DECIMAL FLOATING POINT ARITHMETIC UNIT BASED ON A FUSED MULTIPLY ADD MODULE

by
Ahmed Mohammed ElShafiey ElTantawy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
August 2011

DECIMAL FLOATING POINT ARITHMETIC UNIT BASED ON A FUSED MULTIPLY ADD MODULE

by
Ahmed Mohammed ElShafiey ElTantawy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Under the Supervision of

Associate Prof. Dr:
Hossam A. H. Fahmy
Principal Adviser

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
August 2011

DECIMAL FLOATING POINT ARITHMETIC UNIT BASED ON A FUSED MULTIPLY ADD MODULE

by
Ahmed Mohammed ElShafiey ElTantawy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS ENGINEERING

Approved by the
Examining Committee

____

Associate Prof. Dr: Hossam A. H. Fahmy, Thesis Main Advisor

____

Prof. Dr: Ashraf M. Salem, Member

____

Associate Prof. Dr: Ibrahim M. Qamar, Member

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
August 2011

Abstract

Although the binary representation is convenient to computer arithmetic; it is not natural to humans. So, decimal arithmetic has proved its necessity in some applications such as business. Accuracy is the main reason to include the decimal floating point specifications in IEEE 745- 2008. This can be performed either in software or hardware. However, hardware implementations speed up the operation with more energy savings. One of the operations in the standard is the Fused Multiply-Add (FMA).

The implementation of a direct Fused Multiply-Add unit rather than successive multiplication and addition has three main advantages: (1) The operation A + (B × C) is performed with only one rounding instead of two, hence more accuracy is obtained. (2) Several components can be shared. Therefore, this results in a reduction in the area. (3) Efficient parallel implementation can result in a reduced delay in the critical path of execution the multiply-add operation. This is besides replacing the delay of fetching and decoding of two instructions (multiply then add) by the delay of only one instruction (FMA).

The FMA is designed to reduce the critical path delay. It uses a minimally redundant radix-10 recoding in the multiplier tree. This leads to a fast carry free reduction for the partial products. It uses a leading zero anticipator (LZA) that anticipates the leading zero count (LZC) of the intermediate result. The LZC is important to determine the final alignment value to reach a correct result that conforms to the standard. After alignment, the rounding position is known, hence a combined add/round module is used to replace a successive addition and rounding steps. The design supports the five rounding directions listed in the IEEE Std 754-2008 as well as two more widely used rounding directions. The FMA also handles exceptions such as overflow, underflow, inexact and invalid operations. The resulting design can be used also for multiplication and addition using the same hardware.

The design is tested using more than 720,000,000 selected test vectors that test the different corner cases for FMA, addition and multiplication and it passed all test vectors correctly.

Contents

List of Tables

List of Figures

Chapter 1

Decimal Floating Point Arithmetic

The invention of numbers is one of the great and remarkable achievements of the human race. The numeration system used by humans have always been subjected to developments in order to satisfy the needs of a certain society at a certain point in time. The variation of these needs from a culture to another along with the evolution of numbering demands led to many different numeration systems across the ages. The traces of these systems were found and tracked by both linguists and archaeologists [1].

However, the fundamental step in developing all number systems was to develop the number sense itself which is the fact that the number is an abstract idea independent of the counted object. The sense of numbers evolved into three main stages. The first was to assign different sets of numbers to different types of objects. The second stage was matching the counted items against other more available and/or accessible ones. For example, counted items of any type were matched against a group of pebbles, grain of corns, or simply fingers. There were many bases for various numeration systems, however, the most common number systems at this stage were based on ten which is logically justified by the ease of counting on fingers. This fact is also, most probably, the main reason for the stability of our nowadays decimal system. Finally, once the sense of numbers is completely developed, distinct names should be assigned to numbers [1].

The need to record the results of counting led to inventing different ways to express numbers in a symbolic written format. This step in the numerals evolution led to two distinct systems, additive and positional. The additive system assigns distinct symbols to certain numbers. A combination of these symbols with the possibility of repeating any symbol as much as necessary can represent any number. This system was used by old Romans and Egyptians. It is easy for counting and simple for calculations, however, it is very complex with advanced arithmetic operations. On the other hand, in the positional system, the symbols representing numbers are positioned in a string with each position indicating a certain weight for the digit inside it. The Chinese and Babylonians used positional number systems. However, a main drawback with these systems was that, there was no symbol for 'zero' to indicate an empty position. This led to both complexity and ambiguity in their numbering system [2].

The decimal numbering system was completely represented by Al-Khwarizmi in his book “The Keys of Knowledge” [3]. In the ninth century, while he was working as a scholar in the House of Wisdom in Baghdad, he developed the science of Algebra based on decimal numeration. The most remarkable achievement was introducing the digit 'zero'. In his book, he indicates that he learned this numbering system from Indians. This system, known as the Hindu-Arabic number system, spreaded gradually in Europe until it almost completely replaced the previously widespread Roman system at the 17th century [2].

The rest of this chapter is organized as follows: Section 1.1 gives an overview about the history of the decimal numeration system in computers. Next, Section 1.2 explains the increasing importance of decimal floating point arithmetic. The decimal floating point standard format with its arithmetic operations is discussed in Section1.3. Section 1.4 surveys the recent published hardware implementations for different decimal floating point operations. Finally, a brief review for processors that support decimal is presented in Section 1.5.

1.1 Decimal Arithmetic in Computers

Since the decimal number system was completely the dominant used numbering system at the 17th century, the first trials for mechanical computers adopted this system for calculations. A well-known example for these mechanical computers is the analytical engine by Charles Babbage [4]. However, the decimal numeration system was questionable again when the computer industry entered the electronic era.

The early electronic computers that depended on vacuum tube technology such as the ENIAC maintained the decimal system for both addressing and numbers. The main representation used was BCD (Binary Coded Decimal) [5]. The superiority of binary system over decimal was first discussed by Burks, Goldstine and von Neumann [6]. Despite the longstanding tradition of building digital machines using decimal numbering system, they argued that a pure binary system for both addressing and data processing would be more suitable for machines based on the two-state digital electronic devices such as vacuum tubes. They stated that binary system will be simpler, more reliable and more efficient than decimal. According to their reports, the simplicity stems from the fact that the fundamental unit of memory is naturally adapted to the binary which leads to more efficient representation and hence more precision. Also, they pointed out to the prevalence of binary system in elementary arithmetic and, of course, logical operations which can be performed much faster than in decimal case. Due to its simplicity, it implies greater reliability due to the reduced number of components. Meanwhile, they underestimated the problem of conversion between binary and decimal, that is more familiar to humans. They argued that this conversion problem can be solved by the computer itself without considerable delay.

On the other hand, other researchers [7] outlined that, the format conversions between decimal and binary can contribute significantly to the delay in many applications that perform few arithmetic operations on huge data workloads. They concluded that the best solution for such case is to build separate arithmetic units. One of them is binary for addressing and the other is decimal for data processing. This debate ended up with two separate lines of computers around the 6th decade of the 20th century, one of them is dedicated to scientific and engineering applications which do complex calculations on small amount of input data and this line uses a fully binary ALU. While the other line is dedicated to the commercial applications which operates on huge data amounts with simple operations so it uses decimal ALU for data processing and binary ALU for addressing [8].

Two main factors led to merging these two lines in a single product between 1960 and 1970. First, the evolution of the solid-state semiconductor technology which contributed to the large scale production of computers with reduced area and cost. Second, the fact that customers used to run commercial applications on scientific computers as well as business-oriented computers were used for some research purposes. These two reasons provided both the ability and the desire to merge both binary and decimal arithmetic units in one ALU [9].

In the 1970s, huge research efforts were exerted to speed up arithmetic operations in binary with limited equivalent efforts for decimal [10, 11, 12, 13, 14]. This led to more popularity for binary systems. Therefore, the early personal computers integrated only binary ALUs with limited decimal operations on the software layer performed on a binary hardware. A remarkable example is the Intel x86 microprocessor which provides some instructions for BCD such as DAA (Decimal Adjustment after Addition) and DAS (Decimal Adjustment after Subtraction) which adjust the binary result of addition or subtraction as if the operation was conducted on decimal hardware [15]. On the other side, binary floating point, which was first proposed in 1914, was supported in the x86 by specialized chips called floating-point accelerators. This was mainly because of its complexity and hence the difficulty to integrate it within the microprocessor chip [16].

The floating point units gained increased popularity, specifically for scientific applications. This led to many designs with different formats and rounding behaviors for arithmetic operations. Therefore it was necessary to standardize a floating-point system so that the same operation can provide the same result on different designs. Thus, the IEEE 754-1985 standard was issued as a binary floating-point standard.

In 1987, another standard for radix independent floating-point arithmetic (IEEE 854-1987) was released [17]. However, it found no echo in the market. This was, from one hand, due to a shortage in the standard itself which lacked some features such as an efficient binary encoding. On the other hand, there was no sufficient demand in the market for decimal floating point processing, particularly that, a decimal floating point unit was still relatively complex enough not to be integrated into a general-purpose microprocessor with the fabrication technologies available at that time [9].

At the beginning of 2000s, there was growing importance of decimal arithmetic in commercial and financial applications, along with technological improvements that allow integration of more complex units. This resulted in a demand for standard specifications for decimal floating-point arithmetic. Thus, the new revision of the IEEE standard for floating-point arithmetic (IEEE 754-2008) includes specifications for decimal floating-point arithmetic [18].

In the next section, the importance of decimal floating point that led to its adoption in the new standard will be explored.

1.2 Importance of Decimal Floating Point Arithmetic

The controversy over binary and decimal numeration system that was opened in the 1970s led initially to merging both systems in the same ALU and ended up with the complete adoption of binary system and depending only on software to perform decimal calculations. Yet, the same debate was reopened again in the 2000s.

Banking, billing, and other financial applications use decimal extensively. Such applications should produce final results that are expected by humans and required by law. Since conversion of some decimal fractions to their binary equivalents may result in endless fractions, this implies a loss of accuracy due to limited storage in case of using pure binary arithmetic. For example, simple decimal fractions such as 0.1 that might represent a tax amount or a sales discount yield an infinitely recurring number if converted to a binary representation ($0.0001100110011\cdots$). This conversion error accumulates and may lead to significant losses in the business market. In a large telephone billing application such an error may end up to $5 million per year [19].

In addition to the accuracy problem, the user of a human oriented application expect trailing zeros to be preserved in different operations. Without these trailing zeros the result of operation appears to be vague. For example, if the specifications of a resistor states that it should be of 1.200 kΩ resistance, this implies that this measurement is to the nearest Ohm. However, if this specification is altered to 1.2 kΩ, then the precision of the measurement may be understood to be to the nearest 100 Ω. This example shows that it is not only the numerical value of a number that is significant, however, the full precision of a number should be also taken into consideration. The binary floating point arithmetic does not follow this rule because of its normalized nature.

Such applications may rely on either a low level decimal software library or use dedicated hardware circuits to perform the basic decimal arithmetic operations. However, as stated in [8], some applications use the decimal processing in 50% to 90% of their work and that software libraries are much slower than hardware designs. So, instead of pure software layering on binary floating-point hardware, one solution is to use decimal fixed-point (DXP) hardware to perform decimal arithmetic. Yet, there are still several reasons to use direct decimal floating-point (DFP) hardware implementations. First, financial applications often need to deal with both very large numbers and very small numbers. Therefore, it is efficient to store these numbers in floating-point formats. Second, DFP arithmetic provides a straightforward mechanism for performing decimal rounding, which produces the same results as when rounding is done using manual calculations. This feature is often needed to satisfy the rounding requirements of financial applications, such as legal requirements for tax calculations. Third, DFP arithmetic also supports representations of special values, such as not-a-number (NaN) and infinity ∞, and status flags, such as inexact result and divide by zero. These special values and status flags simplify exception handling and facilitate error monitoring.

A benchmarking study [20] estimates that many financial applications spend over 75% of their execution time in Decimal Floating Point (DFP) functions. For this class of applications, the speedup for a complete application (including non-decimal parts) resulting from the use of a fast hardware implementation versus a pure software implementation ranges from a factor of 5.3 to a factor of 31.2 depending on the specific application running.

Besides the accuracy and the speed up factors, savings in energy are very important. A research paper estimates that energy savings for the whole application due to the use of a dedicated hardware instead of a software layer are of the same order of magnitude as the time savings. It also indicates that the process normalized Energy Delay Product (EDP) metric, suggested in [21], clearly shows that a hardware implementation for DFP units gives from two to three orders of magnitude improvement in EDP as a conservative estimate if compared with software implementations.

The decimal arithmetic seems to take the same road map of binary. After the domination of binary ALUs in processors, a common trend now is to include either separated Decimal (including DFP) ALUs besides their binary equivalents [22, 23]or to use combined binary and decimal ALUs [24]. This leads to a question whether the decimal arithmetic will dominate if the performance gap between the decimal and binary implementations shrinks enough.

1.3 IEEE Decimal Floating-Point Standard

As previously indicated, there was an increasing need to DFP arithmetic. Hence, there were many efforts to find out the most appropriate DFP formats, operations and rounding modes that completely define the DFP arithmetic. These efforts ended up with the IEEE 754-2008 floating-point arithmetic standard. This section gives a brief overview to this standard [18].

1.3.1 Decimal Formats

The IEEE 754-2008 defines DFP number as : $(-1)^s \times (10)^q \times c$, where: S is the sign bit, q is the exponent, $c = (d_{p-1}d_{p-2}\cdots d_0)$ is the significand where $d_i \in 0,1,2,3,4,5,6,7,8,9$, and p is the precision.

Figure4.3 shows the basic decimal interchange format specified in the IEEE 754-2008 standard. S is the sign bit which indicates either the DFP number is positive ($S = 0$) or negative ($S = 1$) and G is a combination field that contains the exponent, the most significant digit of the significand, and the encoding classification. The rest of the significand is stored in the Trailing Significand Field, T, using either the Densely Packed Decimal (DPD) encoding or the Binary Integer Decimal (BID) encoding, where the total number of significand digits corresponds to the precision, p. The DPD encoding represents every three consecutive decimal digits in the decimal significand using 10 bits, and the BID encoding represents the entire decimal significand in binary.

Before being encoded in the combination field, the exponent is first encoded as binary excess code and its bias value depends on the precision used. There are also minimum and maximum representable exponents for each precision. The different parameters for different precision values are presented in Table 1.1.

Parameter	decimal32	decimal64	decimal128
Total storage width (bits)	32	64	128
Combination Field (w+5) (bits)	11	13	17
Trailing significand Field (t) (bits)	20	50	110
Total Significand Digits (p)	7	16	34
Exponent Bias	101	398	6176
Exponent Width (bits)	8	10	14

Table 1.1: Parameters for different decimal interchange formats

In a decimal floating-point format a number might have multiple representations. This set of representations is called the floating-point number's cohort. For example, if c is a multiple of 10 and q is less than its maximum allowed value, then (s, q, c) and $(s, q+1, c/10)$ are two representations for the same floating-point number and are members of the same cohort. In other words, a one-digit floating-point number might have up to p different representations while a p-digit floating-point number with no trailing zeros has only one representation (a n-digit floating-point number might have fewer than $p-n+1$ members in its cohort if it is near the extremes of the format's exponent range). A zero has a much larger cohort: the cohort of $+0$ contains a representation for each exponent, as does the cohort of -0. This property is added to decimal floating-point to provide results that are matched to the human sense by preserving trailing zeros as discussed before. Hence, different members of a cohort can be distinguished by the decimal-specific operations. In brief, for decimal arithmetic, besides specifying a numerical result, the arithmetic operations also select a member

S (sign)	*G* (combination field)	*T* (trailing significand field)

Figure 1.1: Decimal interchange floating-point format

of the result's cohort. And thus, decimal applications can make use of the additional information cohorts convey.

1.3.2 Operations

The standard specifies more than 90 obligatory operations classified into two main groups according to the kinds of results and exceptions they produce:

- Computational Operations:

These operations operate on either floating-point or integer operands and produce floating-point results and/or signal floating-point exceptions. This general category can be also decomposed into three classes of operations.

General-computational operations: produce floating-point or integer results, round all results and might signal floating-point exceptions. For example, all arithmetic operations such as addition, subtraction, multiplication and so on.

Quiet-computational operations: produce floating-point results and do not signal floating-point exceptions. It includes operations such as negate, absolute, copy and others.

Signaling-computational operations: produce no floating-point results and might signal floating point exceptions; comparisons are signaling-computational operations.

- Non-Computational Operations:

These operations do not produce floating-point results and do not signal floating-point exceptions. It includes, for example, operations that identify whether a DFP number is negative/positive, finite/infinite, Zero/Non-zero and so on.

Operations can be also classified in a different way according to the relationship between the result format and the operand formats:

Homogeneous operations: in which the floating-point operands and floating-point results are all of the same format

FormatOf operations: which indicates that the format of the result, independent of the formats of the operands.

Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to an infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format . In some cases, exceptions are raised to indicate that the result is not the same as expected or invalid operations. On the other hand, as indicated before, a floating-point number might have multiple representations in a decimal format. All these operations, if producing DFP numbers, do not only specify the correct numerical value but they also determines the correct member of the cohort.

It should be highlighted that, besides the required operations for a standard compliant implementation, there are other recommended operations for each supported format. These operations mainly include the elementary functions such as sinusoidal and exponential functions and so on.

The details of some decimal operations will be presented later in next chapters.

1.3.3 Rounding

There are five rounding modes defined in the standard, Round ties to even, Round ties to away, Round toward zero, Round toward positive infinity, and Round toward negative infinity. Also, there are two well-known rounding modes supported in the Java BigDecimal class [25]. Table 1.2 summarizes the different rounding modes with their required action.

Rounding Mode	Rounding Behavior
Round Ties To Away RA	Round to nearest number and round ties to nearest away from zero, the result is the one with larger magnitude.
Round Ties to Even RNE	Round to nearest number and round ties to even, the result is the one with the even least significand digit.
Round Toward Zero RZ	Round always towards zero , the result is the closest DFP number with smaller magnitude.
Round Toward Positive RPI	Round always towards positive infinity, the result is the closest DFP number greater than the exact result.
Round Toward Negative RNI	Round always towards negative infinity, the result is the closest DFP number smaller than the exact result.
Round Ties to Zero RZ	Round to the nearest number and round ties to zero.
Round To Away RA	Round always to nearest away from zero, the result is the one with larger magnitude.

Table 1.2: Different Rounding Modes

1.3.4 Special numbers and Exceptions

Special numbers:

Operations on DFP numbers may result in either exact or rounded results. However, the standard also specifies two special DFP numbers, infinity and NaN.

Normal and Subnormal numbers:

A normal number can be defined as a non-zero number in a floating-point representation which is within the balanced range supported by a given floating-point format. The magnitude of the smallest normal number in a format is given by $b^{e_{min}}$, where b is the base (radix) of the format and

Operation	Exception	Operation	Exception
$\infty + x = \infty$	None	$\infty / x = \infty$	None
$\infty + \infty = \infty$	None	$x / \infty = 0$	None
$\infty - x = \infty$	None	$\infty / \infty = NaN$	Invalid
$\infty - \infty = NaN$	Invalid	$\sqrt{\infty} = \infty$	None
$\infty \times x = \infty$	None	$\sqrt{-\infty} = NaN$	Invalid
$\infty \times \infty = \infty$	None	$\pm x / 0 = \pm\infty$	Division by Zero
$\infty \times 0 = NaN$	Invalid	$subnormal \div x$	Underflow

Table 1.3: Examples of some DFP operations that involve infinities

e_{min} is the minimum representable exponent. On the other hand, subnormal numbers fill the underflow gap around zero in floating point arithmetic. Such that any non-zero number which is smaller than the smallest normal number is 'subnormal'.

Infinities:

Infinity represent numbers of arbitrarily large magnitudes, larger than the maximum represented number by the used precision. That is: $-\infty < \{each\, representable\, finite\, number\} < +\infty$. In Table 1.3, a list of some arithmetic operations that involve infinities as either operands or results are presented. In this table, the operand x represents any finite normal number.

NaNs (Not a Number):

Two different kinds of NaN, signaling and quiet, are supported in the standard. Signaling NaNs (sNaNs) represent values for uninitialized variables or missing data samples. Quiet NaNs (qNaNs) result from any invalid operations or operations that involve qNaNs as operands. When encoded, all NaNs have a sign bit and a pattern of bits necessary to identify the encoding as a NaN and which determines its kind (sNaN vs. qNaN). The remaining bits, which are in the trailing significand field, encode the payload, which might contain diagnostic information that either indicates the reason of the NaN or how to handle it. However, the standard specifies a preferred (canonical) representation of the payload of a NaN.

Exceptions:

There are five different exceptions which occur when the result of an operation is not the expected floating-point number. The default nonstop exception handling uses a status flag to signal each exception and continues execution, delivering a default result. The IEEE 754-2008 standard defines these five types of exceptions as shown in Table 1.4.

Exceptions	Description	Output
Invalid Operation (Description shows only common examples)	-Computations with sNaN operands -Multiplication of $0 \times \infty$ -Effective subtraction of infinities -Square-root of negative operands -Division of $0/0$ or ∞/∞ -Quantize in an insufficient format -Remainder of $x/0$ or ∞/x (x: finite non zero number)	Quite NaN
Division by Zero	The divisor of a divide operation is zero and the dividend is a finite non-zero number.	Correctly signed ∞
Overflow	The result of an operation exceeds in magnitude the largest finite number representable.	The largest finite number representable or a signed ∞ according to the rounding direction.
Underflow	The result of a DFP operation in magnitude is below $10^{e_{min}}$ and not zero	zero, a subnormal number or$\pm 10^{e_{min}}$ according to rounding mode.
Inexact	The final rounded result is not numerically the same as the exact result (assuming infinite precision)	The rounded or the overflowed result.

Table 1.4: Different Types of Exceptions

1.4 Standard Compliant Hardware Implementations of DFP Operations

As mentioned earlier, support of DFP arithmetic can either be through software libraries such as the Java BigDecimal library [25], IBM's decNumber library [26], and Intel's Decimal Floating-Point Math library [27], or through hardware modules. Many hardware implementations have been introduced in the last decade to perform different operations defined in the standard. This includes adders, multipliers, dividers and some elementary functions and others.

Many DFP adder designs have been proposed for the last few years. Thompson et al. [28] proposed the first published DFP adder compliant with the standard. A faster implementation with architectural improvements is proposed in [29]. An extension and enhancement of this work is proposed again in [30]. Further improvements are proposed by Vazquez and Antelo in [31]. Fahmy et al [21] proposed two other different adder implementations, one for high speed and the other for low area. Yehia and Fahmy [32] proposed the first published redundant DFP adder to allow for a carry-free addition.

There are also many designs for integer decimal multiplication [33][34]. Erle et al. [35] published the first serial DFP multiplier compliant with the IEEE 754-2008 standard. While Hickmann et al. [36] published the first parallel DFP multiplier. Raafat et al. [37] presented two proposals to decrease the latency of parallel decimal multiplication. Also Vazquez, in [38], proposed two high performance schemes for DFP multiplications, one optimized for area and the other optimized for delay.

An incomplete decimal FMA based floating-point unit is developed and combined with a known binary FMA algorithm in [24] . This incomplete unit supports the decimal64 and binary64 formats and claims conformance to the standard's specification for rounding and exceptions, but not underflow and subnormal numbers. However, the first known conforming hardware implementation for decimal FMA is presented in [39]. More details about these implementations are discussed in Chapter 2.

Early proposals for DFP dividers are introduced in [40, 41]. However, the first DFP standard compliant designs can be found in IBM POWER6 [23] and Z10 [23] microprocessors. Also, another compliant DFP divider is proposed by Vazquez in[42].

Since the IEEE 754-2008 standard has been approved, many designs and implementations for elementary functions in decimal are introduced. For example, different proposals for modifying the CORDIC method to work on decimal without conversion to binary [43]. The CORDIC algorithms is also used to implement different transcendental functions [44]. A comprehensive library of transcendental functions for the new IEEE decimal floating-point formats is presented in [45]. There is also different proposal for a DFP logarithmic function in [46] and [47].

1.5 IEEE 754-2008 DFP Support in Microprocessors

As discussed in section 1.2, decimal arithmetic was supported by many processors. Moreover, the first generations of processors, such as ENIAC, support only decimal. However, the zSeries DFP facility was introduced in the IBM System z9 platform. The z9 processor implements the facility with a mixture of low-level software - using vertical microcode, called millicode - and hardware assists using the fixed point decimal hardware [48]. Because the DFP was not fully defined when the z9 processor was developed, there was only basic hardware support for decimal. Yet, more than

50 DFP instructions are supported in millicode. Millicode enables implementing complex instructions where hardware support is not possible, and to add functions after hardware is finalized. This leaves System z9 as the first machine to support the decimal floating point (DFP) instructions in the IEEE Standard P754.

The POWER6 is the first processor that implements standard compliant decimal floating-point architecture in hardware. It supports both the 64-bit and the 128-bit formats. As described in [49, 50], 54 new instructions and a decimal floating-point unit (DFU) are added to perform basic DFP operations, quantum adjustments, conversions, and formatting. The POWER6 implementation uses variable-latency operations to optimize the performance of common cases in DFP addition and multiplication.

The IBM System z10 microprocessor is a CISC (complex instruction set computer) microprocessor. It implements a hardwired decimal floating-point arithmetic unit (DFU) which is similar to the DFU of the POWER6 with some differences [23, 22]. The differences are mainly about the DXP unit architecture and its interface with DFP unit. However, many of the DFP operations are implemented in hardware in both POWER6 and System z10, but there are other operations that are not. For example, the FMA operation which is required for a standard compliant DFP unit is not implemented in hardware.

In this chapter, an introduction to the decimal floating-point arithmetic is presented. The second chapter gives details about the FMA operation and survey the its binary and decimal implementaions. The third chapter discusses the basic blocks used in the FMA design. In the fourth chapter, we introduce our own propsal for a decimal fused multiply-add unit. We exten this unit to a pipelined decimal arithmetic unit that performs floating-point multiplication, addition and fused multiply-add. Finally, at the last chapter, we present the results of our proposal.

Chapter 2

Fused Multiply-Add Operation

In this chpater, we introduce the specifications of the Fused Multiply-Add (FMA) operation defined in the standard in detail. We also explain the importance of this operation and survey the recent hardware implementations for this operation in both binary and decimal.

2.1 Multiply-Add Operation Importance

The multiply-add operation is fundamental in many scientific and engineering applications. For examples, Digital Signal Processing (DSP) involve algorithms that utilize the (A × B) + C single-instruction equation. This is illustrated by an investigation that shows that almost 50% of the multiply instructions are followed by add or subtract instructions [51]. Some of the steps involved in the multiply-add operation such as the multiplication and the summation of the result with another operand can be performed concurrently. Hence, the multiply-add can be considered as a single operation called fused multiply-add (FMA).

