **A Combined Interval and Floating-Point Comparator/Selector** Proceedings 13th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'02), 2002, pp 208-217

[2] H. A. AlTwaijry **Area and Performance Optimized CMOS Multipliers** PhD. Thesis, Stanford University, 1997

[3] A. Amaricai, M. Vladutiu, L.Prodan, M. Udrescu, O. Boncalo **Design of Addition and Multiplication Units for High Performance Interval Arithmetic Processor** Proceedings 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2007, pp 223-226

[4] A. Amaricai, M. Vladutiu, L.Prodan, M. Udrescu, O. Boncalo **Hardware Support for Combined Interval and Floating Point Multiplication** Proceedings 14th Mixed Design Of Integrated Circuits and Systems, pp 278-282

[5] A. Amaricai, M. Vladutiu, L.Prodan, M. Udrescu, O. Boncalo **Exploiting Parallelism in Double Path Adders' Structure for Increased Throughput of Floating Point Addition** Proceedings 10th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools, 2007, pp 132-137

[6] A. Amaricai, M. Vladutiu, L.Prodan, M. Udrescu, O. Boncalo **Floating point multiplication rounding schemes for interval arithmetic** Proceedings 19th IEEE Application-Specific System, Architectures and Processors (ASAP-2008), 2008, pp 19-24

[7] A. Amaricai, M. Vladutiu, L.Prodan, M. Udrescu, O. Boncalo **Floating Point Divide-Add Fused for Newton's Interval Method** Proceedings of the Euromicro Work In Progress Session held in Conjunction with Euromicro SEAA 2008 and Euromicro DSD 2008, September 3-5, Parma, Italy, ISBN 978-3-902457-20-3

[8] E. Antelo, T. Lang, P. Montuschi, A. Nannarelli, **Fast Radix-4 Division with Selection by Comparisons**, Proc. 13th IEEE International Conference on Application Specific Systems, Architectures and Processors (ASAP'02), 2002, pp 185-190

[9] E. Antelo, T. Lang, P. Montuschi, A. Nannarelli, **Digit-Recurrence Dividers with Reduced Logical Depth** IEEE Trans. on Computers, Vol.54, No. 7, 2005, pp. 837-852

[10] ARM **VFP11 Vector Floating Point Technical Reference Manual**, 2002

[11] D.E. Atkins **Higher Radix Division Using Estimates of the Division and Partial Remainders** IEEE Trans on Computer, Vol. 11, no. 10, 1968

[12] A. Beaumont-Smith, N. Burgess, S. Lefrere, C.C. Lim **Reduced Latency IEEE Floating Point Standard Adder Architectures** Proceedings 14th IEEE Symposium on Computer Arithmetic (ARITH-14), 1999, pp 35-42

[13] G.W. Bewick **Fast Multiplication: Algorithms and Implementation** PhD. Thesis, Stanford University, 1994

[14] P. Bonato, V. G. Oklobdzija **Evaluation of Booth's Algorithm for Implementation in Parallel Multipliers** Proceedings 29th Asilomar Conference on Signals, Systems and Computers, 1995, pp 608-610

[15] R. P. Brent, H. T. Kung **A Regular Layout for Parallel Adders** IEEE Transaction on Computers, Vol. 31, No. 3, 1982, pp 260-264

[16] J. D. Bruguera, T. Lang **Leading One Prediction with Concurrent Position Prediction** IEEE Transaction on Computers, Vol.48, No. 10,1998, pp 1083-1097

[17] J. D. Bruguera, T. Lang **Rounding in Floating Point Addition Using a Compound Adder** Internal Report, University Santiago de Compostela, 2000

[18] J. Bruguera, T. Lang, **Floating-Point Fused Multiply-Add: Reduced Latency for Floating Point Addition** , Proc. 17th IEEE Symposium on Computer Arithmetic (ARITH-17), 2005, pp. 42-51

[19] N. Burgess, T. Williams, **Choices of Operand Truncation in the SRT Division Algorithm**, IEEE Trans. on Computers, Vol. 44, No.7, 1995, pp. 933-938