The implementation of a direct fused multiply-add unit has three advantages [52]:

(1) The operation (A × B)+C is performed with only one rounding instead of two, hence more accuracy is obtained.

(2) Several components can be shared. Therefore, it results in a reduced area.

(3) Efficient parallel implementation can result in a reduced critical path delay. This is besides replacing the delay of fetching and decoding of two instructions (multiply then add) by the delay of only one instruction (FMA).

Moreover, the FMA can still perform other standard floating-point operations by replacing operands with constants. Making C = 0.0 or B = 1.0 the FMA unit performs floating-point multiplication or floating-point addition respectively.

On the other hand, combining the two operations in a single instruction increases the complexity of the hardware. Consequently, the pure multiplication or pure addition using FMA unit will have greater latencies than expected when executing them on a dedicated floating-point multiplier or adder respectively. Also, it needs three read ports to read the three operands concurrently, otherwise, the third operand must be read in a separate cycle. A good design should keep the FMA advantages and efficiently address its problems.

Since 1990, many algorithms that utilize the A×B+C single instruction operation have been introduced, for applications in DSP and graphics processing [53, 54], FFTs [55], division [56], etc. To accommodate the increased use of the FMA instruction and to consider the advantages of the FMA operation, several commercial processors have implemented embedded FMA units [57, 53, 54, 55, 56]. However, decimal FMA operation is not included as hardware in any of current processors. Yet, there are only two implementations one of them is incomplete [24] and the other leaves a room for improvements [39].

2.2 Fused Multiply-Add (FMA) standard specifications

As the standard states [18], the Fused Multiply-Add operation for the three operands (A, B, C) 'FMA(A,B,C)' computes (A × B) + C as if they were with unbounded range and precision, with rounding only once to the destination format. Moreover, no underflow, overflow, or inexact exception can arise due to multiplication, but only due to addition; and so Fused Multiply-Add differs from a multiplication operation followed by an addition operation. The preferred exponent is min(Q(A) + Q(B), Q(C)) where Q(x) means the exponent of operand x.

This definition of the FMA operation highlights two important restrictions: the intermediate unbounded result of the multiplication and the single rounding step after addition. This clearly shows that this operation produces more accurate result than a multiplication with a result rounded to the required precision then followed by addition with the final result rounded again.

The standard also stresses that exception decisions are taken based on the final result and not due to the multiplication step. Hence, to have a complete definition of the operation, we should explore the effect of this operation on the different flags.

- Invalid Exception:

The invalid operation exception is signaled if and only if there is no usefully definable result. The standard specifies three cases for the FMA operation. First, the exception is due to invalid multiplication such as FMA(0,$\pm\infty$, c) or FMA($\pm\infty, 0, \text{c}$). In this case the invalid flag is raised unless c is a quiet NaN; if c is a quiet NaN then it is implementation defined whether the invalid operation exception is signaled. In other words, if c is a quiet NaN, it is optional to signal invalid operation or not but the implementation should maintain a unified behavior. Second, the exception may result from an invalid final addition step. For example, FMA(+|c|,$+\infty$, $-\infty$), FMA(+|c|,$-\infty$, $+\infty$) or any combination that leads to $(|\infty| - |\infty|)$. In the last example, c is any representable floating point number and is not quiet NaN. Finally, the invalid exception is signaled if any of the three operands is signaling NaN. In all these cases, the default result of the operation shall be a quiet NaN that may provide some diagnostic information.

- Overflow Exception:

There is nothing special with the overflow exception in case of FMA operation. The overflow exception shall be signaled if and only if the destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result in case of unbounded exponent range. The default result shall be determined by the rounding-direction attribute and the sign of the intermediate result as follows:

a) roundTiesToEven and roundTiesToAway carry all overflows to $\infty$with the sign of the intermediate result.

b) roundTowardZero carries all overflows to the format's largest finite number with the sign of the intermediate result.

c) roundTowardNegative carries positive overflows to the format's largest finite number, and carries negative overflows to $-\infty$.

e)roundTowardPositive carries negative overflows to the format's most negative finite number, and carries positive overflows to $+\infty$.

In addition, under default exception handling for overflow, the overflow flag shall be raised and the inexact exception shall be signaled.

- Underflow Exception:

The underflow exception shall be signaled when a tiny non-zero result is detected. The way to detect an underflow is not the same for binary and decimal. For binary formats, it is optional to detect this tininess before rounding or after rounding. However, the implementation has to detect underflow in the same way for all binary operations. Thus in binary, underflow exception shall be raised either:

a) after rounding, when a non-zero result computed as though the exponent range were unbounded would lie strictly between ± $b^{e_{min}}$ or

b) before rounding, when a non-zero result computed as though both the exponent range and the precision were unbounded would lie strictly between ± $b^{e_{min}}$. Where $|b^{e_{min}}|$ is the magnitude of the smallest normal floating point number.

For decimal formats, tininess is detected only before rounding, when a non-zero result computed as though both the exponent range and the precision were unbounded would lie strictly between $\pm b^{e_{min}}$. The default exception handling for underflow shall always deliver a rounded result which might be zero, subnormal, or $\pm b^{e_{min}}$. In addition, under default exception handling for underflow, if the rounded result is inexact - that is, it differs from what would have been computed were both exponent range and precision unbounded - the underflow flag shall be raised and the inexact exception shall be signaled. If the rounded result is exact, no flag is raised and no inexact exception is signaled.

- Inexact Exception:

If the rounded result of an operation is inexact, i.e. it differs from what would have been computed when both exponent range and precision are unbounded, then the inexact exception shall be signaled. The rounded or overflowed result shall be delivered to the destination.

In the rest of this thesis, the following symbols are given to different flags in any numerical example to simplify their representation:

x: inexact o: overflow u: underflow i:invalid

Table 2.2 compares the results of FMA operation with a multiplication followed by addition. All examples are for 64-bit format of decimal floating point. It shows how the FMA operation produces more accurate result. The flags due to addition in case of multiply then add are Ored with stored flags due to the multiplication step. The rounding mode is indicated in each operation. Refer to Table 1.2 for more details. For complete understanding of the table, it should be noted that the maximum representable exponent in the 64-bit decimal format is $expMax = 369$. Also, before any addition operation the operands are aligned to have the same exponent then the result is shifted to the left to reach or approach the preferred exponent (the minimum of the two operands). In case of FMA, the addition is performed on the exact multiplication result and the addend; while in the successive multiplication and addition steps the addition is performed on the rounded multiplication result and the addend.

2.3 Binary FMAs

As stated previously, several general purpose processors are designed including a binary fused Multiply-Add unit (FMA) .The first fused-multiply add operation was introduced in 1990 on the IBM RS/6000 for the single instruction execution of the equation A×B+C with single and double precision floating-point operands [58, 57]. The architecture of binary floating-point FMAs varies from a basic architecture [57, 52, 59] to a fully parallel one proposed in [60]. However, there are many variants from these architectures that either target better throughput in a pipelined design or efficient execution for the addition and multiplication operations on the FMA. Furthermore, there are some proposals that suggest minor modifications that can result in faster or reduced area designs. Hence in the following, we will first discuss the two main architectures in [52] and [60] then follow them by their variants.

2.3.1 Basic Architecture

The FMA architecture proposed before [52], implemented in several floating–point units of general purpose processors [57, 52, 59], is shown in Figure 2.1 (a). The necessary steps are:

1. Multiplication of A and B to produce an intermediate product A × B in carry-save representation.

2. Bit inversion and Alignment. To reduce the latency, the bit inversion and alignment of C is done in parallel with the multiplication. The alignment is implemented as a right shift by placing C to the left of the most–significant bit of A × B. Two extra bits are placed between C and A × B to allow correct rounding when C is not shifted. This means that the alignment requires a (3p+2) bits right shifter (three p-bits for operand C and the multiplication result of A and B, and the rounding extra two bits). These calculations depend on the fact that binary floating point numbers are normalized.

3. The aligned C is added to the carry-save representation of A×B. Only the (2p) least-significant bits of the aligned C are needed as input to the 3:2 CSA, because the product has only (2p) bits. The (p+2) most-significant bits of the aligned C are concatenated at the output of the CSA to obtain the (3p+2) bits sum.

4. Addition of the carry and sum words in a (3p+2) bits adder and determination of the shift amount for normalization by means of a LZD (Leading Zero Detector).

5. Possible complement if the result is negative. The 2’s complement requires an adder.

6. Normalization of the final result to determine the rounding position.

7. Rounding of the result with a possible one bit shift right in case of carry propagation to the most significant bit. The rounding step is itself an addition that results in a carry propagation, yet, it only operates on (p) bits.

This leaves this basic design with a critical path of : a multiplier tree, a CSA, an adder of width (3p+2), an incrementer of width (3p+2) for complementation, a shifter of width (3p) and finally an adder of (p) bits.

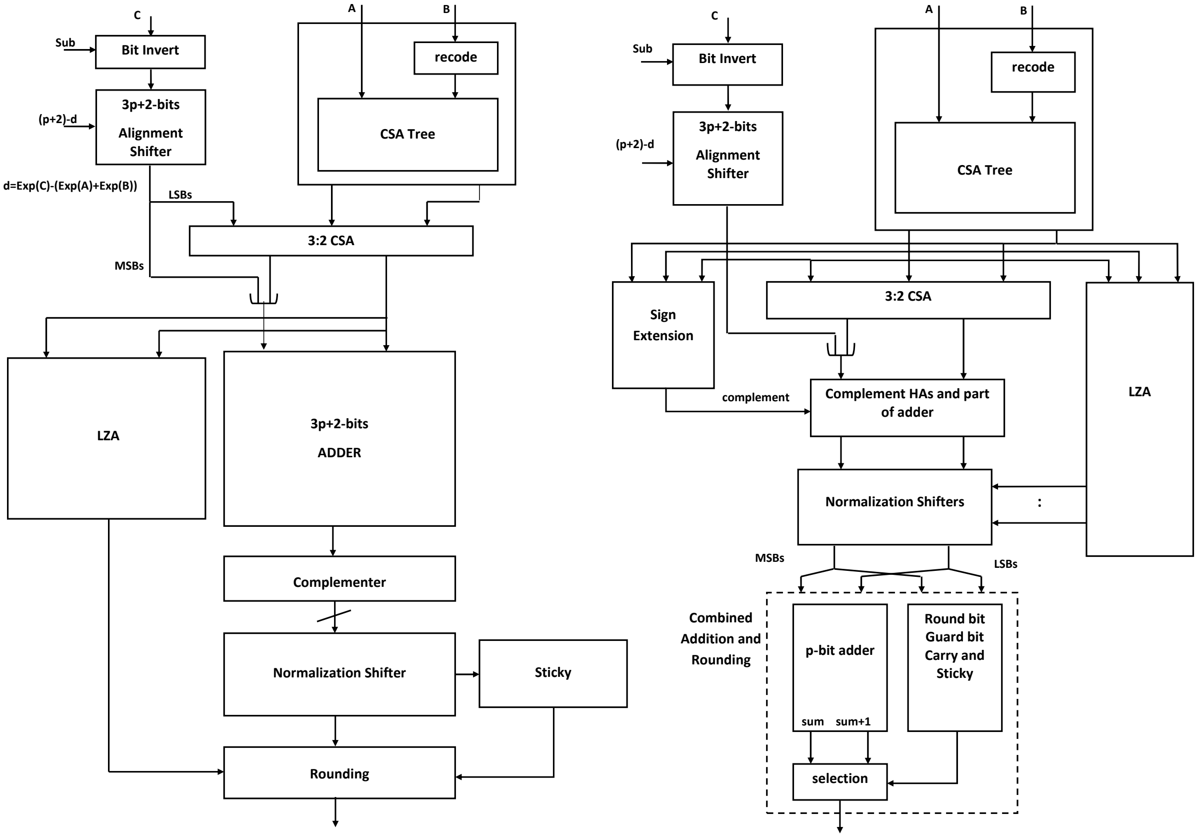


Figure 2.1: (a) Basic Architecture (b)Parallel Architecture

2.3.2 Parallel Architecture

The objective of the FMA architecture proposed in [60] is to reduce the overall delay of the FMA operation. It uses one of the approaches used in floating–point addition and multiplication to reduce latency which is to combine addition with rounding [61, 62, 63]. For this approach, in floating–point addition and multiplication the order of normalization and rounding is interchanged. However, this seems impractical to do for FMA; because the rounding position is not known before the normalization. The solution is to perform the normalization before the addition. The resulting scheme is shown in Figure 2.1 (b). This implementation shares several characteristics with the basic implementation:

(1) The alignment is performed in parallel with the multiplication, by shifting the addend A and

(2) The result of multiplication is in carry-save representation and a 3:2 CSA is used to obtain a redundant representation of the unnormalized and unrounded result.

On the other hand, the proposal is different in the following aspects:

(1) Normalization is performed before the addition. The two resulting vectors of the 3:2 CSA are normalized (before the addition) and the add/round operation is performed. By performing the normalization before the addition the final result is always normalized, which simplifies the rounding step. Note that, in this case there are two normalization shifters, instead of one as in the basic approach. This requires anticipating the leading zero count of the result from the added/subtracted operands. To hide the delay of this anticipation, it is overlapped with the normalization shifter. Hence, the shift count is obtained starting from the most–significant bit (MSB), and once the MSB bit is obtained, the normalization shift can start.

(2) There is a dual adder of (p) bits to obtain both the sum and the sum+1 of the (p) most-significant bits of the two resulting vectors from the normalization step. The final answer is selected based on the rounding decision. Hence, the inputs to the add/round module are split into two parts: the (p) most-significant bits are input to the dual adder and the remaining least-significant bits are inputs to the logic for the calculation of the carry into the most–significant part and for the calculation of the rounding and sticky bits.

(3) Sign detection. This block anticipates the sign of the intermediate result then complements the outputs of the CSA, the sum and carry words, when the result would be negative. Hence, this guarantees a positive intermediate result selected from the sum or the sum+1 produced from the dual adder.

(4) Advance part of the adder. Since the sign detection as well as the part of the LZA that cannot be overlapped have a significant delay, half adders (HAs) are placed to perform some parts of the dual adder before the inversion/ normalization.

Hence, this design has a critical path of : a multiplier tree, a CSA, shifter of width (3p+2), a LZA (3p+2) overlapped with normalization of width (3p+2), and at the end an adder of width (p) bits.

Bruguera and Lang conclude that this improved design achieves about 15% to 20% reduction in the delay if compared to the basic architecture. They do not report area, however, it is excepted to have more area due to two normalization shifters for both the sum and the carry resulting from the CSA rather than only one shifter in the basic architecture. Also, the dual adder requires more hardware to produce the two sums (sum and sum+1). This is in addition to the LZA and the sign detection modules.

2.3.3 Other Variants

While previous parallel FMA architectures compute the three operations with the same latency, Bruguera and Lang propose another architecture that permits to skip the first pipeline stages, those related with the multiplication $A \times B$, in case of an addition [64]. For instance, for an FMA unit pipelined into three or five stages, the latency of the floating-point addition is reduced to two or three cycles, respectively. To achieve the latency reduction for floating-point addition, the alignment shifter, which in previous organizations is in parallel with the multiplication, is moved so that the multiplication can be bypassed. To avoid increasing the critical path, a double-datapath organization is used, in which the alignment and normalization are in separate paths. Moreover, they use the techniques developed previously of combining the addition and the rounding and of performing the normalization before the addition.

The delay of the proposed architecture has been estimated and compared with the single-datapath FMA architectures. The conclusion is that there is a significant delay reduction in the computation of a floating-point addition, about 40% with respect to the basic FMA, and around 30% with respect to the single-datapath FMA with normalization before the addition.

On the other hand, an optimization in the LZA proposed in [60] is presented in [65]. The new LZA accepts three operands directly and generates a pre-encoding pattern directly from them. This approach results in a 16.67% improvement in the LZA speed and 19.63% reduction it its area if compared to the LZA proposed in [60] which accepts only two operands.

Another design that improves design[60] is a fully pipelined implementation of single precision multiply-add-fused operation presented in [66] that can be easily extended for double precision format. The proposed FMA unit is also based on the combination of the final addition with rounding, which is used to reduce the latency of the FMA unit. However, the authors in [66] also present a new implementation of the LZA, in which the shift conditions are subdivided into several types based on the alignment of the addend and the products. Together with a three-step normalization method, the largest shift amount for normalization is reduced to just $\lfloor p/2 \rfloor$-bits . In contrast, it is about p-bits in[60]. Furthermore, only one of those three normalization steps is in the critical path. The other two steps are annihilated into the time gap of other modules. Because the delay of a shifter is mostly generated by the connective wire, the decrease of the shift amount reduces the length of the wire in a large amount and consequently reduces the time delay significantly.

There is another proposal [67] that presents an FMA unit that can perform either one double-precision or two parallel single-precision operations using about 18% more hardware than a conventional double-precision FMA unit [60] and with only 9% increase in delay. It redesigns several basic modules of double-precision FMA unit to accommodate the simultaneous computation of two single-precision FMA operations.

A modified dual-path algorithm [68] is proposed by classifying the exponent difference into three cases and implementing them with CLOSE and FAR paths, which can reduce latency and facilitate lowering power consumption by enabling only one of the two paths. In addition, in case of ADD instructions, the multiplier in the first stage is bypassed and kept in stable mode, which can significantly improve ADD instruction performance and lower power consumption. The overall FMA unit has a latency of 4 cycles while the ADD operation has 3 cycles. Compared with the conventional double-precision FMA [52], about 13% delay is reduced and about 22% area is increased, which is acceptable since two single-precision results can be generated simultaneously.

Since, subnormal numbers are the most difficult type of numbers to implement in float-point units. Because of the difference between normalized and denormalized number, it is a complex problem to fuse the denormalized number into traditional MAF datapath without adding a dedicated unit or introducing extra latency. Many designs avoid handling them in hardware. However, on the other hand, the denormalized number processing costs extra clock cycles in software implementations. An on-fly floating point denormalized number processing implemented in a multiply-add-fused (FMA) with little extra latency is presented in [69].

2.4 Decimal FMAs

There are only two FMA designs in the literature. However, one of them is incomplete and the other is not completely verified. Besides, both designs leave a room for improvement. This section explains in brief the two architectures.

2.4.1 Monsson's Architecture

The first design is proposed in [24] and shown in Figure 2.2. The design is for a combined binary and decimal FMA, however, in this part we are only interested in decimal. In the following, an overview of the basic blocks in this design is presented.

Multiplier Tree:

The partial product generation scheme used in this multiplier utilizes this set of multiplier multiples ($0,A,2A,4A,5A$) to generate the partial products

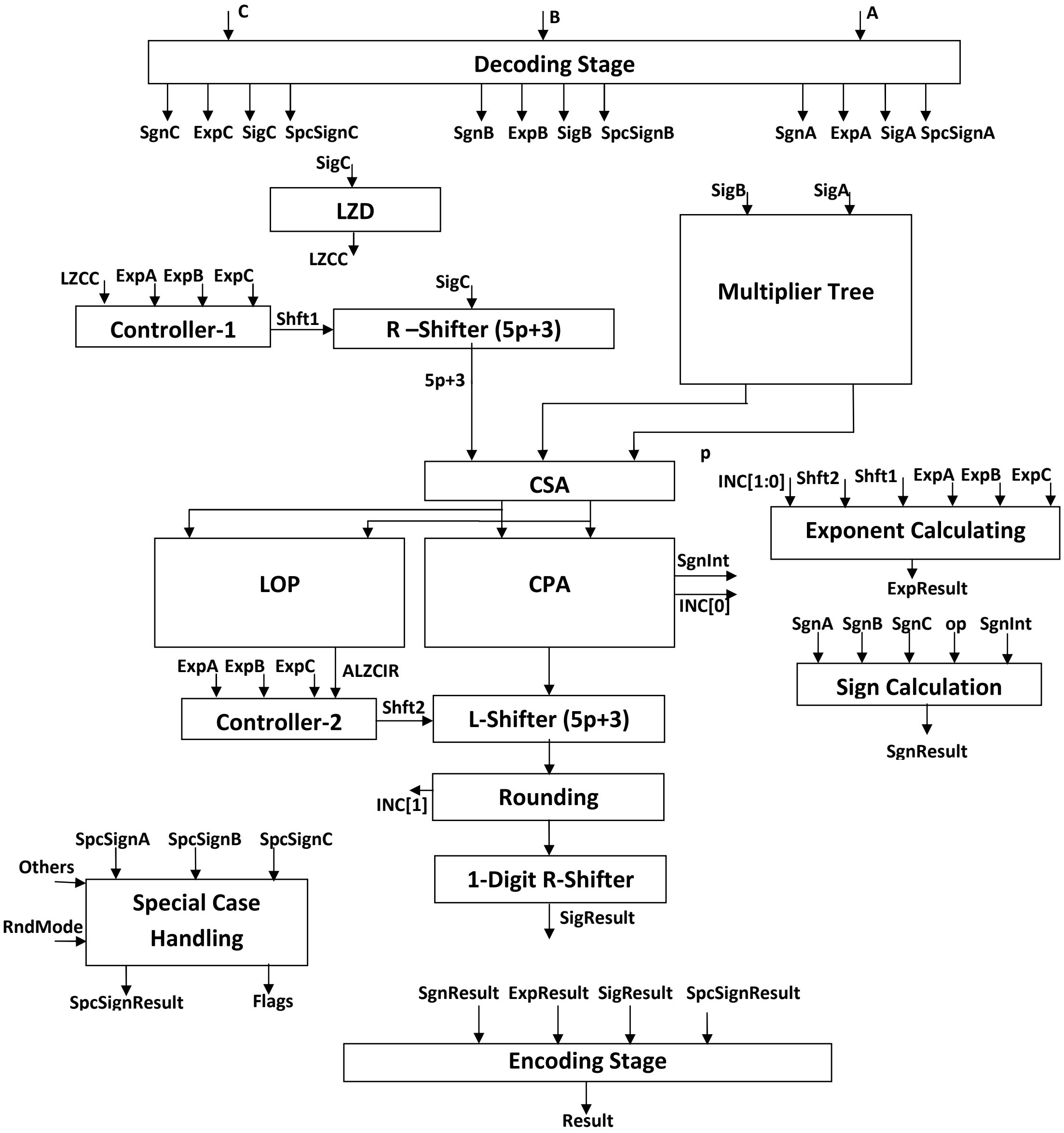


Figure 2.2: Monsson’s Architecture

where A is the multiplier. Each digit in the multiplicand (B) results in two partial products selected using two muxes. This technique is suggested by Erle and Schulte in [70].

It uses regular a 8421-BCD encoding to encode the partial products. The set of multiplier multiples is generated such that '2A' is a left shift of a single bit and adds a correction of '+6' to the shifted digit if $a > 4$ where a is any digit in the multiplier. This correction is needed because the left shift of one bit is equivelent to multiplication by 2 for a binary string; not a decimal one. Therefore, if the binary multiple exceeds '10' (i.e. $a > 4$), '+6' is added to set the digit again in the BCD format. The multiple '4A' is a double application of '2A'. While, the multiple '5A' is a left shift of three bits and adds five to the shifted digit if the LSB of the unshifted digit is one. This is because any digit whose value is odd will initially become a five ('0101'), and any digit whose value is even will initially become a zero. Further, any digit whose value is (≥ 2) will produce a carry-out of (1,2,3 or 4). Thus, when a carry-out does occur, it will not propagate beyond the next more significant digit, which has a maximum value of five before adding the carry.

The reduction tree rows is double the number of digits in the multiplication operands, so for decimal64 where the precision is 16 digits, the reduction tree supports 32 operands. It uses a non-speculative addition for the reduction tree which is simply a Wallace tree of binary CSAs. Every fourth carry out (at digit boundaries) drives a counter for the correction logic.

Alignment:

This design uses a one way shifter and sets the zero of the shifter at the MSB of the 5p + 3 digit wide shifted C. A positive shift value shifts C down to have the same exponent as the multiplication result. The extra positions that decimal addition needs after the base of the product will be shifted out into the sticky bit during normalization.

Addition:

The addition step of the FMA combines the two vectors resulting from the multiplication and the aligned addend ‘C’ into a single result. This step consists of a CSA, a CPA and some correction logic. First, the three vectors are reduced to only two using a decimal CSA.

For fast operation, a binary adder with a carry prefix network is used to add the two vectors resulting from the CSA. This requires a precorrection step by adding ‘+6’ digitwisely. For subtraction one operand must nine’s complement and pre-correct each digit with +6. This is the same as fifteen’s complement which is a simple bit inversion. The adder used is a compund adder that produces both the sum and the sum+1 of the two inputs. This avoids the need to complement the result in case of negative intermediate result. Since the 10’s complement requires an incrementer (adder) to perform (+1) addition. However, in case of the compond adder, if the intermediate result is negative the sum is 9’s complemented (no need for an adder) to produce the final result. On the other hand, if the intermediate result is positive and the effective operation is subtraction, the sum+1 is selected. Both results, sum and sum+1 are computed in parallel in the compound adder. In case of effective addition, the sum is selected. Finally, in all these cases, the result must be postcorrected.

It is important to note that the compound adder in Mosson’s proposal is of with 5p+3 digits.

Leading Digit Prediction (LDP):

Monsson proposes an algorithm for leading digit prediction in decimal operands. The LDP module accepts two inputs and returns a predicted position for the leading non-zero digit, hence it can be more accurately named leading non-zero digit prediction (LD). In case of effective addition, the leading non-zero digit is predicted to be at the same position of the leading non-zero digit of the addend operands. This preliminary anticipation ignores the possibility for a carry out at this digit. In other words, it gives an anticipation with a possible 1-digit error.

In case of effective subtraction, if it is assumed that a decimal digit by digit subtraction is made and the result is the string of digits Y with each digit labeled by a subscript (Y_i). Then each digit in the result is in the range $\{$-9, . . . , 9$\}$. This string is tracked from the MSD to the LSD searching for a specific sequence. Exploring the different digit patterns allows to enode the resulting string Y in an anticipated LD position.

Finally, a combinatorial logic is designed to perform this algorithm. It is important to note that this anticipation is over a 5p+3 digits width that results from the CSA.

This explains the main strategy used, however, the research discusses more details that is not mentioned here. However, in the next chapter, we explore the leading zero anticipators (LZA) which performs the same functionality of the leading digit predictors in more details.

Normalization:

Taking into consideration both the leading zero count and the preferred exponent of the result according to the standard.The normalization is a two stage left shifter starting with a coarse shift based on the LDP and finishing with a fine correcting shift. The coarse shifter operates on the preliminary anticipation of the LDP with sticky bit calculation. The shifter is of 3p+1 width. Since, it is the maximum required value of shift to either cover leading zeros or to reach the preferred exponent.