[20] N. Burgess, C. Hinds **Design Issues in Radix-4 Square Root and Divide Unit** Proc. 35th Asilomar Conference, 2001, pp 1646-1650

[21] N. Burgess, C. Hinds, **Design of the ARM VFP11 Divide and Square Root Synthesizable Macrocell**, Proc. 18th IEEE Symposium on Computer Arithmetic (ARITH-18), 2007

[22] C. Chen, L.A. Chen, J.R. Chen **Architectural Design of a Fast Multiplication-Add Fused Unit Using Signed-Digit Addition**, Proc. Euromicro Symposium on Digital System Design (DSD'01), 2001,

[23] L. Dadda **Some Schemes for Parallel Multipliers** Alta Frequenza, No. 34, 1965, pp. 349-356

[24] D. Daney, Y. Papegay, A. Neumaier, **Interval Methods for Certification of the Kinematic Calibration of Parallel Robots** Proc. 2004 IEEE International Conference on Robots and Automation, 2004, pp. 1913-1918

[25] M. Daumas, G. Melquiond, C. Munoz **Guaranteed Profs Using Interval Arithmetic**, Proceedings 17th IEEE Symposium on Computer Arithmetic (ARITH-17), 2005, pp 188-195

[26] M. Ercegovac, T. Lang **Digital Arithmetic** Morgan Kaufmann Publishers, ISBN 978-1-55860-798-9, 2003

[27] M. Ercegovac, T. Lang, **On-the-Fly Conversion of Redundant into Conventional Representation**, IEEE Trans. on Computers, Vol. 36, No. 7, 1987, pp. 895-897

[28] M. Ercegovac, T. Lang, **On-the-Fly Rounding**, IEEE Trans. On Computers, Vol. 41, No. 12, 1992, pp. 1497-1503

[29] M. Ercegovac, T. Lang, P. Montuschi **Very High Radix Division with Prescaling and Selection by Rounding**, IEEE Trans. On Computers, Vol. 43, No. 8, 1994 pp. 909-918

[30] G. Even, P.M. Seidel **A Comparison of Three Rounding Algorithm for IEEE Floating-Point Multiplication** IEEE Transactions on Computers, Vol. 49, No. 7, 2000, pp 638-650

[31] P.M. Farmwald **On the Design of High Performance Digital Arithmetic Circuits** PhD. Thesis, Stanford University, 1981

[32] Z. Gallias **Proving The Existence Of Periodic Solutions Using Global Interval Newton Method** Proc. IEEE International Conference on Circuits and Systems (ISCAS'99), 1999, pp. 294-297

[33] C.J. Gau, M.A. Stadtherr **Parallel Interval Newton Using Message Passing: Dynamic Load Balancing Strategies** Proc. ACM/IEEE SC 2001 Conference, 2001, pp 23-46

[34] M. Gavriliu **Towards More Efficient Interval Analysis: Corner Forms and a Remainder Newton Method**, PhD. Thesis, Caltech, 2005

[35] G. Gerwig, M. Kroener **Floating Point Unit in Standard Cell Design with 116 Bit Wide Dataflow** Proceedings 14th IEEE Symposium on Computer Arithmetic (ARITH-14), 1999, pp 266-273

[36] D. Goldberg **What Every Computer Scientist Should Know about Floating Point Arithmetic** ACM Computing Surveys, Vol. 23, No.1, 1991, pp 5-48

[37] V.Y. Gorshtein, A.I. Grushin, S.R. Shevstov **Floating Point Addition Methods and Apparatus** US Patent 5808926, Sun Microsystems, 1998

[38] T.D. Han, W.C. Park **Apparatus and Method for Performing Rounding and Addition in Parallel in Floating Point Multiplier** US Patent 6269385, Hyundai Electronics, 2001

[39] M. Hansen, H. Yalcin, J.P. Hayes **Unveiling he ISCAS-85 Benchmarks: A Case Study in Reverse Engineering** IEEE Design and Test, vol. 16, no. 3, pp. 72-80, 1999