Rounding:

The rounding step is split up in two parts: A rounding decision and an increment. The rounding decision is simply a selection of one of the different calculations based on the rounding mode and the operation mode. A combinational Boolean logic triggers the round up decision. The rounding itself is performed by a decimal CPA of width (p-digits) which acts as an incrementer.

After rounding the result may overflow and it will have to be renormalized, and furthermore the exponent has to be updated.

Design Disadvantages:

First of all, according to Monsson, the design is tested over only 30 test vectors and some of them failed. The researcher states clearly that the design does not work correctly in case of underflow and for some cases near overflow. However, even if not mentioned in the thesis, it is very likely that it does not work correctly for many other corner cases, because 30 test vectors is a very tiny number of tests compared to the massive testing space in decimal floating point operations.

Also, regardless the functionality and the cost required to correct it, the design itself is not efficient. It uses a very wide datapath (5p+3) which is wider than necessary. The worst case carry propagation happens only for 3p width. Moreover, the design uses two successive adders for initial addition and then for rounding, although this can be avoided as shown in Burgura's binay implementation.

2.4.2 SillMinds' Architecture

This is the first published design that was tested on a considerable test vectors and claims a complete conformance to the standard. The design in [39] is shown in Figure 2.3.

Multiplier Tree:

The proposed design generates multiples of the multiplicand based on the multiplier digits using a signed digit recoding technique to generate the partial products in parallel. It uses the Signed Digit Radix-10 architecture proposed in [33]. The radix-10 recoding converts the digit set [0, . . . , 9] into the SD set [-5, . . . ,5]. It encodes partial products in a redundant BCD (i.e. redundancy is only in the representation of each digit and not the whole number) format to simplify both the generation and the reduction steps of the partial products.

Alignment:

The addend C is shifted either to the right or to the left within 4p width to align it to the expected multiplication result. The fractional point is assumed to follow the least significant digit of the multiplication result.

Addition:

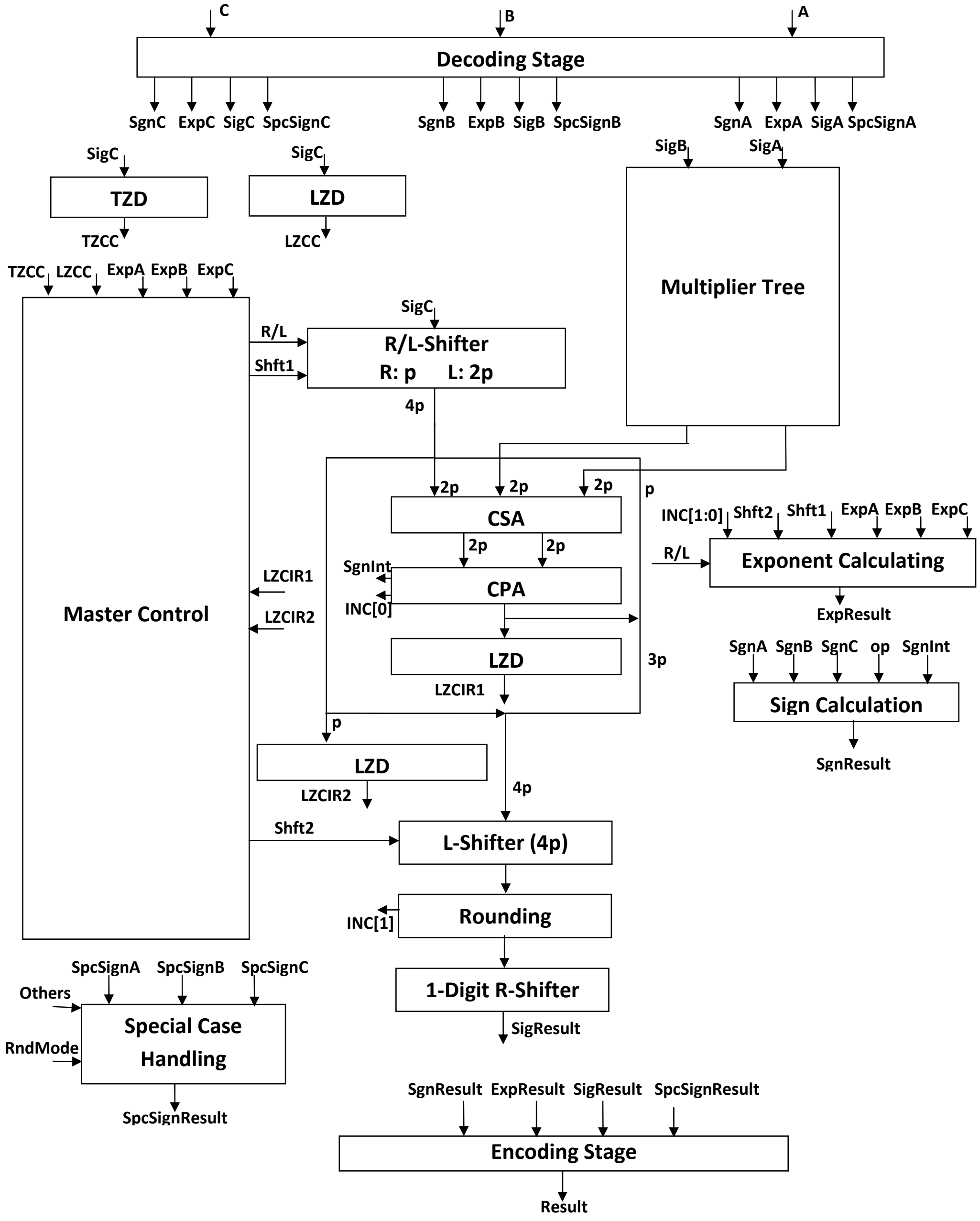


Figure 2.3: SillMinds' Architecture

The middle 2p digits of thr aligned addend is fed to the carry save adder (CSA) tree as one of the partial products. The final sum and carry vectors resulring from the reduction tree are added using a fast decimal carry propagation adder (CPA) to generate 2p intermediate result digits. The least and most p digits of the aligned addend are handled out of the CSA tree

Normalization:

The result may need a left shift to reach or to approach the preferred exponent. This requires to get the leading zero count (LZC) of the intermediate result, in order to know maximum allowed value of shift. To speed up the leading zero detection, two leading zero-digit counters are used after the CPA adder in parallel to get the LZC of the intermediate result. One of them counts the zero-digits in the most significant 2p digits of the result and the other counts zeros in the middle 2p digits. The final LZC is either the LZC of the most significant part or the sum of the two LZCs if all the most significant part is zeros.

Rounding:

The Rounder unit depends on the rounding mode and the intermediate sign to round the result. It takes p+1 digits of the shifted-unrounded result and the sticky bit. If a rounding up decision is made, the result is incremented by one, else the intermediate result is truncated selecting the shifted-unrounded FMA result truncated to p digits. It supports the five rounding directions listed in the IEEE Std 754-2008 as well as two more rounding directions supported in the Java BigDecimal class [25].

Master Control:

In this unit, the control signals that determines the amount of the addend shift and the amound of the intermediate result shift are calculated. Hence, it controls the shifters in the architecture.

Design Disadvantages:

Although the design claims conformance to the standard, it does not explain how the worst case of carry propagation over 3p digits is handled while the CPA of only 2p width. The design also does not clearly indicate how a correct underflowed result can be obtained because this case must be considered in the final shift of the result. Moreover, the flags are not correctly set. Since, according to [39], the underflow flag is set in case of underflow exception even if the result is exact which is not the case in decimal. On the other hand, the inexact flag is raised in case of underflow which is not necessarily correct.

This is besides that the design is not completely parallel and uses a leading zero detection in the critical path and a successive additions operations for both initial addition and rounding.

From this brief survey on both binary and decimal implementations for FMA operation. We can conclude that, there are much more reseach performed in binary FMAs compared to decimal. This led to higher performance architectures that use more parallelism than found in decimal. For example, Bruguera's binary implementation uses a combined add/round module with a leading zero anticipator to speed up the operation [60]. This is not explored in decimal yet. Also, there are reseach products that explores performing addition and multiplication with a variable latencies on a binary fused multiply-add module [64]; which is not also explored in decimal. In brief, there are still many improvements and extensions that can be performed on decimal FMA architectures. In thesis, we explore a part of this.

This chapter explored the fused multiply add operation as defined in the standard and as surveyed its different implementations in both binary and decimal literature. In the following chapter, the building blocks of the FMA module are discussed in more details.

Example-1		
Operation		$9999999999999999E369 \times 1E1 - 9999999999999999E369RPI$
Exact Multiplication Result		$+9999999999999999E370$
Multiply Then Add	Rounded Multiplication	$+inf\,xo$
	Rounded Final Result	$+inf\,xo$
Rounded FMA Result		$+8999999999999991E369\,x$
Example-2		
Operation		$9999999999999999E369 \times 1E1 - inf\ \ RPI$
Exact Multiplication Result		$+9999999999999999E370$
Multiply Then Add	Rounded Multiplication	$+inf\,xo$
	Rounded Final Result	$+qNaN\,xo$
Rounded FMA Result		$-inf$
Example-3		
Operation		$-246913578024696E0 \times 49999999999999E-13 + 1234567890123456E0\ RZ$
Exact Multiplication Result		$-12345678901234553086421975304E-13$
Multiply Then Add	Rounded Multiplication	$-1234567890123455E0$
	Rounded Final Result	$+1E0\,x$
FMA Result		$+6913578024696E-13\,x$
Example-4		
Operation		$1234567891011120E0 \times 1234567891011120E0 - 670895963654400E0\,RZ$
Exact Multiplication Result		$+1524157877515644670895963654400E0$
Multiply Then Add	Rounded Multiplication	$+1524157877515644E15x$
	Rounded Final Result	$1524157877515643E15x$
FMA Result		$1524157877515644E15$

Table 2.1: Examples of FMA operations compared with successive multiplication and addition

Chapter 3

Fused Multiply-Add Building Blocks

This chapter gives an overview of some basic modules that will be used in our proposed FMA. It mainly focuses on three modules: the multiplier tree, the leading zero anticipator and the BCD addition and rounding. It gives a background of the functionality required from each module and a literature review. In our FMA, we select the design that gives higher performance (speed). However, there are some modifications performed to enable the module to fit in our FMA as a whole, achieving higher speed or removing unnecessary area. The selected design and its modifications will be discussed in details in the next chapter, however, in this chapter, only a brief overview is presented on the available design options.

3.1 Multiplier Tree

3.1.1 Background

Multiplication is implemented by the accumulation of partial products, each of which is generated via multiplying the whole multiplicand by a weighted digit of the multiplier. Multiplication may be accomplished sequentially

[71, 35] or in parallel [34, 33]. However, in decimal multiplication, there is an increased complexity, compared to binary, in both the generation and reduction of partial products. Since there are several multiples of the multiplicand (from 0 to 9), there is a considerable complexity in the generation of the partial products. Also, the reduction must be complex enough to handle carries across digit boundaries in decimal and the implied correction steps.

Since we target a high performance design, we will focus only on fully parallel BCD multiplication, which is generally composed from three-steps; briefly described below:

1. Partial Product Generation (PPG): A partial product $P^i = Y^i \times X$, as in Figure 3.1, is generated via multiplying the multiplicand X by a multiplier digit Y^i, where the positional weight of P^i and Y^i is equal to the ith power of the radix (e.g., 10^i).

2. Partial Product Reduction (PPR): This is the major step in parallel multiplication, which is a special case of the more general problem of multioperand decimal addition [72]. A parallel solution for this problem may use a reduction tree that leads to a redundant intermediate representation of product (i.e. two vectors of BCD numbers).

3. Final Product Computation: The redundant product computed in Step 2 is converted to the final BCD product. This is regularly performed using any type of carry propagation adders.

Figure 3.1 illustrates Steps 1, 2, and 3 for a k-digit BCD multiplication, where uppercase letters denote decimal digits, superscripts indicate the decimal weighted position of the digits, and subscripts are meant to distinguish the relative decimal weights of partial products.

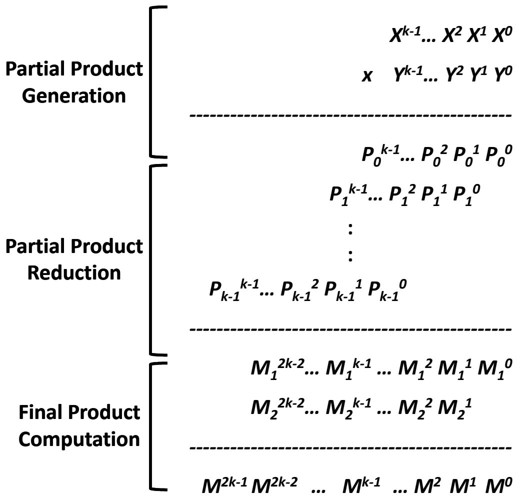

Partial Product Generation
$X^{k-1} \ldots X^2 X^1 X^0$
x $Y^{k-1} \ldots Y^2 Y^1 Y^0$
$P_0^{k-1} \ldots P_0^2 P_0^1 P_0^0$
Partial Product Reduction
$P_1^{k-1} \ldots P_1^2 P_1^1 P_1^0$
$P_{k-1}^{k-1} \ldots P_{k-1}^2 P_{k-1}^1 P_{k-1}^0$
Final Product Computation
$M_1^{2k-2} \ldots M_1^{k-1} \ldots M_1^2 M_1^1 M_1^0$
$M_2^{2k-2} \ldots M_2^{k-1} \ldots M_2^2 M_2^1$
$M^{2k-1} M^{2k-2} \ldots M^{k-1} \ldots M^2 M^1 M^0$

Figure 3.1: Decimal Fixed Point Multiplication Steps

3.1.2 Literature Review

As stated previously, some decimal multipliers use sequential techniques to accumulate partial products, others add all the partial products in parallel using and result in a redundant representation then convert the result back to non-redundant BCD representation. In this subsection, we survey the parallel proposals for multiplication found in the literature so far.

Lang and Nannarelli [73] present the first fully parallel decimal multiplier, in which X is the multiplicand and Y is the multiplier. In their design, they recode each digit of the multiplier as $Y^i = Y^{Ui} + Y^{Li}$ such that $Y^{Ui} \in \{0, 5, 10\}$ and $Y^{Li} \in \{-2, -1, 0, 1, 2\}$. Each partial product P^i is converted into carry-save format, such that $P^i = P^{Ui} + P^{Li}$. For decimal64 operands, this process generates 16 P^{Ui} terms and 16 P^{Li} terms. By using 17 radix-10 CSAs and two carry counters, the partial products are accumulated into a 32-digit sum-and-carry pair. These values are then fed into a simplified decimal carry-propagate adder to generate a 32-digit result.

Lang and Nannarelli present the first fully parallel decimal multiplier with recoding of only multiplier digits. Only $2X$ and $5X$ are formed for partial product generation; and a mixture of radix-10 CSAs and carry counters to improve the accumulation of partial products.

Vazquez et al. [33] present a new family of parallel decimal multipliers. To efficiently generate partial products, two recoding methods for multiplier operands are proposed. The radix-10 recoding converts the digit set [0, . . . , 9] into the SD set [-5, . . . ,5]. Each digit in the radix-5 recoding is represented as $Y^i = 5 \times Y^{Ui} + Y^{Li}$, where $Y^{Ui} \in \{0,1,2\}$ and$Y^{Li} \in \{-2,\cdots,2\}$. The radix-5 recoding gives simpler logic to generate partial products than the radix-10 but with more partial products to reduce. Furthermore, redundant BCD format is used to encode partial products in order to simplify both the generation and the reduction steps of the partial products. The partial product accumulation is done via a decimal CSA tree. The final carry-propagate addition is performed by a quaternary-tree decimal adder to generate the final 32-digit decimal product. The quaternary adder uses a full binary parallel prefix carry network to obtain all the binary carries. Since decimal and binary carries are equal when the operands are pre-corrected, the speculative sum digits can be obtained directly from the XOR operation of input operands bits and binary carries. A post–correction scheme is necessary after carry evaluation to correct the speculative sum digits when they are wrong. The sum digits are pre–evaluated in a parallel pre–sum stage of 4–bit carry–select adders. The decimal carries computed in the quaternary–tree select the right pre-sum. The correction of wrong speculative sum digits is done in the pre–sum stage for each of the two possible values of the input decimal carry, in parallel with carry computation [?] (conditional speculative).

Castellanos and Stine [74] compare the approaches of decimal partial product generation presented by Lang and Nannarelli and Vazquez et al. and introduce two new hybrid approaches. The hybrid approaches involve implementing the Boolean expressions for the multiplicand double and quintuple, instead of the BCD recoding schemes in the case of the architecture from Vazquez et al. [33], and performing the converse for the case of the architecture from Lang and Nannarelli [73]. The authors show that the hybrid approaches yield faster and smaller partial product generation than their original counterparts.

Jaberipur and Kaivani [34] propose an improved partial product generation approach involving the following multiplicand multiple set $X, 2X, 5X, 8X, 9X$, where $8X$ and $9X$ are compsed of 2 components that are easily generated. This approach allows for fast multiple set generation at the expense of doubling the number of multiples to be compressed in the partial product reduction step. To improve the delay of partial product reduction, they describe how to combine the reduction technique by Lang and Nannarelli [73] with a modified decimal full adder. The authors also adapt the adder developed to use a Kogge–Stone tree. The authors compare their implementation to those of Lang and Nannarelli [73] and Vazquez et al. [33] and show a notable improvement in delay, yet at the expense of area.

3.2 Leading Zero Anticipator

3.2.1 Background

A leading zero detector (LZD) is a basic unit in FP addition and FMA either in binary or decimal. Since the intermediate result has to be normalized to save the precision or to meet the decimal preferred exponent, it is necessary to detect the number of leading zeros in the intermediate result. Hence, a LZD waits for the result of the adder/subtractor to count the number of

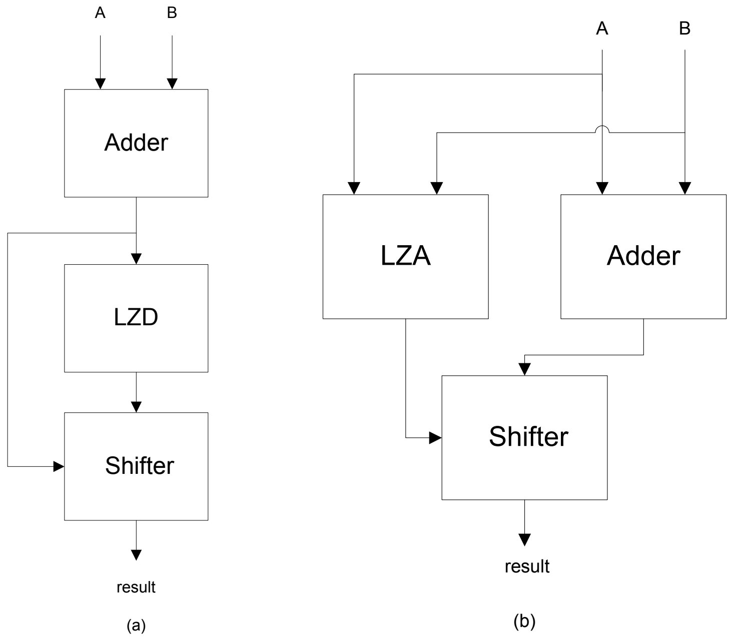


Figure 3.2: (a) Leading Zero Detection (b) Leading Zero Anticipation

leading zeros (LZC), then this count is used to shift the result to the left by an appropriate value. However, this adds the delay of the LZD to the critical path. As a solution, the LZD may be replaced by a leading zero anticipator (LZA) that anticipates the LZC in parallel to the addition. The top-level of the two architectures is shown in Figure 3.2.

The LZA anticipates the number of the leading zeros in the result directly from the input operands. Most of the LZA designs in binary and decimal consist mainly of two blocks, the preliminary anticipation block and the correction block that corrects the error of one bit/digit in the preliminary anticipation. The bulk of the research in the LZAs is in the correction circuitry. This is mainly to avoid more overhead in the critical path or to reduce the area of this circuitry.

3.2.2 Literature Review

Preliminary Anticipation Step

In binary, there is no need to anticipate leading zeros in case of effective addition. Since binary operands are already normalized. However, it is not the case in decimal. For effective subtraction, both binary [43, 75] and decimal LZAs [30] perform the preliminary anticipation on three steps.

First, the two operands are encoded digitwisely to produce a string of signals 'W'. In binary, $w_i = a_i - b_i$ where w_i is the signal corresponding to bit 'i' in the string W; and a_i ,b_i are the bits number 'i' in the operand A or B respectively, where it is required to calculate A-B. However, in case of decimal, more signals are generated to cover the larger digit set in decimal. The decimal case will be explained in details in the next chapter.

Second, the string 'W' generated from the last step is encoded again taking into consideration the different patterns of cancellation into a new binary string 'P' with leading zero bit count equals to the leading zero count supposed to be in the intermediate result. However, in order to simplify the logic of generating the string 'P', it is generated based only on each two successive digits. This leaves some patterns with a wrong anticipation of one digit.

Finally, the number of the leading zero bits in the string 'P' is detected using a leading zero detector.

In case of decimal effective addition, the proposal in [30] assumes the preliminary anticipation of the leading zero count in calculating (A+B) as Min{LZCA,LZCB}, where LZCA and LZCB are the leading zero counts in A, B respectively. However, this anticipation is also of one digit error due to a possible carry out when adding the two operands.

The decimal preliminary leading zero anticipation is dicussed in detail in the next chapter with a numerical example to give more explanation.

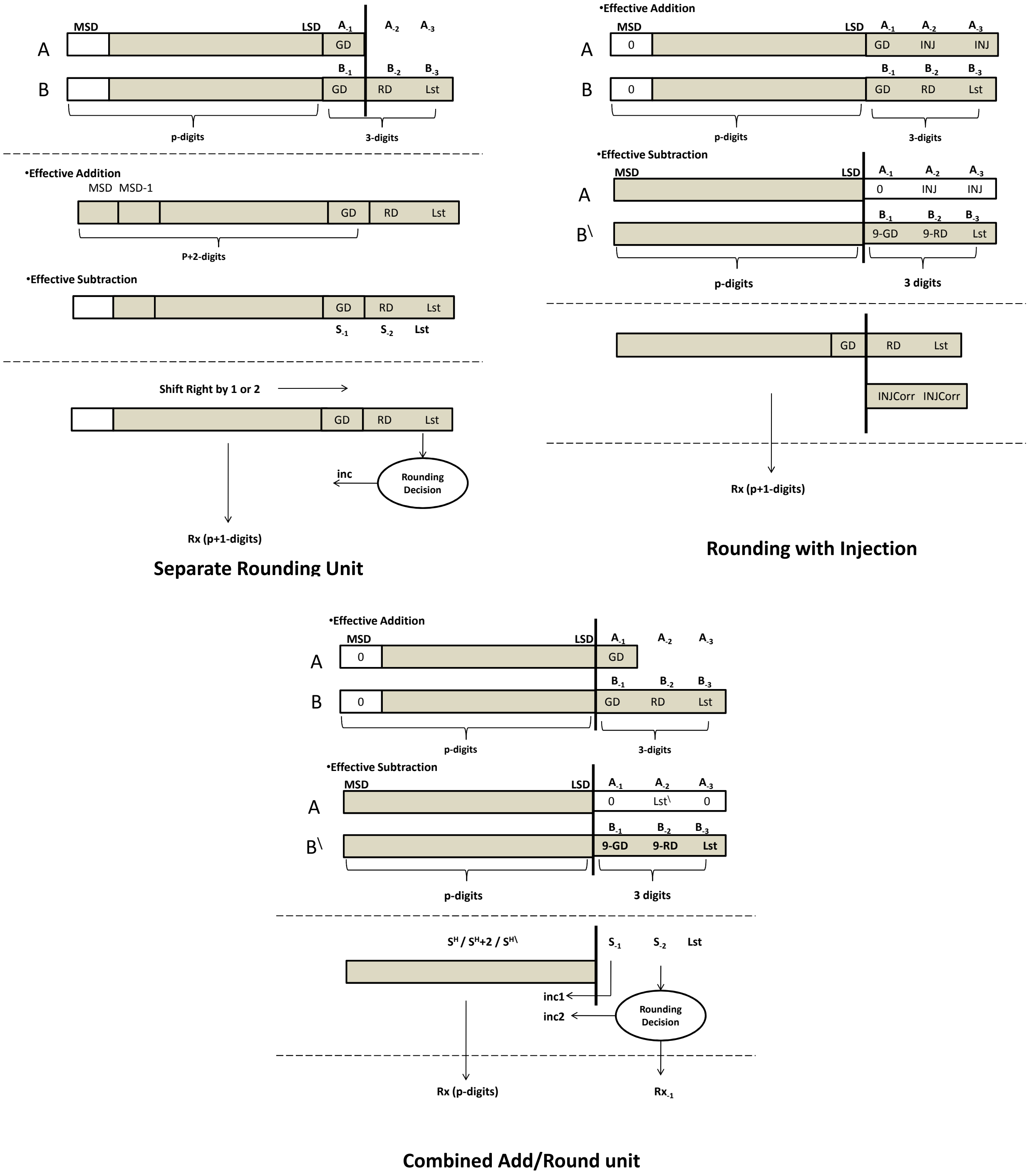
Final Correction Step

Amin et al. survey five different architectures for binary correction circuitry and one for decimal. Then, three different proposals are suggested for decimal. As stated, these proposals intend to minimize the cost of the correction circuitry on either the delay or the area. As we do not use a correction circuitry in our FMA, then it is not discussed here. However, the reader can refer to [?] (Asilomar) for more details.

3.3 Significand BCD Addition and Rounding

3.3.1 Background

The BCD sign-magnitude addition is a fundamental step in the floating-point addition, multiplication or fused-multiply add. The result of the addition has to be rounded to fit in the required precision. Also, the final result must be in a correct sign-magnitude representation (i.e. negative result has to be complemented). There are different proposals to perform these steps. However, we are interested only in the high performance implementations. Hence, we will briefly survey three high performance proposals: a separate rounding stage, rounding by injection and combined add/round. In the next subsection, we assume A and B are the two operands to be added.



Separate Rounding Unit

Rounding with Injection

Combined Add/Round unit

Figure 3.3: Different Algorithms for Significand BCD Addition and Rounding

3.3.2 Literature Review

Separate Rounding Stage

This technique is used in the first published DFP adder [28]. First, the inputs A and B are aligned as shown in Figure 3.3.Then, the sign-magnitude BCD addition is performed using a single compound adder that results in both (S , LS) where

$$S = \begin{cases} A+B & eop = 0 \\ A+\overline{B} & eop = 1 \end{cases} \; where\ \overline{B}\ is\ nine's\ complement\ of\ B \tag{3.1}$$

and

$$LS = S+1 \tag{3.2}$$

Rounding requires an additional +1 ulp increment of the BCD magnitude to the sum. It uses a separate decimal rounding unit. This sign-magnitude BCD adder uses a binary (compound) flagged prefix adder to compute the BCD sums S andLS.