[40] D.L. Harris. S.F. Oberman, M.A. Horowitz **SRT Division and Architectures**, Proc. 13th IEEE Symposium on Computer Arithmetic (ARITH-13), 1997, pp. 18-25

[41] B. Hayes **A Lucid Interval** American Scientist, Vol. 91, No.6, 2003, pp.484-488

[42] J. L. Hennessy, D. A. Patterson **Computer Architecture, Fourth Edition: A Quantitative Approach**, Morgan Kaufmann Publishers, ISBN 978-0123704900, 2006

[43] E. Hokenek, R.K. Montoye **Leading Zero Anticipator in the IBM RISC System/6000 Floating Point Execution Unit** IBM Journal of Research and Development, vol. 34, No.1, 1990, 71-77

[44] C. Huang, X. Wu, J. Lai, C. Sun, G. Li **A Design of High Speed Double Precision Floating Point Adder Using Macro Modules** , Proceedings 2005 Asia-Pacific Design Automation Conference (ASP-DAC), 2005, pp D11-D12

[45] Institue for Electric and Electronical Engineers (IEEE), **IEEE Standard VHDL Reference Manual**, ANSI/IEEE Std 1076/1987

[46] R. Jessani, M. Putrino, **Comparison of Single and Dual Path Multiply-Add Fused Floating Point Units**, IEEE Trans. on Computers, Vol. 47, No. 9, 1998, pp. 927-937

[47] R.B. Kearfott **Interval Computations: Introduction, Uses and Resources,** Euromath Bulletin, Vol. 2, No. 1, 1996, pp. 95-112

[48] R. B. Kearfott, M. Novoa, **INTBIS, a Portable Interval Newton/Bisection Package**, ACM Trans. On Mathematical Software, Vol. 16, Issue 2, 1990, pp 152-157

[49] R. Kirchner, U. Kulisch **Hardware Support for Interval Arithmetic** Reliable Computing, Vol 12, No. 3, 2007, pp 225-237

[50] S. Knowles **A Family of Adders** Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH-15), 2001, pp 277-285

[51] I. Koren **Computer Arithmetic Algorithms** AK Peters Ltd, ISBN 978-1-56881-160-4, 2001

[52] P. Kornerup **Digit Selection for SRT Division and Square Root**, IEEE Trans. On Computers, Vol. 54, No.3, 2005, pp 294-303

[53] U.W. Kulisch **Advanced Arithmetic for the Digital Computer** Springer-Verlag, ISBN 978-3-21183-870-9, 2002

[54] U.W. Kulisch **Letters to IEEE Computer Arithmetic Standard Revision Group**, 2008

[55] T. Lang, J. Bruguera, **Floating Point Fused Multiply-Add with Reduced Latency**, Proc. 2002 IEEE International Conference on Computer Design (ICCD-2002), 2002, pp 145-150

[56] G. Li, Z. Li **Design of a A Fully Pipelined Single-Precision Multiply-Add-Fused Unit** Proc. 20th International Conference on VLSI Design (VLSID'07), 2007, pp 318-323

[57] Z. Li, G. Li **Design of a Double-Precision Floating Point Multiply-Add Fused with Consideration of Data Dependence**, Proc. IEEE Annual Symposium on VLSI (ISVLSI), 2007, pp 492-497

[58] D. Lozier **The Use of Floating Point and Interval Arithmetic in the Computation of Error Bounds** IEEE Trans. On Computers, Vol. 32, Issue 4, 1983, pp. 414-417

[59] A.A. Liddicoat **High Performance Arithmetic for Division and Elementary Functions** PhD. Thesis, Stanford University, 2002

[60] A.A. Liddicoat **High Performance Floating Point Divide**, Proc. Euromicro Symposium on Digital System Design (DSD'01), 2001, pp 354-360

[61] R.E. Moore **Interval Arithmetic and Automatic Error Analysis**, PhD Thesis, Stanford University, 1962

[62] Z.Y. Mou, F. Jutand **Cellular Multiplier Comprising of a Tree of Overturned Stairs Type, and Method of Design** US Patent 5497342, France Telecom, 1996