In case of effective addition, the required result is the sum of two operands (i.e. A+B), hence, the final result is S. However, in case of effective subtracion, the required result is ($|A-B|$). If A is larger than or equals to B, then the intermediate result ($A-B = A+\overline{B}+1$) is positive. Therefore, the final result in this case is selected as $LS = A+\overline{B}+1$. On the other hand, in case of $B > A$, then the intermediate result ($A-B = A+\overline{B}+1$) is negative and requires another 10's complement. In other words, the final result should be $(\overline{A+\overline{B}+1}+1 = \overline{(A+\overline{B})} = \overline{S}$.

In order to know the larger operand in both A and B, the carry out ($Cout$) to the most significant digit (MSD) is used. If the $eop = 1$ and $Cout = 1$, then the intermediate result is positive and $A \geq B$. Otherwise, if $eop = 0$ and $Cout = 0$, then the intermediate result is negative and $A < B$. Therefore, the $MSD = Cout\,.\,\overline{eop}$, since the carry out in case of effective subtraction indicates only to a negative result.

Once the addition is complete, the result may need to be shifted two positions to the right if $MSD = 1$ or one position to the right if $MSD - 1 \neq 0$. The decimal rounding unit is placed after the significant BCD addition and the R1/R2 shifter. Thus, it receives the BCD p-digit truncated and shifted (SX) with the round digit (S_{RD}) and the sticky bit (S_{st}). It uses a straightforward implementation of IEEE 75-2008 rounding. This involves detecting a condition for each rounding mode in order to increment SX or not by one ulp. Each increment condition (inc) is determined from the rounding digit, the sticky bit, the sign of the result and SX_{lsb}, the least significant bit SX. The $+1$ ulp increment can generate a new carry propagate from the LSD (least significant digit) up to the MSD of SX. This carry propagation is determined examining the number of consecutive 9's starting from the LSD of SX. A prefix tree of AND gates detects this chain. A vector of muxes selects the correct sum digit at each position (SX_i or $SX_i + 1$) the selection line for each mux is generated from the AND gates prefix tree.

Rounding with Injection

This method was proposed for binary [61] and extended to decimal [29]. It is based on adding an injection value (INJ) to the input operands that reduces all the rounding modes to round towards zero (truncated by the round digit position). However, when the truncated result has more significant digits than the format precision p, it is necessary to inject a correction amount (INJcorr) to obtain the correctly rounded result.

The layout of the inputs A and B is shown in Figure 3.3. The decimal injection values INJ are inserted in the round and sticky positions of A such that:

$$round_{mode}(X) = round_{RZ}(X + INJ)|_{within\, the\, required\, precision} \tag{3.3}$$

If the MSD of the sum is not zero (i.e. result has more significant digits that the required precision), an injection correction value INJcorr is used to compute the correctly rounded result R such that:

$$round_{mode}(X) = round_{RZ}(X + INJ + INJcorr) \tag{3.4}$$

The INJcorr value is added using a 2-digit BCD adder. The increment signal inc is true if carry-out is generated to the least significant digit position. This correction can generate a decimal carry propagation, determined by the trailing chain of 9's of S. To reduce the critical path delay of the unit, the trailing 9's detection is performed examining the uncorrected binary sum digits and the decimal carries obtained before the BCD correction in the sign magnitude BCD adder. This hides the logarithmic delay of the trailing 9's detection. Hence, the carries produced from the trailing 9's detection selects digitwisely the final rounded result.

Combined Add/Round

This technique was proposed by Vazquez and Antelo in [76] based on a previous binary version proposed by [60]. The aligned BCD input operands A and B (including the guard, round and sticky digits) are aligned as shown in Figure 3.3. To incorporate rounding into the significant BCD addition, they perform the computation in two parts. First, they split BCD operands A and B in a p-digit most significant part (A^H, B^H) and a least significant one, which includes the digits/bits placed at the guard (A_{-1}, B_{-1}), round (B_{-2}) and sticky (B_{st}) positions. The p-digit operands S^H ($\bar{S}^H$), $S^H + 1$ and $S^H + 2$ are computed speculatively, with:

$$S = \begin{cases} A^H + B^H & eop = 0 \\ A^H + \overline{B}^H & eop = 1 \end{cases} \quad where\ \overline{B}\ is\ nine's\ complement\ of\ B\,. \quad (3.5)$$

Two signals, inc1 and inc2 represent the carries into the least significant digit of S^H due to the addition of the least significant part and the rounding. Signal inc1 is a decimal carry into the least significant digit of S^H due to the addition of the 3 least significant digits of A and B. Also rounding generates another decimal carry inc2 into the least significant of S^H. The values of inc1, inc2, the intermediate sign of the result and the most significant digit of the result before rounding selects the appropriate result for both the most significant part ($S^H, S^H + 2, \overline{S}^H$) and also the correct value of the least significant digit.

A modified version of the combined add/round module will be discussed in more details in the next chapter.

In this chapter we introduced the most important basic building blocks in any FMA design with a literature review on each block. In the next chapter, we will explain in detail our proposed architecture with its implementation details. In our proposed FMA, we select, modify and use some of the ideas presented in this chapter to build our basic modules.

Chapter 4

Proposed Fused Multiply Add

As stated in Chapter 2, the IEEE 754-2008 standard specifies the requirements of the decimal fused multiply-add (FMA) operation which computes A×B±C with only one rounding step.

This Chapter introduces the basic architecture of a 64-bit fused multiply-add module. The design explores the potential of using a leading zero anticipator and a combined add/round module to speed up the FMA operation. Hence, this chapter starts with a quick overview of this architecture. Then the detailed datapath for both the significand and the exponent is discussed. Some exceptional cases are handled apart from the default datapath. These cases are discussed at the end of this chapter.

4.1 Quick Overview

The top level of this architecture is shown in Figure 4.1. As shown in the figure, the architecture can be divided into three main stages. The first stage is the multiplier tree which produces the multiplication result. Meanwhile, the addend is prepared for addition, in parallel to the multiplier tree. This eliminates the need for further processing on the multiplication result which in turn reduces the critical path delay. The second stage is the leading zero anticipator which is important to align the operands preparing them for the third stage which is the combined add/round unit. The three stages is shown is Figure 4.2 which represents in brief the default datapath in the architecture.

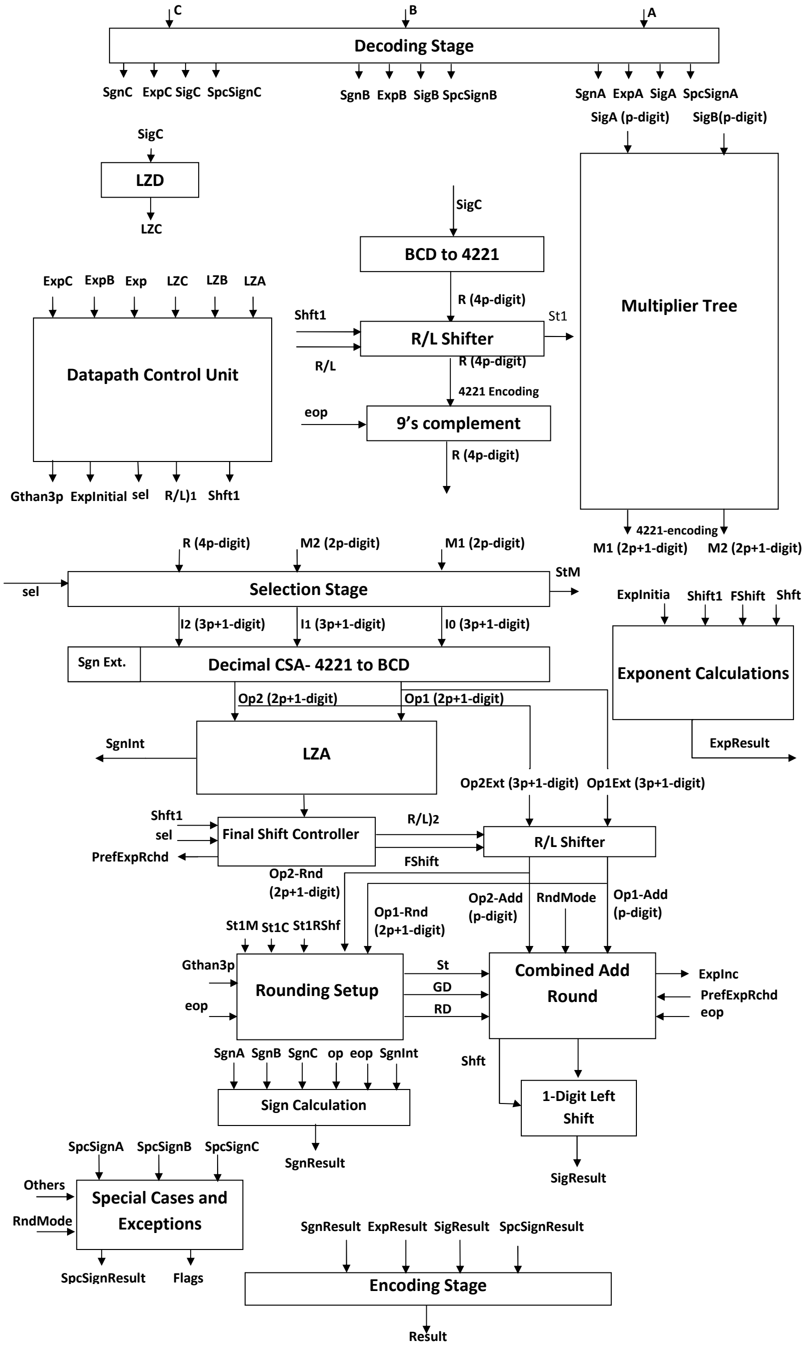
The default alignment of the addend is such that the fractional point is to the right of the multiplication result. While the multiplier tree produces the multiplication result, the addend is aligned by shifting it to the right (case-1) or to the left (cases-2,3,4). From a 4p width, where p is the number of significand digits (16 digits in case of 64-bits format), it is required to anticipate the leading zeros in only 2p-digits. According to the exponent difference and the leading zeros of the addend, the appropriate 2p width is selected. In next sections, more details are discussed for these different cases.

A leading zero anticipator anticipates the leading zeros in the result supposed to outcome from the selected width of the two operands. Based on the anticipated leading zero count and taking into consideration the preferred exponent, a final shift amount is determined for the two operands. The rounding position in the aligned operands is approximately known (with an error of one digit). Hence, a combined add/round module can be used to get the final result instead of successive addition and rounding steps.

This overview ignores some details and leaves others without explanation. However, it is a good start point to understand the architecture. In the following, the default datapath is explored in detail.

4.2 Default Significand Datapath

As stated in Chapter 1, the IEEE 754-2008 standard specifies formats for both binary floating-point (BFP) and decimal floating-point (DFP) numbers



A, B	multiplier operands
C	Addend
SgnX	sign of operand X
ExpX	exponent of operand X
SigX	significant of operand X
SpcSignX	special signals of X (includes NaN, Infinity, Zero)
LZX	leading zero count of X
op	Operation
StX	Contribution of signal X to the sticky.
CSA	carry save adder
R/L	control signal determines a right or left shift
SgnInt	Intermediate Result Sign
Shft1,2	right of Left shift amount
RndMode	rounding mode
ExpInitial	intermediate exponent
ExpInc	exponent increment
sel	control signal selects a narrower width to operate on
LZA	leading zero anticipator
eop	effective operation
GD,RD	Guard and Round digits
BCD	Binary coded decimal (8421 weights)
Others	Other control signals from other modules
Gthan3p	Signaled if shifting to left > 3p+1

Figure 4.1: Top Level Architecture

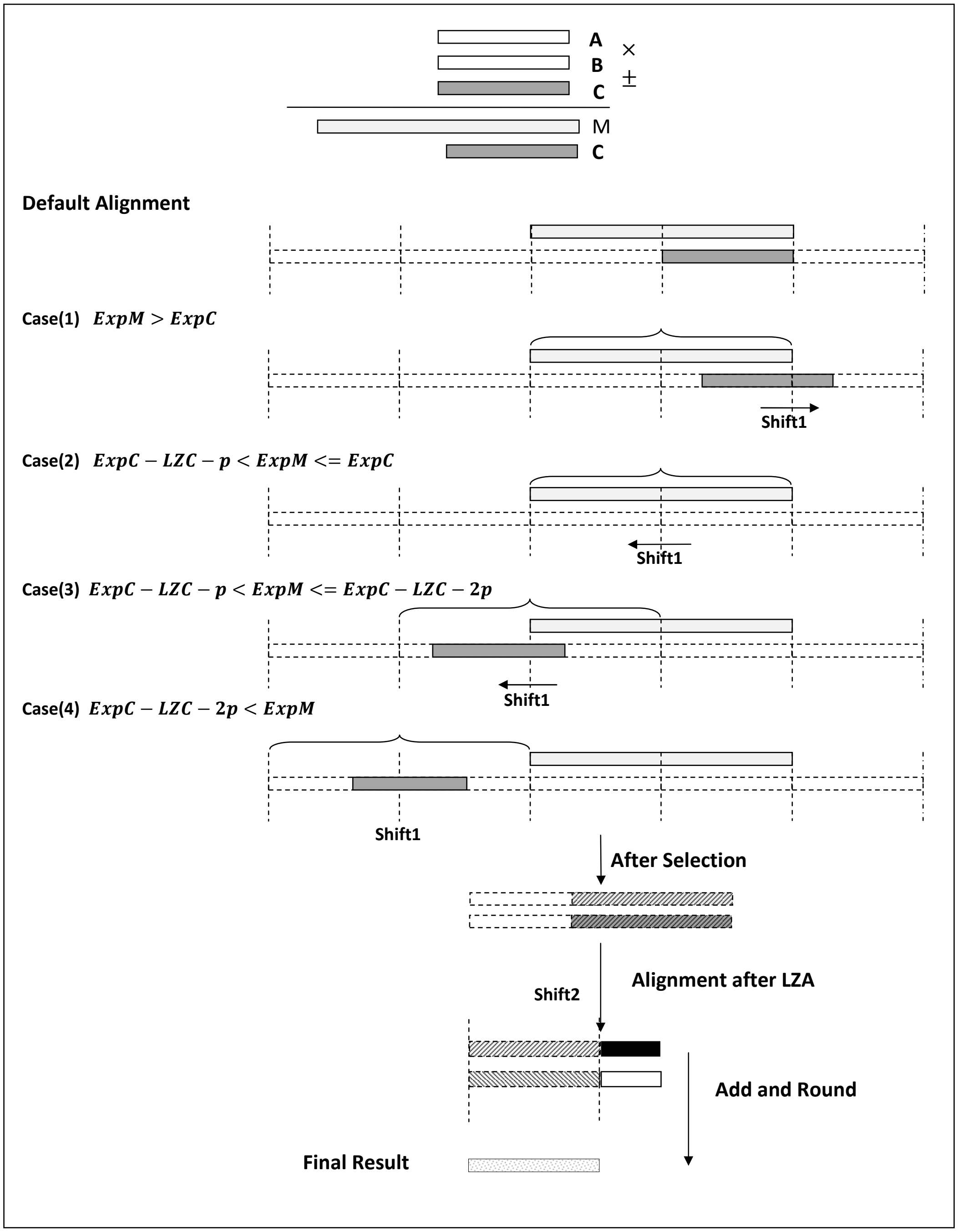


Figure 4.2: Default Significand Datapath

[18]. An IEEE 754-2008 DFP number contains a sign bit, an integer significand with a precision of p digits, and a biased exponent. The value of a finite DFP number is:

$$D = (-1)^S \times C \times 10^{E-bias} \tag{4.1}$$

where S is the sign bit, C is the non-negative integer significand, and E is the biased non-negative integer exponent. The significand can be encoded either in Binary Integer Decimal (BID) format or in Densely Packed Decimal (DPD) format; which is referred to in the standard as the binary and decimal encodings respectively. In the BID encoding, the number is encoded as a whole in the significant bits. However, the DPD encoding encodes each three digits of the significand in 10-bits with the most significant digit encoded separately. The exponent must be in the range [Emin,Emax] biased with a certain bias value. Representations for infinity and Not-a-Number (NaN) are also provided.

Since the fused multiply-add design presented here uses 64-bit DFP numbers. This format has p = 16 decimal digits of precision in the significand, an unbiased exponent range of [−383, 384], and a bias of 398.

The encoding used in the significand format is the DPD encoding rather than the BID encoding; since the former encodes each three digits in 10-bits, hence the digit boundries are easier to detect; then it is faster to decode the number to BCD which is much easier for the addition operation.

Therefore, the 64-bits are encoded as follws: the MSB represents the sign of the number (S). S=0 for a positive number and S=1 for a negative one. The least significant 50 bits called the trailing (T) encodes the least significant 15 digits in DPD format (i.e. each 10-bits represents three decimal digits). Finally, the combination field (G) follows the sign directly to encode both the exponent and the most significant digit (MSD) in the

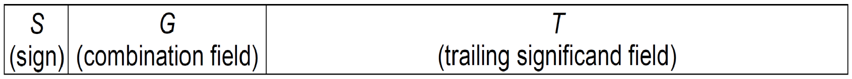


Figure 4.3: Decimal Interchange Format

significand. Also, it is used to encode special signals such as NaNs and Infinities.

4.2.1 Decoding the Operands

The operation begins with reading the three operands in the IEEE 754-2008 format and decoding each one to produce the sign bit, significand, exponent, and flags for special values of Not-a-Number (NaN) or infinity. The significands of the three operands are then decoded from the DPD encoding to Binary Coded Decimal (BCD).

As shown in Figure 4.3, the DFP number is decoded such that its sign is the MSB. The 13-bit combination field ,in case of 64-bit format, is decoded according to Table 4.2.1 to get the MSD in the DFP Number and the exponent and also to determine if the number is an infinity, quite NaN or Signaling NaN. The other 15 digits of the DFP number are decoded from the trailing part. Each successive 10-bits are decoded according to Table 4.2.1 to get three digits from the significand.

4.2.2 Multiplication

The decoded significands of both the multiplier and the multiplicand go directly to the multiplier tree to produce the multiplication result. The multiplier tree is very critical to both the speed and the area of the fused multiply-add unit. Hence, it should be fast for high performance architectures, however, its complexity should also be taken into account.

Combinational Filed (G) $G_{12}G_{11}\cdots\cdots G_7$		Special Case or Exponent and MSD	
$0xxxxx$		$MSD = 0G_5G_6G_7$,$E_{biased} = G_{12}G_{11}G_6G_5\cdots G_0$	
$10xxxx$		$MSD = 0G_5G_6G_7$,$E_{biased} = G_{12}G_{11}G_6G_5\cdots G_0$	
$110xxx$		$MSD = 100G_7$,$E_{biased} = G_9G_8G_6G_5\cdots G_0$	
$1110xx$		$MSD = 100G_7$,$E_{biased} = G_9G_8G_6G_5\cdots G_0$	
$1111xx$	$11110x$	$MSD = 0000$ $E_{biased} = 00\cdots 0$	Infinity
	111110		Signaling NaN
	111111		Quiet NaN

Table 4.1: Decoding each DPD to BCD

Declet $(I_9\cdots\cdots I_0)$ $(I_3I_2I_1I_6I_5)$	BCD $\{O^2\}\{O^1\}\{O^0\}$ $\{O_3^2O_2^2O_1^2O_0^2O_3^1O_2^1O_1^1O_0^1O_3^0O_2^0O_1^0O_0^0$
0xxxx	$\{0,I_9,I_8,I_7\}\{0,I_6,I_5,I_4\}\{1,I_2,I_1,I_0\}$
100xx	$\{0,I_9,I_8,I_7\}\{0,I_6,I_5,I_4\}\{1,0,0,I_0\}$
101xx	$\{0,I_9,I_8,I_7\}\{1,0,0,I_4\}\{1,I_2,I_1,I_0\}$
110xx	$\{1,0,0,I_7\}\{0,I_6,I_5,I_4\}\{0,I_9,I_8,I_0\}$
11100	$\{1,0,0,I_7\}\{1,0,0,I_4\}\{0,I_9,I_8,I_0\}$
11101	$\{1,0,0,I_7\}\{0,I_9,I_8,I_4\}\{1,0,0,I_0\}$
11110	$\{0,I_9,I_8,I_7\}\{1,0,0,I_4\}\{1,0,0,I_0\}$
11111	$\{1,0,0,I_7\}\{1,0,0,I_4\}\{1,0,0,I_0\}$

Table 4.2: Decoding each DPD to BCD

The radix-5 implementation suggested in [38] is a very good option for a high performance design. Moreover, it has a property that can be used to simplify the rest of the design; that is the sum vector resulting from the reduction tree is always negative. The importance of this property will be discussed later in Section 4.2.5. Other proposals discussed in the previous chapter are either of lower performance, or comparable performance with larger complexity.

The multiplier tree used in our FMA design is essentially the same as that presented in [38]. We present it in some detail for completeness and becasue it is the basis for next stages operation of the FMA.

4.2.2.1 SD-Radix 5 Architecture

Figure 4.4 shows the block diagram of the multiplier tree that uses SD-Radix 5 recoding. The multiplier consists of the following stages: generation of decimal partial products coded in (4221/5211) and reduction of partial products into two final vectors (M_1 and M_2). Coding the partial products in (4221/5211) representation facilitates the generation of the multiplicand multiples and also the reduction of the partial products which would be explained later.

Each digit in the multiplier (Y) is recoded from regular BCD format where$Y_i \in \{0, \cdots, 9\}$ into an SD radix-5 $Y_i = 5 \times Y_i^U + Y_i^L$, where $Y_i^U \in \{0, 1, 2\}$ and$Y_i^L \in \{-2, \cdots, 2\}$. This results in a 2p digit recoded multiplier (p-digits Y^U and p-digits Y^L). Then each recoded digit in the multiplier controls two MUXs; the first one selects a positive multiplicand multiple out of $\{0, X, 2X\}$ coded in (4221), while the other selects a mutiple out of $\{0, 5X, 10X\}$ coded in (5211). For a negative multiplicand multiple $\{-X \text{ or } -2X\}$; the corresponding positive multiple is inverted. This inversion is equivelent to 9's complement when the digits are coded in 4221

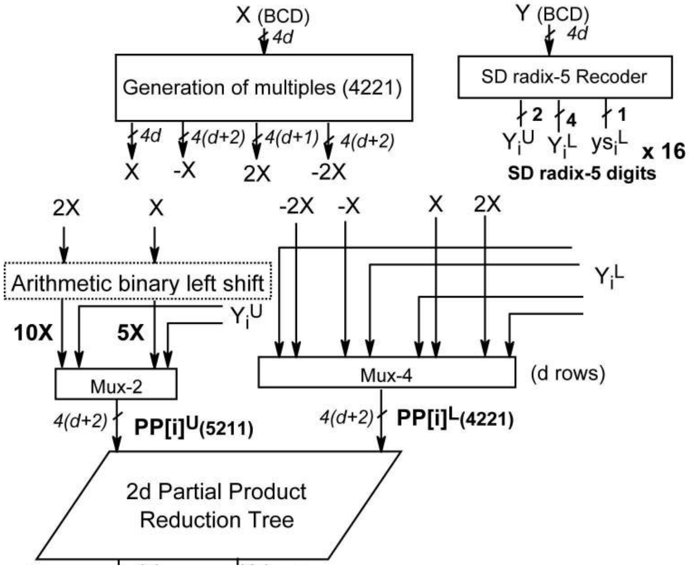


Figure 4.4: Multiplier Architecture for SD-Radix 5 Encoding
[38]

format. A (+1) is added to each negated partial product to get the 10's complement. This (+1) digit is inserted within the partial products array without extra overhead delay.

Before being reduced, the 2p partial products are aligned according to their decimal weights. Each p-digit column of the partial product array is reduced to two (4221) decimal digits using one of the decimal digit 2p:2 CSA trees.

4.2.2.2 Partial Product Generation

Multiplier Recoding As explained earlier, each BCD digit in the multiplier $Y_i \in \{0,\cdots,9\}$ is recoded into an SD radix-5 digit such that$Y_i = 5 \times Y_i^U + Y_i^L$, where $Y_i^U \in \{0,1,2\}$ and$Y_i^L \in \{-2,\cdots,2\}$. In order to select the correct partial products (multiplicand multiple), each digit Y_i^U is represented as two signals $\{y1_i^U, y2_i^U\}$. Also each digit Y_i^L is represented as four signals $\{y(+2)_i^L, y(+1)_i^L, y(-1)_i^L, y(-2)_i^L\}$ and a sign bit ys_i. By examining the different possibilities, these signals can be obtained directly from the BCD multiplier digits Y_i using the following logical expressions:

$$(Y_i^U) \quad \begin{cases} y2_i^U = y_{i,3}; \\ y1_i^U = y_{i,2} \,|\, (y_{i,1} \cdot y_{i,0}); \end{cases} \tag{4.2}$$

$$(Y_i^L) \quad \begin{cases} y(+2)_i^L = y_{i,1} \cdot ((y_{i,2} \cdot y_{i,0}) \,|\, (\overline{y_{i,2}} \cdot \overline{y_{i,0}})); \\ y(+1)_i^L = (\overline{y_{i,3}} \cdot \overline{y_{i,2}} \cdot \overline{y_{i,1}} \cdot y_{i,0}) \,|\, (y_{i,2} \cdot y_{i,1} \cdot \overline{y_{i,0}}); \\ y(-1)_i^L = (y_{i,3} \cdot y_{i,0}) \,|\, (y_{i,2} \cdot \overline{y_{i,1}} \cdot \overline{y_{i,0}}); \\ y(-2)_i^L = (y_{i,3} \cdot \overline{y_{i,0}}) \,|\, (\overline{y_{i,2}} \cdot y_{i,1} \cdot y_{i,0}); \\ ys_i = y(-2)_i^L \,|\, y(-1)_i^L; \end{cases} \tag{4.3}$$

In the following, we will present an example of how the multiplier is recoded.