[63] A. Naini, A. Dhablania, W. James, D. Das Sarma **1-GHz HAL SPARC64 Dual Floating Point Unit with RAS Features** Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH-15), 2001, pp 173-184

[64] S.F. Oberman **Design Issues in High Performance Floating Point Arithmetic Units** PhD. Thesis, Stanford University, 1996

[65] S.F. Oberman **Floating Point Arithmetic Unit Including an Efficient Close Data Path** US Patent 6094668, Advanced Micro Devices, 2000

[66] S.F. Oberman, H. Al-Twaijri, M.J. Flynn **The SNAP Project: Design of Floating Point Units** Proceedings 13th IEEE Symposium on Computer Arithmetic (ARITH-13), 1997, pp 157-166

[67] S.F. Oberman, N. Quach, M.J. Flynn **The Design and Implementation of High Performance Floating Point Divider**, Technical Report CSL-TR-94-599, Stanford University, 1994

[68] S.F. Oberman, M.J. Flynn **Measuring the Complexity of SRT Tables**, Technical Report CSL-TR-95-679, Stanford University, 1995

[69] S.F. Oberman **Design Issues in High Performance Floating Point Arithmetic Units** PhD. Thesis, Stanford University, 1996

[70] S.F. Oberman, M.J. Flynn **Design Issues in Division and Other Floating Point Operations** IEEE Trans. On Computers, Vol. 46, No.2, 1997, pp 154-161

[71] S.F. Oberman, M.J. Flynn **Division Algorithms and Implementations** IEEE Trans. On Computers, Vol. 46, No. 8, 1997, pp. 833-854

[72] N. Ohkubo, M. Suzuki, T. Shinbo, T. Yamanaka, A. Shimizu, K. Sasaki, Y. Nakagome **A 4.4 ns CMOS 54*54 Multiplier Using Pass Transistor Multiplexer** IEEE Journal of Solid State Circuits, Vol. 30, No. 3, pp 251-257, 1995

[73] V.G. Oklobdzija **An Algorithmic and Novel Design of Leading Zero Detector Circuit: Comparison with Logic Synthesis** IEEE Transactions on Very Large Scale Integration System, Vol.2, No. 1, 1994, pp 124-128

[74] B. Parhami **Computer Arithmetic: Algorithms and Hardware Designs** Oxford University Press, ISBN 978-0-19512-583-2, 1999

[75] W.J. Paul, P.M. Seidel **To Booth or Not to Booth** Integration, the VLSI Journal, Vol. 32, No. 11, 2002, pp 5-40

[76] V. Petcu, A. Amaricai, M. Vladutiu **A Dual-Threaded Architecture for Interval Arithmetic Coprocessor with Shared Floating Point Units** Proc. 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2008,

[77] J. A. Prabhu, G. B. Zyner **167 MHz Radix-8 Divide and Square Root Using Overlapped Radix-2 Stages** Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH-15), 1995

[78] N.T. Quach, M.J. Flynn **An Improved Algorithm for High Speed Floating Point Addition** Technical Report CSL-TR-90-442, Stanford University, 1990

[79] N.T. Quach, N. Takagi, M.J. Flynn **On Fast IEEE Rounding** Technical Report CSL-TR-91-459, Stanford University, 1991

[80] N.T. Quach, M.J. Flynn **Leading One Prediction – Implementation, Generalization and Application** Technical Report CSL-TR-91-463, Stanford University, 1991

[81] H. Ratschek, J. Rokne **New Computer Methods for Global Optimization**, Ellis Horwood Ltd., 1988

[82] N. Revol, **Interval Newton Iteration in Multiple Precision for the Univariate Case**, Technical Report 4334, INRIA, 2001

[83] J.E. Robertson **A New Class of Digital Division Methods** IRE Trans. On Electronic Computer, Vol. EC-7, 1958, pp. 218-222

[84] J.F. Sanjuan-Estrada, L.G. Casado, I. Garcia, **Reliable Algorithms for Ray Intersection in Computer Graphics Based on Interval Arithmetic**, Proc. 16th Brazilian Symposium on Computer Graphics and Image Processing, 2003