Example:

Given Multiplicand (Y_i)	6	2	3	7	8	9	1	0	5	2	3	1	1	5	0	1
Recoded Multiplicand (Y_i^U)	5	0	5	5	10	10	0	0	5	0	5	0	0	5	0	0
Recoded Multiplicand (Y_i^L)	1	2	-2	2	-2	-1	1	0	0	2	-2	1	1	0	0	1
Signal $y1_i^U$	1	0	1	1	0	0	0	0	1	0	1	0	0	1	0	0
Signal $y2_i^U$	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Signal $y(+2)_i^U$	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0
Signal $y(+1)_i^U$	1	0	0	0	0	0	1	0	0	0	0	1	1	0	0	1
Signal $y(-1)_i^U$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
Signal $y(-2)$	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0

Multiplicand Multiples Generation All the required decimal multiplicand multiples are obtained in a few levels of combinational logic using different digit recoders and performing different fixed m-bit left shifts (LmShift) in the bit-vector representation of operands. In the following, we will use various representations for BCD numbers with different weights, rather than the regular 8421 format. These different representations will simplify the generation of the multiplicand multiples and also the reduction of the partial products. So, we will firstly list some properties for these different codes that will be used later in this section.

Property 1)

Among all the possible decimal codes, there is a family of codes suitable for simple decimal carry-save addition. This family of decimal codings verifies that the sum of their weight bits is 9, that is,

$$\sum_{j=0}^{3} r_j = 9, \tag{4.4}$$

where r_j represents the weight of bit number j in the BCD digit.

This set of codes includes the (4221) and (5211) codes. Moreover, they are redundant codes, since two or more different 4-bit vectors may represent the same decimal digit.

These codes have an important property, that is all the sixteen 4-bit vectors represent a decimal digit ($X_i \in [0,9]$). Therefore, any Boolean function (AND, OR, XOR,. . .) operating over the 4-bit vector representation of two or more input digits produces a 4-bit vector that represents a valid decimal digit (input and output digits represented in the same code).

Property 2)

The 9's complement of a digit X_i can be obtained by inverting their bits (as a 1's complement) since

$$9-X_i=\sum_{j=0}^{3} r_j-\sum_{j=0}^{3} x_{i,j}r_j=\sum_{j=0}^{3}(1-xi,j)r_j=\sum_{j=0}^{3} r_j \qquad (4.5)$$

Hence, negative operands can be obtained by inverting the bits of the positive bit vector representation and adding a 1 ulp, as follows,

$$-X_{r_3r_2r_1r_0}=\overline{X_{r_3r_2r_1r_0}}+1 \qquad (4.6)$$

where $X_{r_3r_2r_3r_1}$ means the number X is encoded by BCD weights of $(r_3r_2r_1r_0)$.

Property 3)

If a decimal number X is encoded such that the 4-bits of each digit is weighted as (5421) then shifted left by '1' bit and and the result is read again as if weighted by (8421) for each digit, it is the same as multiplying the decimal number X by 2 in its original format of (8421). In other words:

$$(L_{1shift}\{X_{5421}\})_{8421}=(2\times X)_{8421} \qquad (4.7)$$

This can be simply proved as follows: (v_i^j bit number (i) in digit (j) in 5421 encoding)

$X_{5421}=(\cdots v_0^{i+1}v_3^iv_2^iv_1^iv_0^i\ v_3^{i-1}\cdots)_{5421}$

$=\cdots 10^{i+1}\times(\cdots+v_0^{i+1})+10^i\times(5v_3^i+4v_2^i+2v_1^i+v_0^i)+10^{i-1}\times(5v_3^{i-1}+\cdots)$

with one bit left shift:

$(L_{1shift}\{X_{5421}\})_{8421}=(\cdots v_0^{i+1}v_3^iv_2^iv_1^iv_0^iv_3^{i-1}\ v_2^{i-1}\cdots)_{8421}$

$= \cdots 10^{i+1} \times (\cdots + v_3^i) + 10^i \times (8v_2^i + 4v_1^i + 2v_0^i + v_3^{i-1}) + 10^{i-1} \times (5v_2^{i-1} + \cdots)$

$= 2 \times (\cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 4v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots))$

$= 2 \times (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \cdots)_{5421} = (2X)_{8421}$

This proves the initial claim.

Property 4)

If a decimal number X is encoded such that the 4-bits of each digit is weighted as (4221) then shifted left by '3' bits and the result is read again as if weighted by (5211) for each digit, it is the same as multiplying the decimal number X by 5 in the format of (4221). In other words:

$$(L_{3shift}\{X_{4221}\}p)_{5211} = (5 \times X)_{4221} \tag{4.8}$$

This can be simply proved as follows: (v_i^j bit number (i) in digit (j) in 5421 encoding)

X_{4221}= $(\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i\ v_3^{i-1} \cdots)_{4221}$

$= \cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (4v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (4v_0^{i-1} + \cdots)$

with three bits left shift:

$(L_{3shift}\{X_{4221}\})_{5211}= (\cdots v_1^i v_0^i v_3^{i-1} v_2^{i-1} v_1^{i-1}\ v_0^{i-1} \cdots)_{5211}$

$= \cdots 10^{i+1} \times (\cdots + v_1^i) + 10^i \times (5v_0^i + 2v_3^{i-1} + v_2^{i-1} + v_1^{i-1}) + 10^{i-1} \times (5v_0^{i-1} + \cdots)$

$= 5 \times (\cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (4v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (4v_0^{i-1} + \cdots))$

$= 5 \times (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \cdots)_{4221} = (5X)_{5211}$

This proves the initial claim.

Property 5)

If a decimal number X is encoded such that the 4-bits of each digit is weighted as (5211) then shifted left by '1' bit and the result is read again as if weighted by (4221) for each digit, it is the same as multiplying the decimal number X by 2 in the format of (4221).

$$(L_{1shift}\{X_{5211}\}p)_{4221} = (2 \times X)_{4221} \tag{4.9}$$

This can be simply proved as follows: (v_i^j bit number (i) in digit (j) in 5221 encoding)

$X_{5211}= (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i\ v_3^{i-1} \cdots)_{5211}$

$= \cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots)$

with one bit left shift:

Digit	8421	4221	5211	5421
0	0000	0000	0000	0000
1	0001	0001	0001	0001
2	0010	0010	0100	0010
3	0011	0011	0101	0011
4	0100	1000	0111	0100
5	0101	1001	1000	1000
6	0110	1010	1001	1001
7	0111	1011	1100	1010
8	1000	1110	1101	1011
9	1001	1111	1111	1100

Table 4.3: Decimal Codings

$(L_{1shift}\{X_{5211}\})_{4221}= (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1}\ v_2^{i-1} \cdots)_{8421}$

$= \cdots 10^{i+1} \times (\cdots + v_3^i) + 10^i \times (4v_2^i + 2v_1^i + 2v_0^i + v_3^{i-1}) + 10^{i-1} \times (4v_2^{i-1} + \cdots)$

$= 2 \times (\cdots 10^{i+1} \times (\cdots + v_0^{i+1}) + 10^i \times (5v_3^i + 2v_2^i + 2v_1^i + v_0^i) + 10^{i-1} \times (5v_3^{i-1} + \cdots))$

$= 2 \times (\cdots v_0^{i+1} v_3^i v_2^i v_1^i v_0^i v_3^{i-1} \cdots)_{5221} = (2X)_{5211}$

In order to use the previous characteristics of these different encodings, we should first define how the recoding from a format to another is done. Table 4.2.2.2lists the different decimal codings (8421, 5421,4221,5211). Although, the mapping is not unique, the selected mapping reduces logic complexity as much as possible.

In the following, we will summerize how the different multiples are generated in the 4221 format:

The X BCD multiplicand: is easily recoded to (4221) using the logical expressions.

Multiple 2X: Each BCD digit is first recoded to the (5421) decimal coding shown in Table 4.2.2.2. Using property (3) defined by Equation 4.7, an L1shift is performed to the recoded multiplicand, obtaining the 2X multiple in BCD. Then, the 2X BCD multiple is recoded to (4221).

Multiple 5X: It is obtained by a simple L3shift of the (4221) recoded multiplicand, with resultant digits coded in (5211).

Multiple 10X: It is obtained by a simple L3shift of the $2X$ (4221) recoded multiplicand multiples, with resultant digits coded in (5211).

Negative Multiples: For negative multiples (i.e. $ys_i = 1$), the positive multiple is inverted to get the 9's complement. For 10's complement, a (+1) is added at the least significant digit position. Since only the Y_i^L multiples may be negative, the (+1) is inserted in the least significant bit of the corresponding Y_i^U multiple.

4.2.2.3 Partial Product Array

As we detailed before, the SD radix-5 architecture produces 2p partial products coded in (4221/5211). Before being reduced, the 2p partial products P_i are aligned according to their decimal weights by 4i-bit wired left shifts ($PP_i \times 10^i$). The resultant partial product array for 16-digit input operands is shown in Figure 4.5. In this case, the number of digits to be reduced varies from 32 partial products to 2 partial products. For negative partial products we insert '+1' at the least significant bit of the least significant digit of the Y_i^U.

However, the part of the sign extension can be reduced off line (i.e. in design time). Since the digit S is either (S=0) for positive partial products or (S=9) for negative partial products, we can explore the different possibilities of adding two sign vectors and try to reduce them.

	Case(1)	Case(2)	Case(3)	Case(4)
$S_1 \cdots S_1 S_1$	$0 \cdots 00$	$0 \cdots 00$	$9 \cdots 99$	$9 \cdots 99$
$S_2 \cdots S_2 S_2$	$0 \cdots 00$	$9 \cdots 99$	$0 \cdots 00$	$9 \cdots 99$
$S_R \cdots S_R S_R$	$0 \cdots 00$	$9 \cdots 99$	$9 \cdots 99$	$9 \cdots 98$

; where S represents the sign digit.

Or equivelently

	Case(1)	Case(2)	Case(3)	Case(4)
$S_1 \cdots S_1 S_1$	$0 \cdots 01$	$0 \cdots 01$	$0 \cdots 00$	$0 \cdots 00$
$S_2 \cdots S_2 S_2$	$9 \cdots 99$	$9 \cdots 98$	$9 \cdots 99$	$9 \cdots 98$
$S_R \cdots S_R S_R$	$0 \cdots 00$	$9 \cdots 99$	$9 \cdots 99$	$9 \cdots 98$

All these cases can be represented as follows:

$$\begin{array}{|c|c|} \hline Q_1 \cdots Q_1 Q_1 & 0 \cdots 0(000\bar{s}_1) \\ Q_2 \cdots Q_2 Q_2 & 9 \cdots 9(111\bar{s}_2) \\ \hline \end{array}, \; s_i = \begin{cases} 0 \; S_i = 0 \\ 1 \; S_i = 9 \end{cases}$$

; where Q is the sign digit after the first reduction.

This result can be used to reduce partial product array in Figure 4.5 to the one in Figure 4.6 (a).

```
S S S S S S S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S S X X X X X X X X X X X X X X X X X
0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S S X X X X X X X X X X X X X X X X X
0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S S X X X X X X X X X X X X X X X X X
0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
S X X X X X X X X X X X X X X X X X
0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
```

X : Digit Coded in 4221
Y : Digit Coded in 5211
H : Includes the (+1) increment
S : Sign Digit

Figure 4.5: Partial Product Array Generated For 16-Digit Operand using SD-Radix 5 implementation

However, the array in Figure 4.6 (a) can be further reduced. If we follwed the same technique used, we can reduce the trailing leading nines in using each two successive vectors starting from the second row as follows:

	Case(1)	Case(2)
$Q_1 \cdots Q_1 Q_1$	$9 \cdots 99$	$9 \cdots 99$
$Q_2 \cdots Q_2 Q_2$	$0 \cdots 09$	$0 \cdots 10$
$Q_R \cdots Q_R Q_R$	$0 \cdots 08$	$0 \cdots 09$

This can be replaced by:

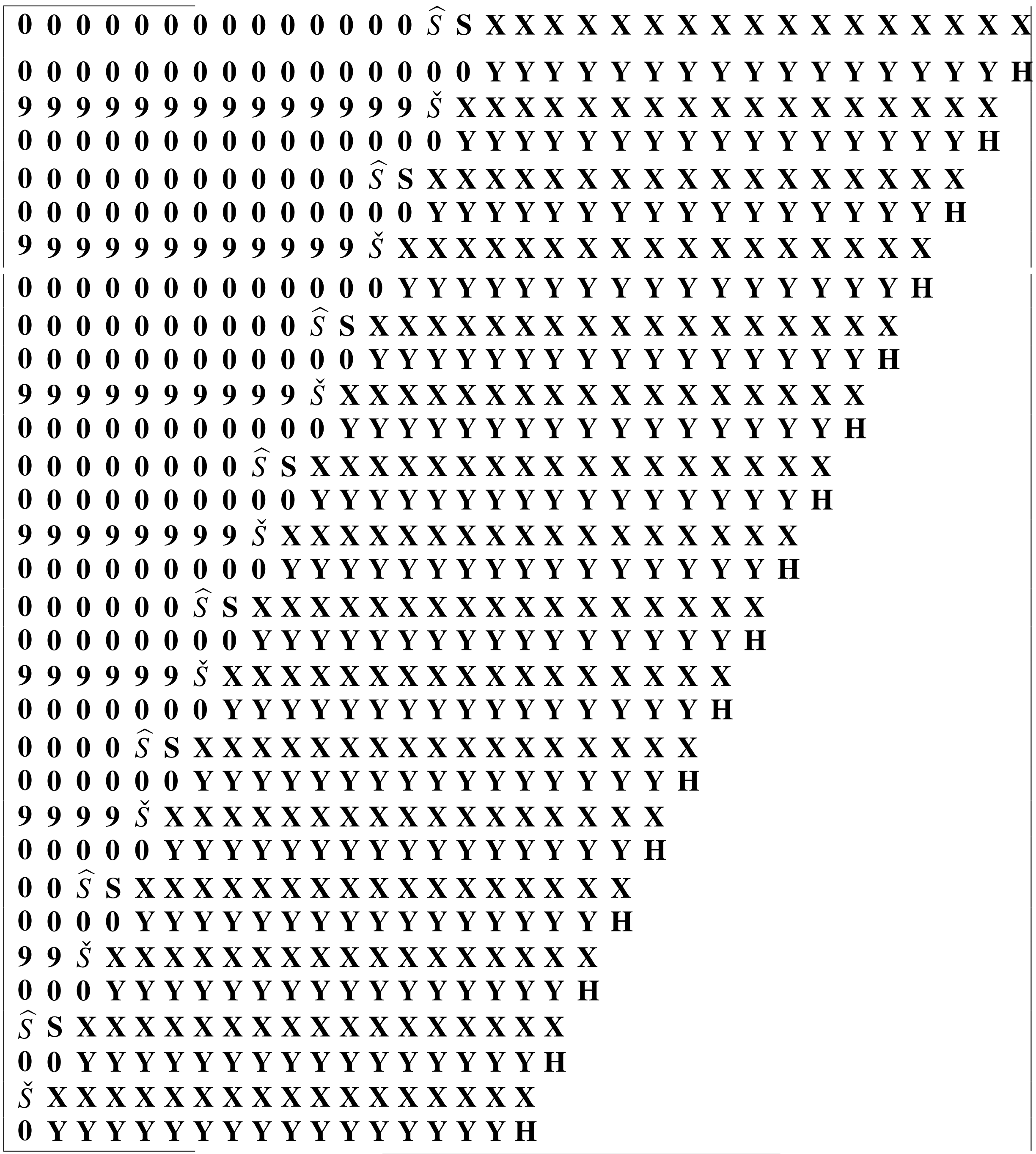
	Case(1)	Case(2)
$W_1 \cdots W_1 W_1$	$0 \cdots 00$	$0 \cdots 00$
$W_2 \cdots W_2 W_2$	$0 \cdots 0(111\bar{s}_2)$	$0 \cdots 0(111\bar{s}_2)$

; where W is the sign digit after the second reduction.

There is an important result we can conclude, that is using this design-time technique for sign extension reduction yields a final array that will produce two vectors one of them is definitly negative and the other must be positive to result in a final positive result. This can be clearly shown from the leading (9) at the most significant position in the multiplier array that will be present in the sum vector.

4.2.2.4 Partial Product Reduction

To simplify the partial product reduction, properties (1) and (2) for 4221/5211 coding are used. Let us assume three decimal digits A_i, B_i, C_i where each digit is coded in 4221 or 5211 format. If a conventional binary carry save adder is used to reduce these three digit to only two digits; sum



X : Digit Coded in 4221

Y : Digit Coded in 5211

H : Includes the (+1) increment

S : Sign Digit

$$\hat{S} = 111\bar{s}\,,\; s = \begin{cases} 0 & S = 0 \\ 1 & S = 9 \end{cases}$$

$$\check{S} = 000\bar{s}\,,\; s = \begin{cases} 0 & S = 0 \\ 1 & S = 9 \end{cases}$$

Figure 4.6: First Reduction Step

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 0 Š X X X X X X X X X X X X X X X X X
0 0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
0 Š X X X X X X X X X X X X X X X X X
0 0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
Š X X X X X X X X X X X X X X X X X
0 Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y H
```

X : Digit Coded in 4221

Y : Digit Coded in 5211

H : Includes the (+1) increment

S : Sign Digit

$$\hat{S} = 1\!7\!1\!7\!1\bar{s}\,,\ s = \begin{cases} 0\,S = 0 \\ 1\,S = 9 \end{cases}$$

$$\check{S} = 000\bar{s}\,,\ s = \begin{cases} 0\,S = 0 \\ 1\,S = 9 \end{cases}$$

Figure 4.7: Final Reduced Array

(S) and carry (H), that can be expressed in Equation 4.10 then, we are only interested in $(r_3, r_2, r_1, r_0) = (4,2,2,1)$ or $(5,2,1,1)$.

$$A_i + B_i + C_i = \sum_{j=0}^{3} (a_{i,j} + b_{i,j} + c_{i,j}) r_j = \sum_{j=0}^{3} s_{i,j} r_j + 2 \times \sum_{j=0}^{3} h_{i,j} r_j$$

$$= S_i + 2 \times H_i \tag{4.10}$$

This equation applies both the logic that the binary carry save adder forces and property (1). Property (1) states that any logical operation on a decimal digit coded in 4221/5211 will result in another valid decimal digit coded in the same format. Also, the binary carry save adder forces the sum bits to be of the same weight as the added bits, yet the carry bit to be of double the weight. These two facts formulate equation 4.10.

However, a decimal multiplication by 2 is required before using the carry digit H_i for later computations. Since property (5) is restricted to 4221/5211 coding out of the decimal coding familty that satisfies Equation 4.4, then they are used to simplify the ×2 operation. The carry digit (H_i) is first converted to 5211 format and shifted to the left by 1-bit, then reused again as coded in 4221. This is equivelent by multiplication by 2.

In order to reduce the partial product array in Figure 4.6 (b), highest columns are first reduced using decimal counters. Then, the result is reduced again by a tree of carry save adders. This tree is designed to reduce the critical path using parallelism as much as possible and balancing the delay of different paths. For this purpose, the intermediate operands with higher multiplicative factors are multiplied in parallel with the reduction of the other intermediate operands using binary 3:2 CSAs. However, this technique may result in carries of four times the weight of sum digits. Hence, in these cases we need a ×4 operation which is simply composed of two ×2 cascaded stages.

To reduce the multiplier array into only two vetcors, we will need 32 trees of carry save adders $(n:2)$ where $n \in \{4, \cdots, 16, 32\}$. An example of these CSA trees can be found in [33].

At the end of this stage in the multiplier tree, two vetcors are obtained: the sum vector (M1) and the carry vector (M2). Both vectors are of 33 digit and coded in the 4221 format. This includes a final digit that represents the sign of each vector.

As shown in Section 4.2.2.3, the sum vector is always negative (i.e. sign digit is 9) and the carry vector is always positive (i.e. sign digit is 0).

4.2.3 Addend Preparation

In parallel to the multiplier tree, the third operand (C) is prepared to be added to the result. The addend is first converted to the 4221 format in order to simplify the decimal carry save addition operation that is performed later on the three operands; the two resulting vectors of the multiplier tree and the addend. Then, the addend's shift amount is determined according to its exponent and the exponent of the multiplication result.

A regular architecture such as the one in Figure 4.8 shifts the result of the multiplication to the right or to the left according to the values of both the exponent of the addend and the multiplication result. However, such architecture requires shifting multiplier result after the end of the multiplication tree, which is at the critical path. Instead, the addend is shifted in parallel to the multiplier tree to the right or to the left maintaining the required relative alignment positions. This is shown in Figure 4.2. This eliminates the need for further processing on the multiplier tree result, which in turn removes some overhead from the critical path.

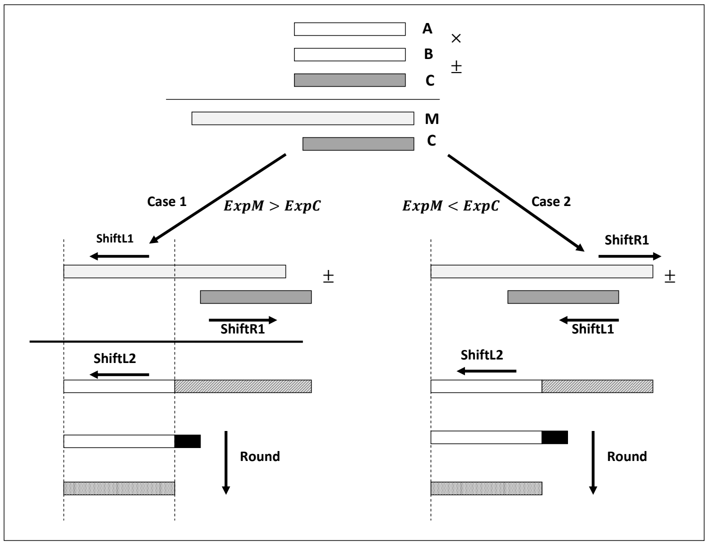


Figure 4.8: Regular Algorithm For FMA Operation

Hence, we should have an R/L shifter that shifts the addend either to the right with a value of $ShiftR$ or to the left with a value of $ShiftL$. This shift amount is called $Shift1$. The shift amount is determined as follows:

$$ExpM = ExpA + ExpB - Bias \tag{4.11}$$

$$ExpDiff = |ExpM - ExpC| \tag{4.12}$$

$$Shift1 = \begin{cases} ShiftR = Max(ExpDiff, 2p+1) & ExpM > ExpC \\ ShiftL = Max(ExpDiff, 3p+1) & ExpM \leq ExpC \end{cases} \tag{4.13}$$

As shown in Equation 4.13, in case of shifting to the right, the maximum shift amount to be considered is (2p+1), otherwise all the addend will contribute to the sticky (unless the multiplication result (M)=0, yet this case is considered as an exception of the default datapath). It should be noted that, although only (p) digits to right of the fractional point may be used again, the maximum shift amount, in the worst case, is 2p+1 with the less significant (p+1) digits out of the (2p+1) width only contribute to the sticky. Also, in case of shifting to the left, the maximum shift amount, in the worst case, is 3p+1, otherwise the multiplication result will only contribute to the sticky (unless the addend (C)=0, yet this case is also considered as an exception of the default datapath). The (+1) digit in both cases to maintain a correct round digit that affects the rounding decision.

In summery, we will need a right shifter of width (2p+1) and a left shifter of width (3p+1), however, only 4p+1 width is required for the shifted addend.

After the alignment of the addend is held, the nine's complement of the addend has to be obtained in case of effective subtraction. Since, the addend now is in the 4221 format, the nine's complment is simply obtained by inverting the aligned addend. This is done using a level of XOR gates that xor addend bits with the effective operation signal (eop), where:

$$eop = signA \oplus signB \oplus signC \oplus op \tag{4.14}$$

Figure 4.9 shows the different stages of addend preparation prior to starting adding it to the multiplication result.

It should be highlighted that the 10's complement requires adding (+1) to the addend, this will be done by injecting (carry in=1) in a later step (rounding set-up) to avoid having an incrementer at the critical path.

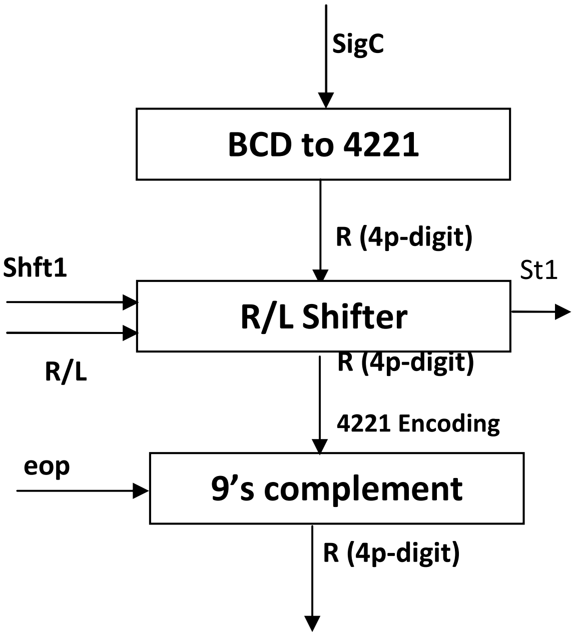


Figure 4.9: Preparing Addend

4.2.4 Selection and Decimal Carry Save Adder

In order to reduce the width of significand path, instead of 4p+1 digits width, a selection stage determines the position of the MSD (Most Significant Digit) that may contribute to the result and ignores digits of higher significance.

This selection is performed according to the four case shown in Figure 4.2. The datapath is controlled by means of the selection lines generated from the Datapath Control Unit as shown in Figure 4.1. This unit also determines the value of $Shift1$ and whether it is a right or a left shift. The four cases are listed in Table 4.2.4.

After this width is selected, a decimal carry save adder is used to add the aligned addend and the two vectors resulting from the multiplier tree. The decimal carry save adder uses the 4221 coding properties to speed up the operation. Figure 4.10 shows the internal struction of the decimal carry

Case #	Condition	Position of the MSD
Case 1	$ExpM > ExpC$	2p+1
Case 2	$ExpC - LZC - p < ExpM \leq ExpC$	2p+1
Case 3	$ExpC - LZC - 2p \leq ExpM \leq ExpC - LZC - p$	3p+1
Case 4	$ExpC - LZC - 2p > ExpM$	4p+1

Table 4.4: Four Cases of Reduced Datapath

save adder. The dotted blocks represent the $\times 2$ operation discussed before. Then a regular binary carry save adder is used. The final sum and carry vectors are converted back to 8421 coding because it is easier in later stages that require carry generation networks as we will see.

Only 2p+1 digits width is required to feed the decimal carry save adder in all cases except in case 2, when the effective operation is subtraction. The two vectors resulting from the multiplier tree has to be added then to a chain of trailing 9's that fills the addend least significant digits of extra p-digits width. If this p-width is removed, this will complicate the design of the following stages in case of effective subtraction.

On the other hand, from section 4.2.2.3, one of the vectors resulting from the multiplier tree (the sum vector) is negative and the other is positive. If we explored the two possibilities of the addend in case of effective addition and subtraction and the possible carry in at the sign digit, we can prove that, in all cases one of the resulting two vectors will be positive and the other will be negative. These different cases are summerized in Figure.

4.2.5 Leading Zero Anticipator

As discussed in Chapter 3, the leading zero anticipator can anticipate the leading zeros, in the result of adding/subtracting two operands, from the operands themselves prior to the addition/subtraction operation. The LZAs

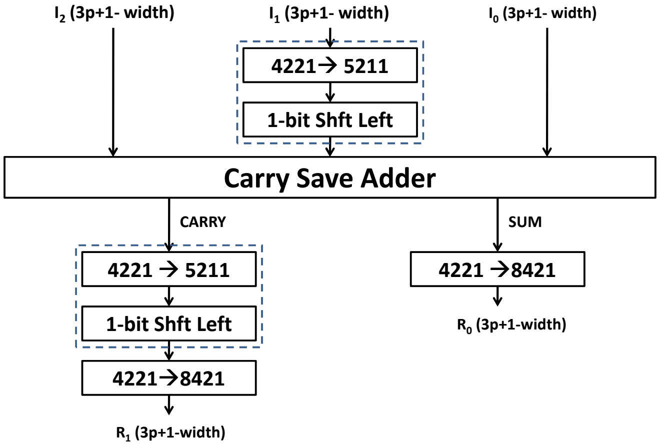


Figure 4.10: Decimal Carry Save Adder

	Sign Ext.	Sign Ext.	Sign Ext.	Sign Ext.
Cin	0	1	0	1
S_{I0}	9	9	9	9
S_{I1}	0	0	0	0
S_{I2}	0	0	9	9
S_{R0}	9	8	0	0
S_{R1}	0	2	8	9

Figure 4.11: Different Cases of Operands to the CSA Signs

consist mainly of two blocks: the preliminary anticipation that anticipates the leading zeros with a possible error of one digit, and the correction unit that corrects that error if found.

However, in our proposed architecture, an error of one digit in the anticipation can be tolerated. This is mainly because the combined add/round block that will be explained later can handle this error. So, in our architecture we will only need the preliminary anticipation stage. This advantage reduces the area and the delay of the LZA step.

In the following, we will discuss the technique used for leading zero anticipation in both effective addition and subtraction operations and also the four different cases of datapath reduction.

4.2.5.1 Inputs to the LZA

Only the most significant 2p+1 width of the two vectors resulting from the decimal carry save adder is considered in the LZA and as stated in the previous section, one of the two vectors resulting from the CSA is negative and the other is positive. Hence, the two operands can be cosidered in the following format:

In case of effective subtraction, the negative operand can be considered in the 9's complement form. This is because, in effective subtraction the addend is negated in the 4221 format to get the 9's complement and the (+1) required for 10's complement is not yet added until this stage of the architecture. So the negative operand can be considered in the 9's complement format and still requires adding (+1) to get the 10's complement.

In case of effective addition, the negative operand is in its 10's complement format because the addend in this case is not 9's complemented. The negative operand comes from the multiplier tree which correctly gets the 10's complement for each negative partial product. Hence, adding the two vectors will result in the required result without further correction (i.e. no need for +1 addition).

4.2.5.2 Effective Subtraction Case

In case of effective subtraction, we use the preliminary leading zero anticipation proposed in [30] with small modifications. Assume the input operands for anticipation are A and $\overline{B}$ (the nine's complement of B), such that A_i is the i^{th} digit in operand A, and B_i is the i^{th} digit in operand B and it is required to anticipate the leading zeros in $(A-B)$. The steps used in the anticipation can be listed as follows:

1) Encode the two operands digitwisely into these signals (g9, g2, g1, zero, s1, s2, s9). Table 4.2.5.2 shows the meaning of each of these signals.

Signal	Condition	Signal	Condition
$g9_i$	$A_i = 9, B_i = 0$	$s9_i$	$A_i = 0, B_i = 9$
$g2_i$	$A_i \geq B_i + 2$	$s2_i$	$A_i \leq B_i - 2$
$g1_i$	$A_i = B_i + 1$	$s1_i$	$A_i = B_i - 1$
$zero_i$		$A_i = B_i$	

Table 4.5: Signaled used for Leading Zeros Anticipation

2) Explore different digit pattern sets and get leading zero count of each pattern.

3) Use these signals to get a binary string P that has leading zero bits equivalent to the leading zero count with a possible error of one digit.

4) Detect the leading zero bits in the binary string using a leading zero detector (LZD) with a logarithmic delay resulting in a preliminary leading zero count PLZC such that:

$$LZA\{A,\overline{B}\} = PLZC\{A-B\} = Correct\,LZC\;or\;Correct\,LZC-1 \quad (4.15)$$

Step 1) Wang and Schulte [30] use a level of one digit subtractors to get the signals (g9, g2, g1, zero, s1, s2, s9) from the result of subtraction digitwisely. They justify this technique by the large fan out that will take place if they processed on the input operands directly. This large fan out is due to the simultaneous usage of the operands in the parallel correction part and in the parallel carry network of their propsed decimal floating point adder, where they use leading zero anticipator, at the first place, in parallel to addition to eliminate the delay of leading zero detection after addition.

However, in our proposal, there is no correction circuitry and also the addition combined with the rounding is done after the leading zero anticipator and the alignment of the operands, so it does not affect the fan out of the signals. Hence, the signals (g9, g2, g1, zero, s1, s2, s9) are deduced from the input operands directly by the logic that satifies conditions in Table 4.2.5.2. This takes into consideration that the negative operand (B_i in this case) is in the 9's complement format.

Step 2) The different digit patterns are explored in [30] and the leading zero count in each case is shown in Table 4.2.5.2.

#	Digit Pattern	***LZCR***	***PLZCR***	#	Digit Pattern	***LZCR***	***PLZCR***
1	$0^k[2,9][-9,9]^m$	k	k	10	$0^k[-9,-2][-9,9]^m$	k	k
2	$0^k1[1,9][-9,9]^m$	k	k	11	$0^k(-1)[-9,-1][-9,9]^m$	k	k
3	$0^k10^t[0,9][-9,9]^m$	k	k	12	$0^k(-1)0^t[-9,-1][-9,9]^m$	k	k
4	$0^k10^t[-9,-1][-9,9]^m$	$k+1$	k	13	$0^k(-1)0^t[1,9][-9,9]^m$	$k+1$	k
5	$0^k1[-9,-1][-9,9]^m$	$k+1$	k	14	$0^k(-1)[1,8][-9,9]^m$	$k+1$	k
6	$0^k1(-9)^j[1,9][-9,9]^m$	$k+j$	$k+j$	15	$0^k(-1)9^j[-9,-1][-9,9]^m$	$k+j$	$k+j$
7	$0^k1(-9)^j[-8,-1][-9,9]^m$	$k+j+1$	$k+j$	16	$0^k(-1)9^j[1,8][-9,9]^m$	$k+j+1$	$k+j$
8	$0^k1(-9)^j0^t[0,9][-9,9]^m$	$k+j$	$k+j$	17	$0^k(-1)9^j0^t[-9,-1][-9,9]^m$	$k+j$	$k+j$
9	$0^k1(-9)^j0^t[-9,-1][-9,9]^m$	$k+j+1$	$k+j$	18	$0^k(-1)9^j0^t[1,9][-9,9]^m$	$k+j+1$	$k+j$

Table 4.6: Different Digit Patterns

Step 3) After the seven signals (g9, g2, g1, zero, s1, s2, s9) are generated for each digit, they are recoded again to produce a binary string P using the following logic Equation:

$$P_i = zero_{i+1} \,.\, (g2_i \,|\, s2_i \,|\, (g1_i \,.\, \overline{s9_{i-1}}) \,|\, (s1_i \,.\, \overline{g9_{i-1}}))$$

$$|\, \overline{zero_{i+1}} \,.\, ((s9_i \,.\, \overline{s9_{i-1}}) \,|\, (g9_i \,.\, \overline{g9_{i-1}})) \qquad (4.16)$$

This binary string is intended to have leading zero bit count equivalent to the leading zero digits in the subtraction result of the anticipated operands. This equation can be devised from Table 4.2.5.2.

The first part of the equation describes these cases:

- case (1) :$zero_{i+1} \,.\, (g2_i)$
- case (2,3,4):$zero_{i+1} \,.\, ((g1_i \,.\, \overline{s9_{i-1}})$
- case (10): $zero_{i+1} \,.\, (s2_i)$
- cases(11,12,13): $zero_{i+1} \,.\, ((s1_i \,.\, \overline{g9_{i-1}})$

Since it does not distinguish between cases (2,3) and case (4), the anticipated leading zero count in case (4) will be of (-1) digit error. The reason behind this is that the digit after the string of zeros (0^t) has to be detected and this will add much overhead to the critical path. Hence, a preliminary leading zero anticipation considers both cases (case(2,3) and case(4)) are the same. The same goes for case (11,12) and case (13).

The second part of the equation describes these cases:

- cases (5,6,7,8,9) :$\overline{zero_{i+1}}\,.\,(s9_i\,.\,\overline{s9_{i-1}})$
- cases (14,15,16,17,18):$\overline{zero_{i+1}}\,.\,(g9_i\,.\,\overline{g9_{i-1}})$

The same discussion is valid in these cases. Cases (7,9,16,18) are anticipated with (-1) digit error to simplify the preliminary anticipation step as much as possible.

Step 4) In this step, the leading zero bits in the binary string P must be detected. Since the binary string (P) is of 33 bits width, the leading zero detection algorithm is critical to the performance of the LZA. It must be as fast as possible to reduce the critical path delay. A logarithmic delay leading zero detector (LZD) algorithm propsed in [77] is used.

The main block in this LZD tree is the LZD of 2-bit sring. This block takes two bits $B1,\ B0$ where $B1$ is the most significant bit and $B0$ is the least significant one. Then, it generates two signals the valid signal (v) and the LZC signal (P) such that:

$$v = B0\,|\,B1 \tag{4.17}$$

$$P = \overline{B1} \tag{4.18}$$

The valid signal indicates if the two bits are zeros or not and the leading zero count signal 'P' indicates if there is a leading zero bit or not. Two of these blocks can be used to detect the LZC signal in a four bit binary string such as Figure 4.12. As shown in Figure 4.12 (b), the two bit signal (P)

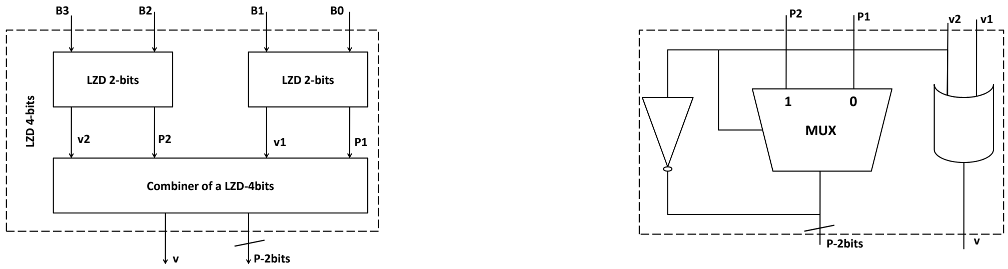


Figure 4.12: (a) Leading Zero Detection for 4-bit Binary String (b)Internal Structure of LZD4

countes the number of leading zeros in the binary string ($B_3B_2B_1B_0$) and the signal $v = 0$ if all the bits in the string are zeros.

This can be generalized for any number of bits. Hence, a 32 LZD can be implemented as 5 levels starting with 16 LZDs of 2-bits as shown in Figure 4.13.

Since in our architecture, there is a 33 bit binary string. The most significant 32-bits are fed to the LZD array and then if it produces an invalid leading zero count (i.e. $v = 0$). This means the leading zero count is either 32 or 33 based on the value of the least significant bit. This gives us the PLZC (preliminary leading zero count) with a maximum of 6-bits width.

This is better than anticipating on the least significant 32 bits, since this will add a 6-bits adder at the critical path delay to increment one in case of having the most significant digit a zero.

Figure 4.14summerizes the steps of the preliminary leading zero anticipator in case of effective subtraction.

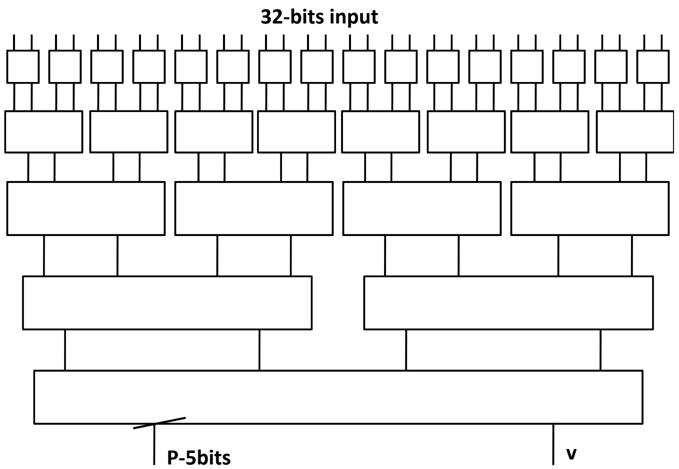


Figure 4.13: Leading Zero Detector of 32-bit Binary String

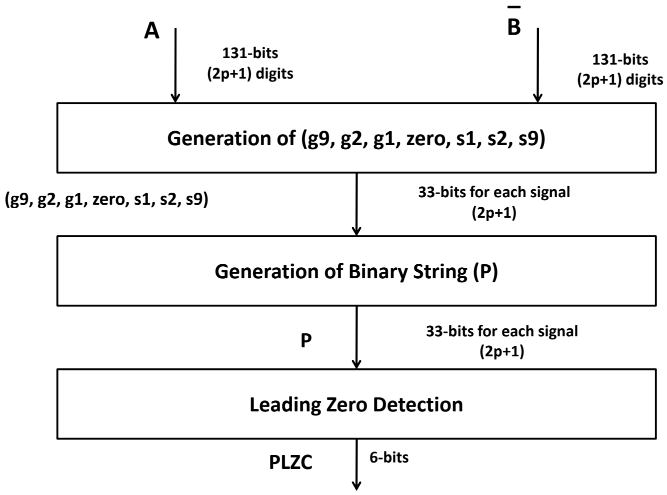


Figure 4.14: Leading Zero Anticipator Block Diagram

4.2.5.3 Effective Addition Case

In this section, we explore the best way to anticipate the LZC in case of effective addition. In the previous decimal leading zero anticipator proposal [30], the preliminary leading zero anticipation in case of addition is calculated as $PLZR_{effadd} = Min\{LZCOp1,\ LZCOp2\}$ where $LZCOp1,\ LCZOp2$ are the leading zero count of the added operands. This anticipation gives the correct result with a possible error of (+1) digit.

However, in our architecture the leading zero count of the operands fed to the LZA (the sum and carry vectors resulting from the decimal carry save adder) are not known and it has to be detected in the critical path.

This can be avoided by using the LZC of initial operands that are required to be added, which are the addend and the multiplication result. The LZC of the addend is detected in parallel to the multiplier tree and there is no problem in it. However, the LZC of the multiplication result is not knwon. Yet, it can be anticipated with a possible error of one digit such that $PLZM = LZCB + LZCC$, for $M = B \times C$. This anticipation may be of (-1) digit error. If we used this anticipation to get the PLZC of the intermediate result such that $PLZR_{effadd} = Min\{PLZM,\ LZCC\}$, this will result in one of three posibilities: $PLZR_{effadd} = Correct\,LZCR - 1\ or\ Correct\,LZCR\ or\ Correct\,LZCR + 1$. This will complicate the combined add/round block since we will have three possible positions for rounding.

To avoid this, we have to devise a logic that anticipates the exact leading zero count of the multiplication result based on the multiplier and the multiplicand in parallel to the multiplier tree. However, this will add very large area to the deisgn.

In brief, the previously proposed technique is not suitable for our architecture. Hence, we will reformulate the problem of the leading zero anticipation in case of effective addition to overcome this issue. As previously stated, in case of effective addition, one of the two operands is postive and the other is negative in the 10's complement format. Using this fact, the problem can be reformulated as follows:

$$LZA\{A,(\overline{B}+1)\} = PLZCR\{A+(\overline{B}+1)\} \tag{4.19}$$

However, this is equivelent to:

$$LZA\{A,(\overline{B}+1)\} = LZA\{A,\overline{(B+1)}\} = PLZCR\{(A-(B+1)\} = PLZCR\{(A-1)-B\} \tag{4.20}$$

Hence, the leading zero anticipation problem in case of effective addition of (A,B) is converted to be a leading zero anticipation problem for effective subtraction but for ((A-1), B). In order to examine the effect of this (-1) in the LZA design, we explored its effect on the different digit pattens and got the new correct LZCC in each case. For example, the next table 4.2.5.2 shows the effect of the (-1) on one of the digit patterns.

Initial Digit Pattern	Subsets of The Initial Digit Patterns	Effect of (-1)
$0^k1(-9)^j0^t[0,9][-9,9]^m$ Case(8), $PLZR = k+j$	All case except $0^k1(-9)^j0^t0(-9)^m$	$0^k1(-9)^j0^t[0,9][-9,9]^{m-1}[-9,8]$ (Case 8)
		$LZR = k+j$
	$0^k1(-9)^j0^t0(-9)^m$	$0^k1(-9)^j0^t(-1)(0)^m$ (Case 9)
		$LZR = k+j+1$

Finally, we concluded that; in all cases, the anticipation on the two operands without any modifications on the logic of the LZA proposed in case of subtraction will result in either the correct LZC of the result in case of effective addition or the correct LZC -1. This (-1) error can be tolerated by the combined add/round module as stated previously.

For more explanation, four numerical examples are given next.

Effective Addion			
Example-1		Example-2	
A	01012100	A	05000438
$\overline{B}$+1	99992400	$\overline{B}$+1	95099560
R	01004500	R	00099998
LZC	1	LZC	3
A	01012100	A	05000438
$\overline{B+1}$	00007599	$\overline{B+1}$	04900439
$g9$	00000000	$g9$	00000000
$g2$	00000000	$g2$	00000000
$g1$	01010000	$g1$	01000000
$zero$	10100000	$zero$	10011110
$s1$	00000000	$s1$	00000001
$s2$	00001111	$s2$	00000000
$s9$	00000011	$s9$	00100000
P	01000010	P	00100001
$PLZC$	1	$PLZC$	2
Error	0	Error	-1

Effective Subtraction			
Example-1		Example-2	
A	01012100	A	05000438
$\overline{B}$	99992400	$\overline{B}$	95099560
R	01004501	R	00100000
LZC	1	LZC	2
A	01012100	A	05000438
B	00007599	B	04900439
$g9$	00000000	$g9$	00000000
$g2$	00000000	$g2$	00000000
$g1$	01010000	$g1$	01000000
$zero$	10100000	$zero$	10011110
$s1$	00000000	$s1$	00000001
$s2$	00001111	$s2$	00000000
$s9$	00000011	$s9$	00100000
P	01000010	P	00100001
$PLZC$	1	$PLZC$	2
Error	0	Error	0

4.2.5.4 Reduced Datapath of Cases (1, 2, 3 and 4)

In case (2), the two operands are limited within the anticipated width. Hence, there is no problem in this case at all. However, in the other three cases, the operands are not limitted to the anticipated (2p+1) width. Yet, this does not affect the correctness of preliminary anticipation, unless the preliminary anticipation finds that all digits within the (2p+1) width are zeros.

In this case, the actual value of leading zeros may be more than (2p+1) depending on the digits outside the anticipation width. However, in all cases, the preferred exponent will be reached with a shift amount smaller than or equals 2p+1. Hence, there is no need to get the exact value of leading zeros if they are more than 2p+1.

4.2.6 Intermediate Sign Detection

In case of effective subtraction, the intermediate result is calculated as (Multiplication Result-Aligned Addend). If (Aligned Addend >Multiplication Result), the intermediate result will be negative and needs complementing to get the absoulte value with appropriate sign as the standard specifies.

The intermediate sign is regularly caluclated after addition by detecting the final carry out. Since effective subtraction is perfromed by adding the 10's complement of the subtracted number, therefore if $Cout_{final} = 1$, then the intermediate result is negative or zero. Otherwise, it is positive. However, there are two main drawbacks of using this simple technique in our architecture, which are:

1) The intermediate sign is required before the addition/rounding step. As we will explain later, the combined add/round has to use the intermediate sign of the result in the precorrection stage, i.e. before actual addition.