[85] M.J. Schmookler, K.J. Nowka **Leading Zero Detection and Anticipation: A Comparisson of Methods** Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH-15), 2001, pp 7-12

[86] M.J. Schulte **A Variable-Precision, Interval Arithmetic Processor** PhD. Thesis, University of Texas at Austin, 1996

[87] M.J. Schulte, E. Swartzlander **Hardware Design and Arithmetic Algorithms for Variable-Precision Interval Arithmetic Coprocessor** Proceedings 12th Symposium on Computer Arithmetic (ARITH-12), 1995, pp 222-230

[88] P.M. Seidel **On the Design of IEEE Compliant Floating Point Units and Their Quantitative Analysis** PhD. Thesis, University of Saarlanden, 1999

[89] P.M. Seidel, G. Even **On the Design of Fast IEEE Floating Point Adders** Proceedings 15th IEEE Symposium on Computer Arithmetic (ARITH-15), 2001, pp 184-194

[90] P.M. Seidel, G. Even **Delay-Optimized Implementation of IEEE Floating Point Addition** IEEE Transaction on Computers, Vol. 53, No.2 , 2004, pp 97-113

[91] R. Shettar, R.M. Banakar, P.S. Nataraj **Implementation of Interval Arithmetic Algorithms on FPGAs**, Proceeding 2007 International Conference on Computational Intelligence and Multimedia Application, 2007, pp 196-200

[92] P. Soderquist, M. Lesser **Area and Performance Tradeoffs in Floating Point Divide and Square Root Implementations**, ACM Computing Surveys, Vol. 28, No. 3, 1996

[93] G. L. Steele Jr. **Comparator Unit for Comparing Values of Floating Point Operands** US Patent 7191202, 2007

[94] J.E. Stine **Design Issues for Accurate and Reliable Arithmetic** PhD. Thesis, Lehigh University, 2000

[95] J. E. Stine, M.J. Schulte **A Combined Interval and Floating Point Divider**, Proc. 32nd Asilomar Conference on Signals, Systems and Computers, vol.1, 1998, pp. 218-222

[96] J.E. Stine, M.J. Schulte **A Combined Interval and Floating Point Multiplier** Proceedings 8th ACM Great Lakes Symposium on VLSI, 1998, pp 208-215

[97] SUN Microsystems, **Interval Arithmetic in High Performance Computing**, 2002

[98] A. Tyagi **A Reduced Area Scheme for Carry Select Adder** IEEE Transaction on Computer, Vol. 42, No. 10, 1993, pp 1163-1170

[99] S.D. Trong, M. Schmookler, E. M. Schwarz, **P6 Binary Floating Point Unit**, Proc. 18th IEEE Symposium on Computer Arithmetic (ARITH-18), 2007, pp 77-86

[100] M. Vladutiu **Arhitectura Calculatoarelor, Vol. 1: Aritmetica Sistemelor de Calcul**, Editura Politehnica, 2008

[101] C.S. Wallace **A Suggestion for a Fast Multiplier** IEEE Transaction on Electronic Computers, EC-13, Issue 1, 1964, pp 14-17

[102] G.W. Walster, E.R. Hansen **Solving A Nonlinear Equation Through Interval Arithmetic and Term Consistency**, US Patent 6823352, Sun Microsystems, 2004

[103] G.W. Walster, E.R. Hansen **Solving Systems of Nonlinear Equations Using Interval Arithmetic and Term Consistency**, US Patent 6859817, Sun Microsystems, 2005

[104] G.W. Walster, E.R. Hansen **Termination Criteria for the One Dimensional Interval Version of Newton's Method**, US Patent 6914320, Sun Microsystems, 2005

[105] G.W. Walster, E.R. Hansen **Methods and Apparatus for Solving Systems of Nonlinear Equations Using Interval Arithmetic**, US Patent 6915321, Sun Microsystems, 2005