2) This technique will not always result in a correct value for the intermediate result sign. This is because, in our architecture, the operands are shifted before addition so the carry out of the shifted operands is not the same as the final carry out of operands before shift. For example:

Non Aligned Operands		Aligned Operands (Shifted left by 3-digits)	
A	9899	$Aaligned$	9000
B	9908	$\overline{B}aligned$	1999
$\overline{B}$	0091	$Aaligned$+$\overline{B}aligned$	1 0999
$A+\overline{B}$	0 9990	$A-B)aligned$	0 9000
$A-B$	0009		
$A-B)aligned$	9000		

The carry out in case of adding $A+\overline{B}$ is '0' , as expected, to indicate a negative intermediate result. However, if we aligned the operands before addition, the carry out is '1' . Hence, we can not depend on the carry out in this case to detect the sign of the intemediate result.

Therefore, we propose a simple sign detection tree that operates in parallel to the preliminary anticipation. We use the $zero_i$ and $Gr_i = g1_i \,|\, g2_i$ signals to detect the intermediate sign in case of effective subtraction. Where $Gr_i = 1$ indicates that digit A_i is greater than digit B_i. This vector Gr and the $zero$ vector are used as inputs to the sign detection tree shown in Figure 4.15.

4.2.7 Final Alignment

The two operands are shifted by the preliminary leading zeros, unless the preferred exponent or the minimum exponent are reached by a smaller value of shift. Moreover, in some cases the operands may be shifted to the right to avoid underflow.

The shifting in case of no underflow is always to the left; according to Equation 4.21.

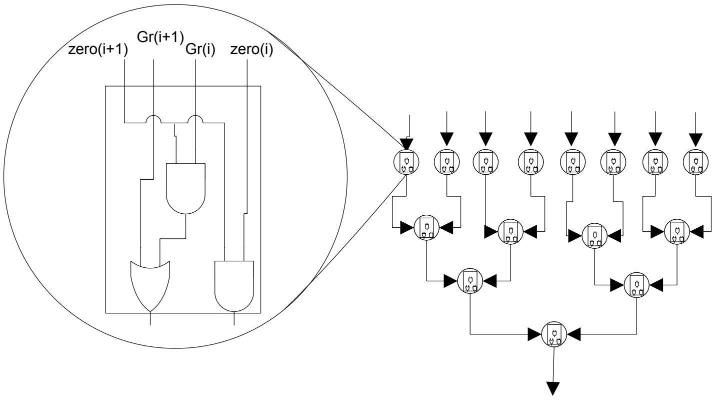


Figure 4.15: Intermediate Sign Detector For 16-Digit Operands

$$Shift2_{no\,underflow} = \begin{cases} LShift = Min\{PLZC, ExpDiff\} & case(1) \\ LShift = Min\{PLZC, p+1\} & case(2) \\ LShift = PLZC & case(3,4) \end{cases} \tag{4.21}$$

In case (1) where $ExpM > ExpC$, the preferred exponent is $ExpC$. At the step of the addend alignment, the addend in case (1) is shifted to the right by the $ExpDiff$ to have the same exponent as $ExpM$. Therefore, if at the final alignment stage, the intermediate result (represented in the two vetcors resulting from the decimal carry save adder) is shifted back to the left by the ExpDiff, the preferred exponent is reached and no need for larger shift.

In cases (2,3,4) where $ExpC > ExpM$, the preferred exponent is $ExpM$. However, the value $ExpM = ExpA + ExpB$ assumes the fractional point at the right of the least significant digit of the multiplication result. On the other hand, the fractional point of the result has different positions according to the selection case. It is at p+1, 2p+1 or 3p+1 for cases (1,2) , (3) or (4) respectively. This new position of the fractional point results in a reduction in the exponent with a value equals to the new position. We will refer to this value as exp_{bias}; defined by Equation:

$$exp_{bias} = \begin{cases} p+1 & case(1,2) \\ 2p+1 & case(3) \\ 3p+1 & case(4) \end{cases} \tag{4.22}$$

Hence, in cases (2,3,4) the intermediate result must be shifted to the left by a value corresponding to the bias of the significand in each case in order to reach a final exponent equals to $ExpM$ which is the preferred exponent. Therefore, if the $PLZC$ is larger than the Exp_{bias}, the operands are shifted by the bias values only. Yet, in cases(3,4), the PLZC is always smaller than the bias value.

Also, we should highlight that, in case of wrong anticipation and the preferred exponent is not reached, the most significant digit will be zero, hence it requires a final 1-digit left shifter.

However, the underflow case limits the value of final shift or shift the operands to the right instead. Since, the two operands at this stage of exponent equals to $expM$. In order to avoid underflow, the final shift amount must convey to Equation 4.23.

$$expM + exp_{bias} \pm Shift2 \geq expMin \tag{4.23}$$

In order to simplify the following equations and expression we define a value $DiffU$ where$DiffU = expM + exp_{bias} - expMin$.

The Shift2 value is considered negative for left shift and positive for right shift. The right shift amount can be limitted to (p+2) to avoid affecting the correct round digit value. This (+2) digits takes into consideraton the round digit position and a possible carry to this position due to the neighbor digit. Hence, Equation 4.23 and Equation 4.21 can be reformulated to produce a formula for the Shift2 value as in Equation .

$$Shift2 = \left\{ \begin{array}{l} \left. \begin{array}{lr} LShift = Min\{PLZC, ExpDiff\} & case(1) \\ LShift = Min\{PLZC, p+1, DiffU\} & case(2) \\ LShift = Min\{PLZC, DiffU\} & case(3,4) \end{array} \right\} DiffU \geq 0 \\ \left. \begin{array}{lr} RShift = Min(DiffU, p+2) & \end{array} \right\} DiffU < 0 \end{array} \right. \tag{4.24}$$

In order to do this final alignment, two left shifters having length equals the maximum required shift, which is equal to (2p+1) digits, are implemented. Also, two right shifter of maximum shift of (p+2) digits are implemented. Each shifter operates speculatively on one of the two vectors resulting from the decimal carry save adder concatenated with less significant p-digits that either contains zeros (case(2)), part of the addend (case(1)) or part of the multiplier tree vectors (case(3,4)). One of the right shifted or the left shifted version are selected based of the sign of $DiffU$. In case of right shift, a sticky share (StR) is calculated due to the part shifted outside the 3p width processed later.

These shifter should maintain a correct sign extension. For right shifter, one of them has to fill the gap to the left with nines to accomodate the negative operand. Also, in case of effective subtraction, one of the left shifter has to fill the gap to the right with nines to accomodate for the nine's complement of the addend.

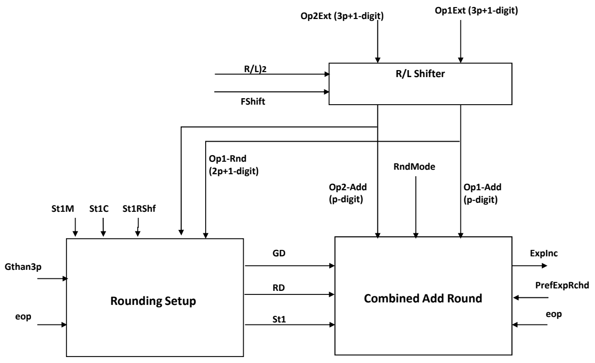


Figure 4.16: Final Alignment Stage

After this, the most significant p-digits is sent to the combined add/round module while the least significant 2p+1 digits are sent to the rounding module. This is shown at Figure 4.16.

4.2.8 Rounding Set Up

4.2.8.1 Top Level Architecture

Figure 4.17 shows the top level architecture of the rounding set up module. The main target of this stage is to calculate, in parallel to the combined add/round module, the guard and the round digits, the sticky bit, and the possible carry in to the most significant (p-digits) fed to the combined add/round module. To calculate these signals, the two vectors passed to the rounding set up has to be added.

In order to maintain only p-digit carry rippling delay at the critical path, the rounding setup is implemented as a conditional adder. The 2p+1 width is divided into p and p+1, the most significant p+1 digits are calculated twice using two carry networks. One of them assumes $C_{in} = 1$ to this part of the adder, while the other assumes $C_{in} = 0$. In order to use a Kogge-Stone binary carry network, a precorrection stage that adds 6 to each digit is

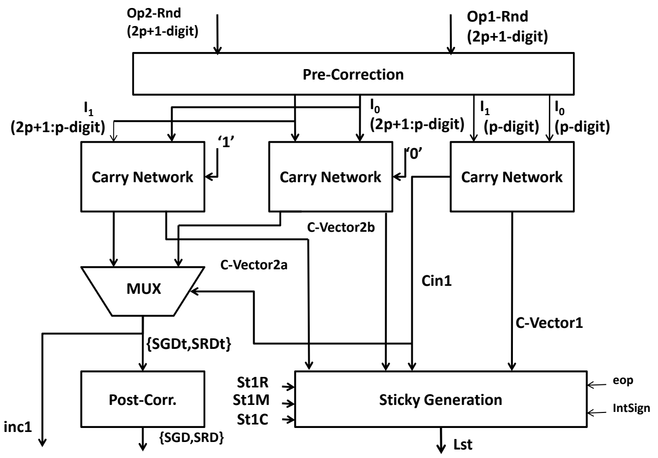


Figure 4.17: Final Alignment Stage

necessray. The carry out resulting from the least significant p digits selects the appropriate carry signals out of the two carry networks.

A small post-correction circuitry is used to generate the correct gurad and round digits. However, the less significant digits contribute only to the sticky and there is no need for their exact values. Hence, no post correction is needed, and they are only processed to get their correct sticky share. The carry out of the most significant carry network is fed to the combined add/round module.

4.2.8.2 The addend 10's complement

In case of effective subtraction, the addend is 9's complemented without adding the (+1) required for 10's complement. Hence, we will use the carry in (C_{in}) of the least significant carry network to perform the required (+1) increment.

However, if the intermediate result is negative, it will need for another 10's complementing which requires adding (+1). In order to avoid this, we use the following property:

$$10's complement(X) = 9's complement(X) + 1 = 9's complement(X - 1) \tag{4.25}$$

Hence, if the effective operation is subtraction and the intermediate result is negative no need to add $C_{in} = 1$ for the 10's complement of the addend and then get the 10's complement again of the intermediate result. Instead, it will be sufficient to get the 9's complement of the intermediate result with the addend in the 9's complement format. However, if the intermediate result is positive, the $C_{in} = 1$ must be added, to account for the (+1) required for the 10's complement of the addend.

4.2.8.3 Generating Final Carry out (inc1), Guard and Round Digits.

As stated, the guard and round digits are calculated using the correct carry signals that are selected according to the carry out of the least significant carry network.

4.2.8.4 Sticky Generation

A portion of the sticky (StC) results from shifting the addend to the right (case-1) with a shift value larger than p-digits while the shifted digits are non-zeros. This is determined from the R/L shifter paralell to the multiplier tree. Another portion (StM) is formed from the least significant p-digits of the multipliction result in (case-4). This is calculated in the selection stage. Also, (StR) is another portion of the sticky due to right shifter at the final alignment stage.

The main portion of the sticky is due to the 2p-digits fed to the rounding set-up stage. A sticky base vector, that represents the zero result, is xored with the sum, and the carries out of the carry network and the carry save adder of the precorrection. After the XOR, the resulting vector is Ored to get the sticky. The sticky base vector takes the form of a trailing nines precorrected (i.e. represented as 15) in case of effective subtraction with a negative intermediate result. Also, in case of affective subtraction a zero result is either a zero that needs post-correction (i.e. represented as 6) or a zero that does not need post-correction (i.e. represented as zero). The post-correction is needed for any digit if there is no carry out from this digit.

4.2.9 Combined Add/Round

This stage performs both BCD addition and rounding. The combined add/round technique is selected to be used in our FMA. Since it achieves the best performance among the three algorithms discussed in Chapter 3. Moreover, it fits better than the rounding by injection algorithm, since rounding by injection requires injecting guard and round digits in the operand with initial larger exponent because it has empty slots at these positions. However, this is not the case in the FMA, where both operands, contribute to the guard and the round. Therefore, the combined add/round solution is better because it allows computing the guard, round and sticky digits in the rounding set-up modules in parallel to the compound addition of the two operands.

However, the proposed combined add/round module dicussed in Chapter 3 is implmented for floating point adders [9]. This design needs some changes and modification to fit in our FMA. We, first, list the main differences between the floating point fused multiply-add operation and the floating point addition operation when using combined add/round at this stage.

1- As stated in the last paragrah, the guard and the round digits are not ready and they have to be computed in the rounding set up module in parallel to the compound addition.

2- Moreover, in case of effective subtraction, the subtracted operand is already complemented at this stage of the FMA.

3- Also in case of intermediate negative result, rounding may be required, however, no rounding is required for negative intermediate result in case of floating-point addition.

4- Finally, as explained in Section 4.2.6, the intermediate sign can not be detected from the final carry out since operands are already aligned by all or part of the leading zeros.

These differences require different modifications in the design to handle them without delay or area overheads.

4.2.9.1 General Algorithm

Combined Add/Round In order to simplify the explanation of the algorithm, we will use at this section a new set of symbols that are independent on the symbols used before. The meaning of each symbol is explained once it is mentioned.

Figure 4.18 (a) explains the regular steps for the addition and rounding of two operands A and B. The most significant (4p-3)-bits of the two operands ,$A^H$and B^H, are fed to the combined add/round module. This is equivelent to the most significant p-digits except the least significant bit. The A^H_{lsb}, B^H_{lsb} are the least significant bits in the most significant p-digits of A,B respectively. While, the least significant 2p+1 digits of the two operands $A^L$and B^L are fed to the rounding set up stage. The signal $inc1$

is a possible increment signal due to carry propagation of the lower significant portion (A^L and B^L). Finally, $inc2$ is a possible increment due to a rounding decision.

Before addition, a pre-correction by a chain of 6's is performed to prepare the operands for a binary Kogge-Stone carry network. The two operands A^H, B^H and the correction vector are added using a carry save addition producing two vectors S^H and C. The two bits A^H_{lsb}, B^H_{lsb} produces a sum bit S^H_{lsb} and a possible carry to the bit at the next significant position in the carry vector C. The final two vectors S^H and C are added using an adder with a binary prefix network. The carry in to this adder is $inc1$. The result of addition is the vector$\{Sum, Sum_{lsb}\}$. In case of negative result, it has to be complemented to result in a new vector $\{R, R_{lsb}\}$. The two vectors inside the curly braces are concatenated. Finally, the rounding decision is decided based on the guard and round digits calculated at the rounding set up module from (A^L,B^L) and the least significant bit of the complemented result (R_{lsb}). This requires another adder to perfrom the incrementation, in case of rounding. The rounded result is then post-correct to produce a final result $\{Result, Result_{lsb}\}$.

However, we want to avoid the two successive addition steps. Therefore, we compute all the possible results due to different possible carries in parallel. Finally, we complement the intermediate result if necessary. Figure 4.18 (b) shows this modified algorithm. As shown, possible carries are '1' , '0' or '-1'. The first two possibilities are very common, howerver, in a certain case we will need the '-1' carry. This is the case of having the intermediate result negative, $inc1 \oplus S^H_{lsb} = Sum_{lsb} = 0$ and $inc2 = 1$. In the regular algorithm, the Sum_{lsb} will be complemented $(R_{lsb} = 1)$ and will produce a carry in to the complement of the most significant part $R = \overline{Sum}$

(i.e. $Result = R + 1 = \overline{Sum} + 1 = \overline{(S^H + C)} + 1)$. However, the complementation is the last step in the modified algorithm, so in order to produce a correct result a carry in equals '-1' has to be added to $S^H\, and\, C$ (i.e. $Result = \overline{(S^H + C - 1)} = \overline{(S^H + C)} + 1)$.

In summery, the three sums $(SUM - 1\,, SUM\,, SUM + 1)$ are possible outputs for the most significant part of the result before complementation, where $SUM = A^H + B^H)_{pre-corrected}$. The final result (after complementation) is selected accroding to the following signals: S_{lsb}, $cmp\,(Intermediate\,Sign)$, $inc1$ and $inc2$. The cmp signal is produced before the combined add/round. It is generated using the intermediate sign detection block that works in parallel to the leading zero anticipator. The $S_{lsb}$is ready immediately after the precorrection stage. However, the $inc1$ and $inc2$ are produced relatively later than this stage. The $inc1$ signal is produced from carry network of the rounding set-up stage and the $inc2$ signal is produced based on the rounding decision.

This can be used to constrain the poossible results in two paris: either $(SUM - 1$ or $SUM)$ in case $\overline{S_{lsb}}.cmp = 1$, or $(SUM$ or$SUM + 1)$ in case $\overline{S_{lsb}}.cmp = 0$. The correct value of each pair will be selected according to $inc1$ and $inc2$. This fact will be used to reduce the area and to avoid unnecessary logic, as we will explain at the precorrection stage. The detailed analysis of the different cases is dicussed in Section 4.2.9.6.

Rounding Position As discussed previously, the rounding position is not exactly determined. It has an uncertainty of one digit due to the uncertainty of the preliminary anticipation of leading zeros. Figure 4.19 shows the unrounded result supposed to be produced from the two operands fed to the combined add/round module with three different cases in which the rounding position has two possibilities: either the least significant digit (LSD), or the temporary gurad digit (GD) produced from the rounding set-up stage.

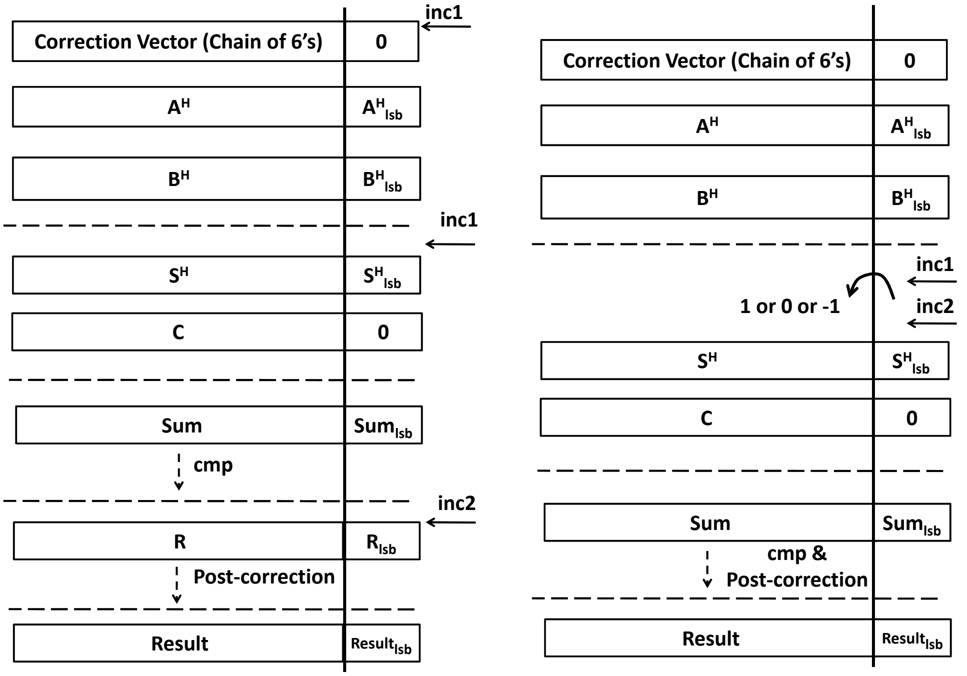


Figure 4.18: Addition and Rounding Steps (a) *Regular* (b) *Proposed*

In the figure, (RD), (Lst) refers to the temporary round digit and sticky digit produced from the rounding set-up stage.

Hence, the rounding signal $inc2$, fed to the most significant highlighted p-digits, must be calculated correctly according to the current case. This requires detecting the MSD of the unrounded result to determine whether it is zero or not, besides using the signal that indicates if preferred exponent is reached or the minimum exponent has reached. Both signals are generated from the final alignment module. To remove any ambiguity, it is important to highlight that; if the preferred exponent is reached the result will be exact and no rounding is required.

Figure 4.20 shows a top-level block diagram for the combined add/round module. In the following, we explain each block in this diagram in details. The symbols on the Figure and in the rest of this section are the same as the discussed before

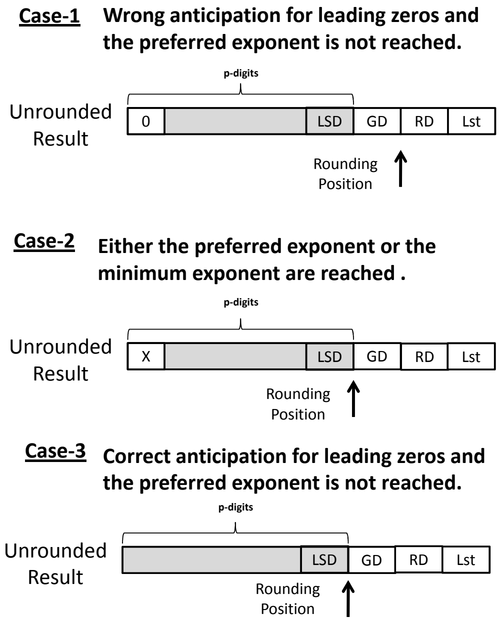

Case-1
Wrong anticipation for leading zeros and the preferred exponent is not reached.
p-digits
Unrounded Result
0
LSD
GD
RD
Lst
Rounding Position
Case-2
Either the preferred exponent or the minimum exponent are reached .
p-digits
Unrounded Result
X
LSD
GD
RD
Lst
Rounding Position
Case-3
Correct anticipation for leading zeros and the preferred exponent is not reached.
p-digits
Unrounded Result
LSD
GD
RD
Lst
Rounding Position

Figure 4.19: Rounding Position

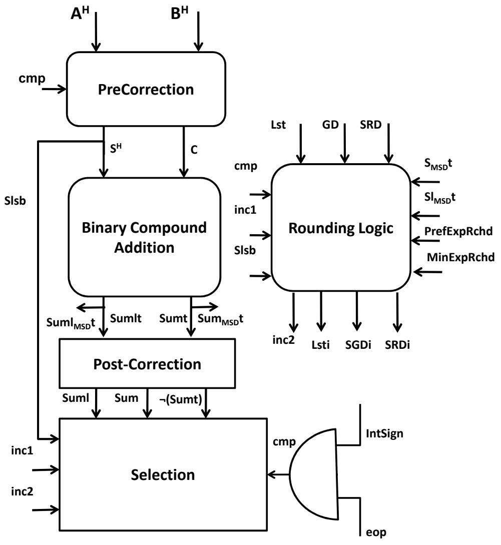


Figure 4.20: Combined/Add Round Module

4.2.9.2 Pre-Correction

To allow the use of a fast binary adder that uses a Kogge Stone carry network, the p-digit BCD operands $\{A^H, A^H_{lsb}\}$ and $\{B^H, B^H_{lsb}\}$ are first precorrected. The precorrection is done by adding (+6) to each digit using a 4p-bit binary 3:2 CSA, obtaining the 4p-bit sum and carry operands S and C_{int}. The carry vector C_{int} is an intermediate carry vector that will be corrected again as we will see later to produce the final carry vector C.

Each +6 bias, coded in BCD as (0, 1, 1, 0), is connected to an input of a 4-bit binary 3:2 CSA. The p-digit BCD operands A and B are fed to the other two inputs.

However, this stage can be also used to prepare for the computation of the $SUM-1$ required in case of $\overline{S_{lsb}}.cmp = 1$ where $SUM = A^H + B^H)_{pre-corrected} = A^H + B^H + \{6, 6, \cdots, 6\}$. This is done by replacing the least significant digit of the correction vector by '5' instead of '6' in this case; which implements '-1' decrementation without needing any extra logic.

The rest of pre-correction unit is the same as the one proposed in [76]. In order to simplify the post-correction stage, the sum resulting from the compound adder is forced to be in excess-6 representation in all cases. hence, for any two digits at the position '*i*' with sum greater than 9 (i.e. $G_i = 1$), adding (+6) is not sufficient to produce a correct decimal sum in the excess-6 representation. Hence, if G_i equals to '1', another (+6) must be added to the result. The (+6) is added to the intermediate carry vector C_{int} digitwisely producing the final carry vector C.

In case of having the digit sum equals 9 and the decimal carry in equals '1', the result of addition will be a zero that is not represented in excess-6 representation. To calculate the result in excess-6, a slight modification in the xor sum cells of the binary adder is performed [76].

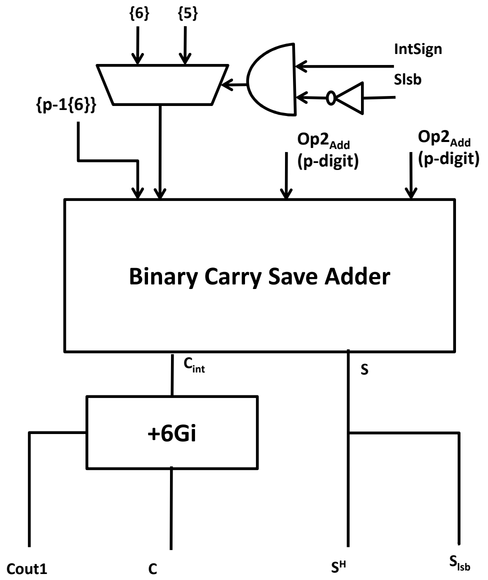


Figure 4.21: Pre-Correction Block

4.2.9.3 Compound Adder

We use the same compound adder proposed in [76], shown in Figure 4.22. This adder computes $Sumt = S + C$ and $Sumlt = S + C + 1$. The $Sumt$ and $Sumlt$ are pre-corrected. However, we should remember that in case of $\overline{S_{lsb}}.cmp = 1$, $S + C = A^H + B^H - 1)_{pr-corrected} = SUM - 1$. Hence, the compound adder produces either $(SUM - 1, SUM)$ pair or $(SUM, SUM + 1)$ pair according to Table 4.2.9.3.

Condition ($\overline{S_{lsb}}.cmp$)	Sumt	Sumtl
0	$SUM - 1$	SUM
1	SUM	$SUM + 1$

Table 4.7: Outputs of the Compound Adder

The binary carry recurrence $c_{k+1} = g_k | a_k.c_k$ is implemented in a prefix Kogge Stone carry tree, where g_k and a_k are the binary carry generate and carry OR-propagate functions for bit number k. The decimal carries $C2d_i$

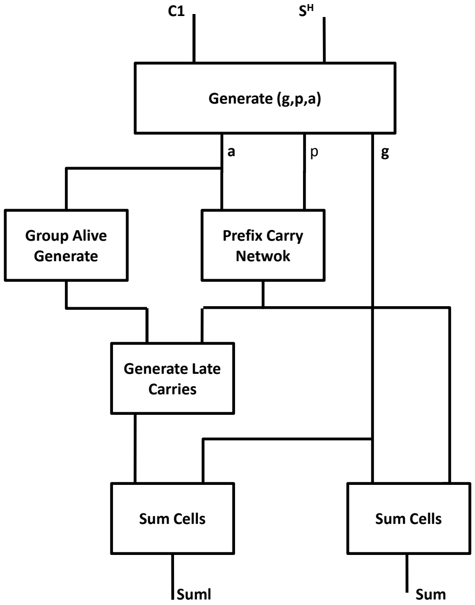


Figure 4.22: Compound Adder

are the binary carries c_k at decimal positions (1 out of 4), where 'i' is the index of the decimal digit. Also the carries$C1d_i$ at the decimal positions of the carry vector $C1$ are decimal carries. Therefore, the decimal carries between digits are $Cd_i = C1d_i | C2d_i$. The 4-bit binary sum $S_i + C_i + C2d_i$ represents the BCD excess-6 sum digit $Sumt_i$.

On the other hand, the length of the decimal carry propagation due to a late +1 ulp increment equals the trailing chain of 1's in the precorrected sum (sum). In other words, a carry due to a +1 ulp increment of Sum is propagated to bit k if the binary carry OR-propagate group $a_{k-1:1} = a_{k-1}.a_{k-2} \cdots a_1$ is true. The binary carries of $Sumlt$ are obtained as $lc_k = c_k . a_{k-1:1}$. The digits $Sumlt_i$ are obtained from the binary XOR-propagates $p_{i,j}$ and the binary carries $lc_{i,j}$ $(k = 4i + j)$ using a 4-bit sum cell similar to the one described before.

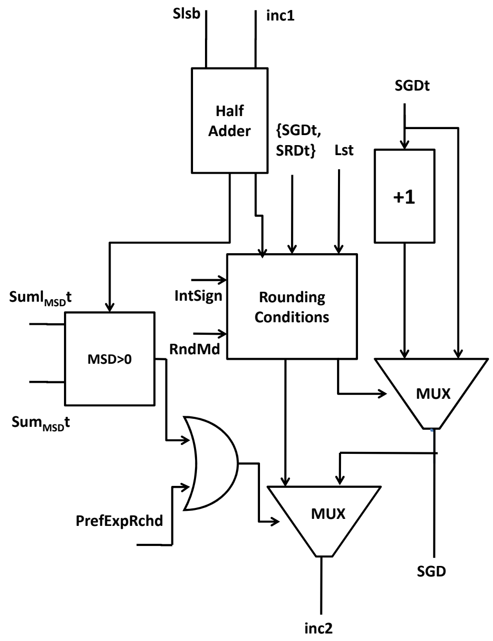


Figure 4.23: Rounding Stage

4.2.9.4 Rounding Stage

In parallel to the binary sum, the decimal rounding unit computes the increment signal inc2 and the guard digit of the result. Apart from the rounding mode, the rounding decision depends on the intermediate sign, the guard, the round and the sticky digits computed in the rounding set-up stage, the sum least significant bit (Slsb) and the rounding position. The rounding position depends on the unrounded result MSD and the two signals that indicate if the preferred exponent is reached or the minmum exponent is reached. The MSD of the unrounded result can be anticipated using the MSD of Sumt, Sumlt, Slsb and inc1.

Rounding Logic As shown in Figure 4.23, it uses a combinational logic to implement directly a rounding condition for each decimal rounding mode. Moreover, different conditions must be applied on the two possible rounding positions.

In other words, the rounding is performed speculatively on the two possible rounding positions. The rounding conditions block produce two rounding decisions for each possible position. The signal inc2 is either due to a rounding decision at the least significant digit (i.e. (case-1) in Figure 4.19) or a rounding decision at the guard digit (i.e. (case-2,3) in Figure 4.19) and carry propagation through the gurad. The final multiplexer selects the correct inc2 signal according to the correct rounding position. The correct rounding position is calculated using the MSD of the unrounded result and the two signals that indicate if the preferred exponent is reached or the minmum exponent is reached

The following logic determines whether the most significant digit of the unrounded result is zero or not.

$$F = cmp\,.\,(inc1 \,|\, (\overline{inc1}\,.\,\overline{Slsb}))\,|\,\overline{cmp}\,.\,(inc1\,.\,Slsb) \tag{4.26}$$

The signal F equals to '1' in all the cases in which the unrounded result is the 'Sumt' and equals to '0' in all the cases in which the unrounded result is the 'Sumlt'.

$$MSDGZ = \begin{cases} SMSDGZ & F = 1 \\ SLMSDGZ & F = 0 \end{cases} \tag{4.27}$$

where: *SMSDGZ* equals to '1' when the 'Sumt' MSD ('SMSD') is not zero and *SLMSDGZ* equals to '1' when the 'Sumlt' MSD ('SLMSD') is not zero. These signals are computed as follows: (it takes into cosideration that they are precorrected and may be complemented)

$$\begin{aligned} SMSDGZ = cmp\,.\,(SMSD[0]\,|\,SMSD[3]) \\ |\,\overline{cmp}\,.\,(\overline{SMSD[3]}\,|\,\overline{SMSD[2]}\,|\,\overline{SMSD[1]}\,|\,\overline{SMSD[0]}) \end{aligned} \tag{4.28}$$

; similarily for *SLMSDGZ*.