[106] A. Weinberger **4-2 Carry Save Adder Module** IBM Technical Disclosure, Vol. 23, 1981

[107] J. Wolf von Gudenberg **Hardware Support for Interval Arithmetic** Scientific Computing with Automatic Result Verification, Academic Press, ISBN 978-0-12044-210-2, 1993, pp 549-570

[108] Xilinx **Xilinx ISE 10.1 Design Suite Software Manual,** 2008

[109] R.K. Yu, G.B. Zyner **167 MHz Radix 4 Floating Point Multiplier** Proceedings 12th IEEE Symposium on Computer Arithmetic (ARITH-12), 1995, pp 149-154

[110] D. Zuras, W. McAllister **Balanced Delay Trees and Combinatorial Division in VLSI** IEEE Journal of Solid State Circuits, Vol. 21, No. 5, pp 814-819, 1986

[111] ***http://www.boost.org/doc/libs/1_35_0/libs/numeric/interval/doc/interval.htm – the Boost C++ Interval Arithmetic

[112] ***http://www-sop.inria.fr/coprin/aol/form.html – the ALIAS interval library

[113] *** http://www.cgal.org/ -the Computational Geometry Algorithms Library

[114] *** http://perso.ens-lyon.fr/nathalie.revol/mpfi_toc.html - the Multiple Precision Floating Point Interval Library (MPFI)

[115] *** http://www.ti3.tu-harburg.de/rump/intlab/index.html - the Interval extension of MATLAB

[116] *** http://standards.ieee.org/announcements/intervalarith.html - IEEE Standard Association WorkGroup 1788

[117] *** http://www.inria.fr/recherche/equipes/coprin.en.html - The COPRIN Project webpage

[118] ***http://www.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html – ISCAS'85 and ISCAS'89 WebPages at University of Michigan at Ann Harbour

[119] *** http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html - Altera Nios II Acceleration of Mandelbrot Algorithm webpage

## Titluri recent publicate în colecția „TEZE DE DOCTORAT" seria 10: Știința Calculatoarelor

1. **Ionel Muscalagiu** – *Contribuții la implementarea, evaluarea și îmbunătățirea performanțelor tehnicilor de căutare asincrone în cadrul programării bazate pe constrângeri distribuite, ISBN 978-973-625-592-2,* (2007);

2. **Daniel Cioi** – *Contribuții la utilizarea realității virtuale în proiectarea asistată de calculator, ISBN 978-973-625-613-4,* (2008);

3. **Sorin Babii** – *Cercetări privind creșterea performanțelor rețelelor neuronale într-un mediu de calcul distribuit, ISBN 978-973-625-559-5,* (2008);

4. **Norbert Neidenbach** - *Das Service-Management eines IT-Outsourcing-Projektes durch ITIL-Best-Practices, IT-Outsourcing kostenoptimiert planen und steuern, ISBN 978-973-625-660-8,* (2008);

5. **Edwin Hans Wolf** - *Das Geschäftsmodell (Business model) MDS (Managed Desktop Support) im IT-Outsourcing, Leistungserbringung im Rahmen des MDS-Geschäftsmodells, ISBN 978-973-625-661-5,* (2008);

6. **Adrian Zafiu** – *Minimizarea sistemelor decizionale multivalente deterministe și nedeterministe, ISBN 978-973-625-678-3,* (2008);

7. **Daniel Iercan** – *Contributions to the Development of Real-Time Programming Techniques and Technologies, ISBN 978-973-625-719-3,* (2008);

8. **Laurenția Timar** – *Contribuții referitoare la configurarea optimală prin prisma performanță-fiabilitate a unor rețele de dispozitive de achiziția datelor cu aplicabilitate la excavatoarele cu cupe, ISBN 978-973-625-775-9,* (2008);

9. **Dan Cireșan** – *Recunoașterea șirurilor numerice scrise de mână, ISBN 978-973-625-777-3,* (2008);

10. **Emanuel Țundrea** – *Contributions to the modelling and the use of software product lines, ISBN 978-973-625-793-3,* (2008).

---

EDITURA POLITEHNICA