Rounding Conditions In addition to the five IEEE 754-2008 decimal rounding modes [18], we implement two additional rounding modes [25]: round to nearest down and away from zero. The conditions for each decimal rounding mode are summarized in Table 4.2.9.4. Refer to Figure 4.19 to understand the meaning of the symbols in this table. The logical operator ‘||’ refers to logical OR and the logical operator ‘&&’ refers to logical AND.

Rounding Mode	Rounding Position	Action

Round Ties to Even	GD	if(RD>5 || RD=5 && Lst=1)
		round to nearst up
		else if (RD=5 && Lst=0 && $GD \Rightarrow odd$)
		round to nearst up
		else if (RD=5 && Lst=0 && $GD \Rightarrow even$)
		truncate

	LSD	if(GD>5 || GD=5 && (RD>0 || Lst=1))
		round to nearst up
		else if (GD=5 && RD=0 && Lst=0 && $LSD \Rightarrow odd$)
		round up
		else if (GD=5 && RD=0 && Lst=0 && $LSD \Rightarrow even$)
		truncate

Round Ties To Away	GD	if(GD$\geq$5) round up
		else truncate

	LSD	if(RD$\geq$5) round up
		else truncate

Round Ties to Zero	GD	if(RD>5 || RD=5 && Lst=1) round up
		else truncate

	LSD	if(GD>5 || GD=5 &&(RD>0|| Lst=1)) round up
		else truncate

Round Toward Zero	GD	Truncate
	LSD	Truncate

Round To Away	GD	if(RD>0 || Lst=1) round up
		else truncate

	LSD	if (GD>0 || RD>0 || Lst=1) round up
		else truncate

Round Toward Positive	GD	If (IntSign=0 && (RD>0|| Lst=1)) round up else truncate
	LSD	If (IntSign=0 && (GD>0 || RD>0|| Lst=1)) round up else truncate
Round Toward Negative	GD	If (IntSign=1 && (RD>0|| Lst=1)) round up else truncate
	LSD	If (IntSign=1 && (GD>0 || RD>0|| Lst=1)) round up else truncate

Table 4.8: Different Rounding Conditions

4.2.9.5 Post-Correction and Selection

The post-correction stage is very simple. In case of positive intermediate result, a '+10' is added digitisely to both Sumt and Sumlt to produce post-corrected sums: Sum and Suml. It is equivelent to subtracting '6' from each digit.

If the intermediate result is negative and the result will be complemnted, it does not need post-correction. Hence, inverting of an excess-6 BCD number is equivelent to getting the 9's complement of an 8421 BCD number.

The selection stage selects the correct output out of Sum, Suml, inverted version of Sumt or Sumlt. Table 4.2.9.5 shows all possible cases with the selected final result in each case.

We have to remeber that in cases (9,11,13,15) where $\overline{S_{lsb}}.cmp = 1$; the (Sum, Suml) pair represents (SUM-1,SUM).

case #	cmp	inc1	inc2	Slsb	Selected Signal
1	0	0	0	0	$\{Sum,0\}$
2	0	0	0	1	$\{Sum,1\}$
3	0	0	1	0	$\{Sum,1\}$
4	0	0	1	1	$\{Suml,0\}$
5	0	1	0	0	$\{Sum,1\}$
6	0	1	0	1	$\{Suml,0\}$
7	0	1	1	0	$\{Suml,0\}$
8	0	1	1	1	$\{Suml,0\}$
9	1	0	0	0	$\{\overline{Sumlt},1\}$
10	1	0	0	1	$\{\overline{Sumt},0\}$
11	1	0	1	0	$\{\overline{Sumt},0\}$
12	1	0	1	1	$\{\overline{Sumt},1\}$
13	1	1	0	0	$\{\overline{Sumlt},0\}$
14	1	1	0	1	$\{\overline{Sumlt},1\}$
15	1	1	1	0	$\{\overline{Suml}t,1\}$
16	1	1	1	1	$\{\overline{Sumt},0\}$

Table 4.9: Selected Final Result

Finally, the result may be shifted to the right or to the left by one digit. If the MSD after rounding is zero and neither the preferred exponent nor the minimum exponent are reached, then the result is shifted to the left by 1-digit. On the other hand, if a final carry out due to rounding is propagated through all precision digits, then the result is shifted to the right by 1-digit. The sticky value is updated in both cases.

4.2.10 Encoding the result

The result is again encoded in the 64-bit DPD format. The logic derived in Section 4.2.1 is inverted to encode the decimal floating point number again. The special signals discussed in Section 4.6 are inputs to this enoding circuit.

4.3 Default Exponent Datapath

The exponent is calculated as follows:

$$expt1 = expM - Shift2 \tag{4.29}$$

$$expt2 = expt1 - dec + inc \tag{4.30}$$

, where dec=1 if the preferred exponent is not reached and the most significant digit of the rounded result is zero and inc=1 if a carry out at the final digit is detected due to rounding in case of effective addition.

$$expt3 = expt2 + exp_{bias} \tag{4.31}$$

$$expR = \begin{cases} expMax & expt3 > expMax \\ expt3 & expt3 \leq expMax \end{cases} \tag{4.32}$$

There are some exceptions that are discussed later.

4.4 Default Sign Logic

The sign is calculated as follows:

$$SignR1 = \overline{SignIR} \,.\, signM \,|\, signIR \,.\, signCeff \tag{4.33}$$

where $SignIR$: is the intermediate result sign.
and $SignCeff = eop \oplus signC$ is the effective sign of the addend.
and $SignM = signA \oplus signB$ is the multiplication result sign.

The intermediate result equals to $M \pm C$; therefore, if the intermediate result is negative this means that $C > M$ and the final result will follow the addend effective sign ($eop \oplus signC$). If $M > C$ then the intermediate sign will be positive and the final result will follow the multiplication result sign. In case of having $M = C$ with the intermediate result being zero, then the final sign will be zero in all cases unless the rounding mode in rounding to negative infinity. this is defined by equation 4.34. Equation 4.35 gives the final sign of the result. Again, there are some exceptions that will be discussed in Section 4.7.

$$signR2 = (RndMode == RM) \tag{4.34}$$

$$signR = \begin{cases} signR1 & ExactIntResult \neq 0 \\ signR2 & ExactIntResult = 0 \end{cases} \tag{4.35}$$

4.5 Flag Generation

4.5.1 Inexact Flag

The inexact flag is raised if the result is rounded. It is detected from the sticky, gurad and round digits.

4.5.2 Invalid Falg

The invalid flag is generated in either of these cases:

- One of the operands is sNaN.

- In case of $FMA(0, \pm\infty, c)$ or $FMA(\pm\infty, 0, c)$; where c is any DFP number including special numbers (NaNs, infnities).

The stadard in this case states that it is optional to raise the invalid flag if the third operand in qNaN. In our implementation we activate the invalid flag even if the third operand in qNaN.

- In case of $FMA(|c|,+\infty,-\infty)$ or $FMA(|c|,-\infty,+\infty)$; where c is a DFP number that is not a NaN.

4.5.3 Overflow Flag

The overflow is detected after rounding. It is signaled if the final exponent exceeds the maximum exponent in the standard. If an overflow is detected, the result is rounded either to infinity or to the largest representable number according to the rounding mode and the final sign.

4.5.4 Underflow Flag

If the intermediate result is a non-zero floating point number with magnitude less than the magnitude of that format's smallest normal number (1×10^{-383}, in case of 64-bit format), an underflow is detected. However, the underflow flag is not raised unless the result is inexact.

In our design, the underflow is detected in the final alignment module controller. Referring to Equation 4.24, the result is underflowed in case of right shift decision or in case of left shift decision constrained by the value of $DiffU$. In other words, the minimum exponent is reached while the preferred exponent is not reached and there are still leading zeros. In case of $PLZC = DiffU$, the underflow exception is not raised unless the preliminary anticipation is wrong and the MSD of the unrounded result is zero.

4.6 Special Signals Generation

4.6.1 Infinity

The result is infinity either due to rounding decision in case of overflow or due to a mathematical operation that invloves infinities such as the operations discussed in chapter 2.

4.6.2 Not a Number (NaN)

Regular operation does not produce signaling NaN (sNaN) result. While, the result is quite NaN (qNaN) in three different cases. . First, the result is qNaN due to invalid multiplication such as FMA($0, \pm\infty, c$) or FMA($\pm\infty$, $0, c$). Second, the result is qNaN due to an invalid final addition step. For example, FMA($+|c|, +\infty, -\infty$), FMA($+|c|, -\infty, +\infty$) or any combination that leads to $(|\infty| - |\infty|)$. In the last example, c is any representable floating point number and is not quiet NaN. Finally, the result is qNaN if any of the three operands is either sNaN or qNaN. In all these cases, the default result of the operation shall be a quiet NaN that may provide some diagnostic information.

4.7 Exceptional Datapath

4.7.1 Zero Addend

This case is detected from the signal iszeroC that results from decoding of the addend. If the addend is zero, the default datapath will not produce a correct result in all cases. If the addend is shifted at the preparation stage by a large amount, the multiplication result may be totally or partially considered in the sticky only; which is not the correct answer. Hence, in this case, the addend is not shifted. The selection signals only indicate either preferred exponent equals to *ExpC* or to *ExpM*.

The final shift amount is determined separately. Since the multiplication result is added to zero. It is only required to either shift the multiplication result to the left to reach or to approach the preferred exponent ($ExpA + ExpB$) or to the right to overcome underflow. The different control blocks are reconfigured to handle this exceptional case. The sign and exponent calculation are also modified to produce a correct result in this case.

4.7.2 Zero Multiplier Result

This case is detected from the signal $iszeroM = iszeroA|iszeroB$. In this case, the result should equal to the addend unless the preferred exponent is the exponent of the multiplication result. If so, the addend has to be shifted to the left to reach or approach the preferred exponent. There is no underflow in this case.

The different control blocks are configured to handle this exceptional case. Also, the sign and exponent calculation are modified to produce a correct result in this case.

4.7.3 $ExpC > ExpM + 3p + 1$ and $ExpC < ExpM - 2p$

In case of $ExpC > ExpM + 3p + 1$, the left shift amount of the addend is limitted to $3p + 1$ only. The multiplication result is Ored in the sticky. While, in case of $ExpC < ExpM - 2p$, the right shift amount of the addend in limitted to $2p$ and the addend is Ored in the sticky. Small modifications in the control blocks are needed to handle these cases.

The different combinations of these three exceptional cases are handled to produce a corect result.

4.8 Improvements in the architecture

4.8.1 Reducing the width of the decimal carry save adder

One of the defects in the proposed architecture is that the decimal carry save adder is wider than necessary. The width of the decimal carry save adder is 3p; however, in the worst case only 2p width needs to be reduced. The main problem was to correctly handle the sign extension of the addend and the sum vector resulting from the multiplication result.

Figure 4.24 shows the layout of the three operands (Sum and carry of the multiplier tree and the shifted addend). The two vectors representing the multiplication result are fixed at this position with the sign of the first vector is sign extended to the left with trailing nines (it is always negative). The addend position is variable according to its shift amount. The addend is extended by trailing 9s to the left (as sign extension) and to the right (for a correct 9's complement format) in case of effective subtraction. As shown in the figure, only the width from 3p down to p requires adding three operands. While, to the right and to the left of this width a maximum of two operands have to be added.

Therefore, the decimal carry save adder is reduced to a width of only 2p digits. It is fed by the operands from 3p down to p. However, there is a problem that occurs due to a possible carry out at the MSD of the selected part (i.e. at the digit position 3p). If the carry save adder produces a carry out ($Cout_{msd}$) at this position, it has to be taken in consideration. Thus, the if a carry out is detected, it flips the nines sign extension of the first vector resulting from the multiplier tree to zeros.

However, the LZA operates based on the fact that one of the operand in positive and the other in negative (10's complemented in case of effective addition and 9's complemented in case of effective subtraction). Besides

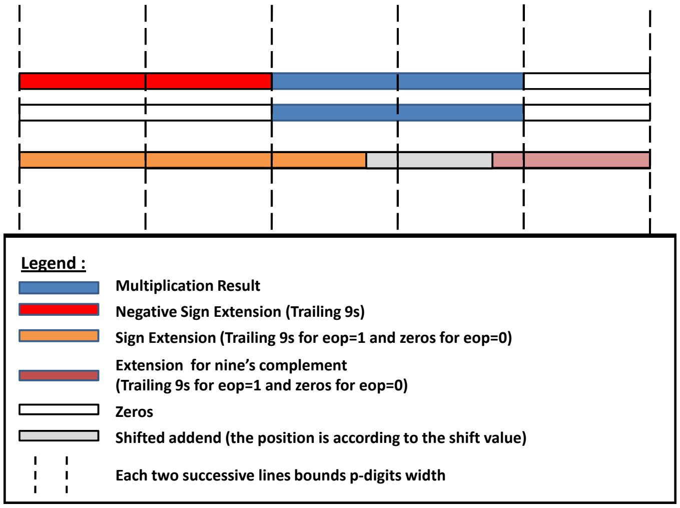

Legend :
Multiplication Result
Negative Sign Extension (Trailing 9s)
Sign Extension (Trailing 9s for eop=1 and zeros for eop=0)
Extension for nine's complement
(Trailing 9s for eop=1 and zeros for eop=0)
Zeros
Shifted addend (the position is according to the shift value)
Each two successive lines bounds p-digits width

Figure 4.24: Layout of the three operands prior to the decimal carry save adder

that, the resulting two vectors out of the carry save adder without prior reduction for the most significant part (5p down to 3p) may be both negative ($eop = 1$ and $Cout_{msd} = 0$) or both positive ($eop = 0$ and $Cout_{msd} = 1$). Therefore, an appropriate sign extension digit is added at the most significant position of the selected part in the selection stage. This sign extension forces one of the operand to first operand to be negative and the second to be positive without changing the correct value. For example, if the two operands are positive they are sign extended as (9) and (1). While, if they are negative they are sign extended as (8) and (1).

4.8.2 R/L shifter in parallel to the LZA

One of the disadvantages of the propsed design is that the LZA is in the critical path. Moreover, it is followed by a control unit that determines the final alignment based on the resulting anticipation, the preferred exponent and the minumum exponent as discussed in section 4.2.7. After the final shift is determined the operands are shifted, hence, the shifter delay is at the critical path as well.

In order to avoid this, the final stage of the LZA that detects the leading zeros in the binary string P (refer to section 4.2.5) is modified such that the most significant bits in the $PLZC$ value are known earlier and compared bitwisely with the other possible value of shift. This allows the shifter to work in parallel with the final stage of the LZA. This final stage consumes the larger delay portion in the LZA since it has a logarithmic delay that depends on the width of the operand while the previous stages work on each digit in parallel. Therefore, it is very important to hide its latency.

Moreover, it is important to highlight that, the right shift amount does not depend on the $PLZC$ of the intermediate result. Hence, there is no problem for it to work in parallel with the LZA. The left shift amount depends on the $PLZC$. The left shift value is the minimum of the $PLZC$, the $DiffU$ which is the shift value that bring the intermediate result to the minimum representable exponent and the $ExpDiff$ in case (1) or $p+1$ in case (2) which bring the intermediate result to the preferred exponent in either cases. The values of the $DiffU$, $ExpDiff$ and the selection control signals are already known prior to the LZA. Hence, we can determine the other possible value of shift that must be compared to the $PLZC$ once anticipated. We will call this value of shift as $Shft1$, that will be of width 11-bits (the same as the exponent width in the 64-bit format). The problem now is to get the $Min(PLZC,Shft1)$.

First, in order to compute the $PLZC$ most significant bits as early as possible, a modified version of the LZD that was presented in [?] is used. As shown in Figure 4.25, the LZD composes of an OR network muxes. NOR gates are used to detemine if there is some 1 in different goups of the

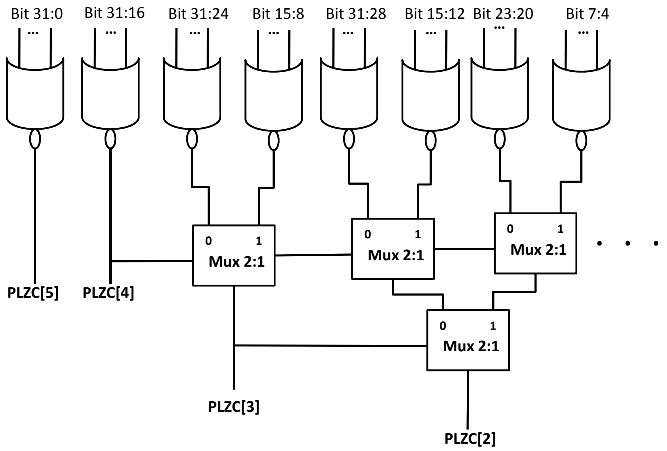


Figure 4.25: Modified Leading Zero Detector Circuitry

string: goups of 32 bits for $PLZC[5]$, of 16 bits of $PLZC[4]$, of 8 bits for $PLZC[3]$, etc. The groups are selected by the set of multiplexers with an increasing number of inputs. Note that muxes of more than 2 inputs are organized as a tree, in such a way that bit $PLZC[i]$ is used as control signal for a 2:1 mux (the last level in the tree) to obtain $PLZC[i+1]$. In this way, the shift is calculated starting from the most significant bit. After an initial delay, due to the wide NOR gate where the first bit $PLZC[4]$ of the $PLZC$ is determined, each bit is obtained with the delay corresponding to a 2-input mux.

In order to perform bit by bit comparison, the resulting $PLZC$ is fed to a comparator circuit that performs the following logic in Equations 4.36 to 4.38.

$$G_i = G_{i+1} \,|\, (Shft1[i] \,.\, \overline{PLZC[i]} \,.\, \overline{S_{i+1}}) \tag{4.36}$$

$$E_i = E_{i+1} \,.\, (\overline{Shft1[i] \oplus PLZC[i]}) \tag{4.37}$$

$$S_i = \overline{G_i \,|\, S_i} \tag{4.38}$$

where:
G_i : indicates that $Shft1[10:i] > PLZC[5:i]$
S_i : indicates that $Shft1[10:i] < PLZC[5:i]$
E_i : indicates that $Shft1[10:i] = PLZC[5:i]$
i : is the bit index where $i <= 5$

The initial values G_6, E_6 and S_6 are given by Equations 4.39 to 4.41:

$$G_6 = Shft[10] \,|\, \cdots \,|\, Shft[6] \tag{4.39}$$

$$E_6 = \overline{Shft[10] \,.\, \cdots \,.\, Shft[6]} \tag{4.40}$$

$$S_6 = 0 \tag{4.41}$$

Finally, the final shift amount is given by Equation 4.42:

$$Shift2[i] = (G_i \,.\, Shft1[i]) \,|\, (S_i \,.\, PLZC[i]) \,|\, (Shft1[i] \oplus PLZC[i]) \tag{4.42}$$

If the designer want to remove the comparator delay, the operands can be shifted speculatively by both $PLZC$ and $Shft1[5:0]$. The comparator in this case works in parallel to the shifter. The correct shifted value is selected according to the signal G_0. If $G_0 = 1$, then the correct shift is $Shft1[5:0]$ else the correct shift is $PLZC$.

4.8.3 Proposed enhancements

There are other ideas that can be applied to enhance the performance of the design. For example, to hide the latency of the first part of the LZA, the LZA may start anticipating in parallel to the decimal carry save adder. This needs the LZA to work on three operands. A 3-operand LZA is proposed for binary in [65]. This work may be extended to decimal.

Alternatively, a part of the compound adder in the combined add/round unit may be advanced to fill the gap of the first stage of the LZA. This will be a decimal extension for the work presented by Lang and Buruguera in their binary FMA [60].

In this chapter, we presented our proposal for the decimal fused multiply-add module. The propsal targets high performance and use many of the ideas of binary FMAs such as the ideas discussed in chapter 2. It also uses modified decimal blocks among the ones discussed in chapter 3. In the next chapter, we extend the FMA unit to an arithmetic unit that performs floating-point addition and multiplication as well as fused multiply-add.

Chapter 5

Pipelined Decimal Floating-Point Arithmetic Unit

In this chapter, we extend our proposed FMA design to a decimal arithmetic unit that performs decimal floating-point addition, multiplication and fused multiply-add. The unit is pipelined to enhance the throughput where the pipelined versions of the design can handle different operations at different cycles.

5.1 Addition

The FMA unit perofrms $A \times B \pm C$. If we need to perform $A \pm C$, it can be easily done by setting $B = 1 \times 10^0$. No other modifications are required in the design. However, this leaves some unnecessary delay in case of addition; such as the multiplier tree and the decimal carry save adder. This effect can be reduced by a pipelined version of the FMA that bypasses the unused stages in case of addition; as we will discuss later.

5.2 Multiplication

In order to perform floating-point multiplication using our proposed FMA, the addend must be set to zero. Also, the exponent of addend must be set to the maximum exponent in order to gurantee that the resulting exponent is that of the multiplication result. Hence, the addend is set to $C = 0 \times 10^{Emax}$. Also, for a correct sign calculation , a signal that indicates a multiply operation should be high; this avoids the effect of the addend sign on the final sign.

There are also some blocks that are not required in case of multiplications such as the leading zero anticipator and the decimal carry save adder. These blocks can be bypassed in case of multiplication.

5.3 Fixed Latency Pipelining

The number of pipeline stages are chosen to balance the delay of each stage and to minimize the required number of flip-flops. In this section, all operations take the same number of clock cycles (i.e. stages).

5.3.1 3-Stages Pipeline

As shown in Figure 5.1, the design in pipelined in three stages. At this depth of pipelining, the multiplier tree can fit in one stage. This reduces the latching overhead required to latch the internal signals in the carry save adder tree. The second stage contains the decimal carry save adder, the leading zero anticipator and the R/L Shifter. Finally, the combined add/round and the rounding set-up modules are in the third stage.

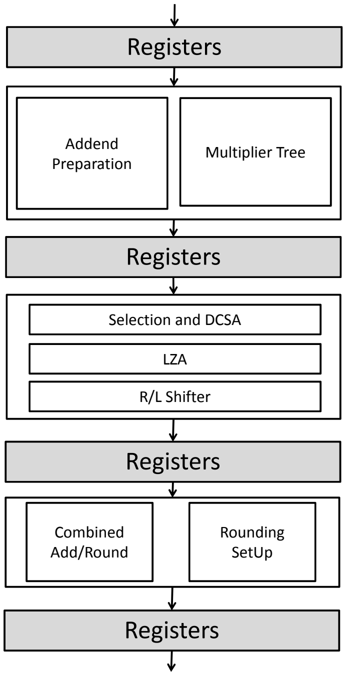


Figure 5.1: 3-Stages Pipeline

5.3.2 6-Stages Pipeline

In order to enhance the throuput, the depth of the pipeline is extended to 6 stages; as shown in Figure 5.2. At this depth, the multiplier tree has to be divided. It is divided such that the partial product generation and the decimal counters used at the beginning of the reduction are placed at the first stage. Then, the carry save adder tree which reduces the partial products is placed at the second stage with the addend preparation. This avoids the complexity of partitioning the carry save adder tree; since it is very hard to balance the delay in the different branches and also it requires large latching overhead. Moreover, partitioning after the decimal counters reduces the latching overhead compared to partitioning after the partial product generation directly.

The addend is not processed at the first stage; this gives an advantage for the microprocessor of only two read ports. Hence, it can read the addend after once cycle of reading the multiplier and the multiplicand without needing to wait for the addend to start the operation. In other words, the latency of reading the third operand is hided by the first stage of the multiplier at the first cycle.

Other blocks are partitioned to balance the delay between the different stages of the pipeline.

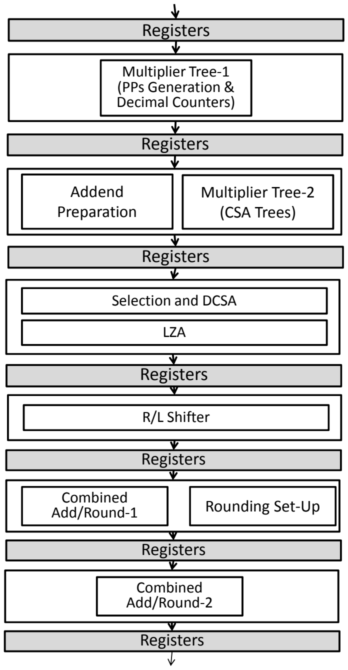


Figure 5.2: 6-Stages Pipeline

5.4 Variable Latency Pipelining

5.4.1 6-Stages Pipeline

In this pipeline, the fused multiply-add operation takes 6 cycles; however, the addition and multiplication operations take only 5 cycles. The addition bypasses the first stage and the multiplication bypasses the third stage. At the end of the second stage, a mux selects between the multiplication result of the multiplier tree in case of FMA and multiplication operations, and the operand *B* in case of addition operation. Also, the multiplication leading zero count is anticipated at the first stage using the leading zero count of both the multiplier and the multiplicand. This allows the multiplication operation to bypass the third stage that contains the leading zero anticipator.

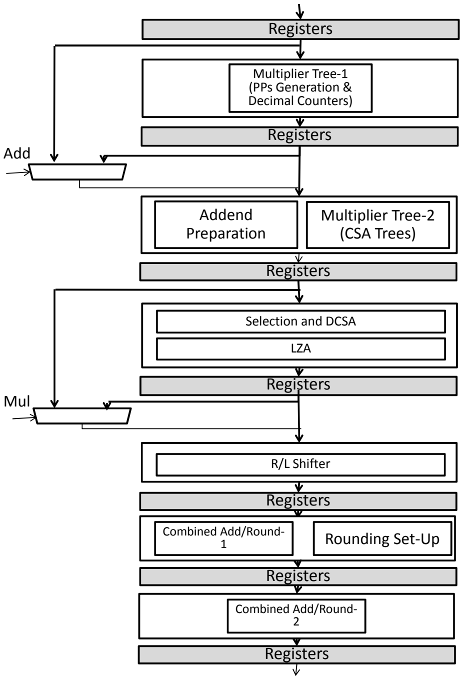


Figure 5.3: 6-Stages Variable Latency Pipeline

In this chapter, we extend our proposed FMA module to an arithmetic unit that perform floating point addition, multiplication and fused multiply-add operations. The arithmetic unit is pipelined into variable depths to enhance its throughput. The testing and synthesis results are presented in the next chapter.

Chapter 6

Results and Future Work

In this chapter, we present the testing and the synthesis results of the proposed implementations. At the end of this chapter, we suggest some points as a future work to extend or enhance our proposal in this thesis.

6.1 Verification

The verification of floating-point designs is very challenging due to the large test space and the numerous corner cases. The test space for the three operands' operation with 64 bit precision is about $2^{3*(64)+4}$ including the rounding modes (3-bits) and the operation (1-bit). Two main techniques are adopted to verify the designs of FP operations; these are: formal verification technique to verify the IEEE compliance of gate level designs [78], and the simulation verification technique based on coverage models[79].

The verification is performed using the simulation based coverage models proposed by Sayed-Ahmed [79]. They developed three separate engines to solve fused multiply add models, addition-subtraction models and multiplication models for 64-bit decimal format. and generate test vectors consistent with these models.

Operation	Number of Test Vectors
Fused Multiply Add	502066
Addition	136340
Multiplication	96845
Total	735251

Table 6.1: Number of test vectors applied for each operation

The generated test vectors are used to verify the corner cases of the three operations in different designs. The test vectors were efficient in discovering bugs in tested designs of IBM [26], Intel[27], SillMinds[39, 37, 21] and other Cairo University reaseach designs [32].

Table 6.1 shows the number of test vectors applied to the design for each operation. The design passes all these test vectors correctly.

6.2 Synthesis Results

The designs were synthesized using Design Compiler tool on a 65nm technology low power kit (TSMC65LP). The tool is allowed to flatten the design and the target of optimization was the delay.

6.2.1 Combinational FMA

The synthesis results in the area-delay curve shown in Figure . Since we mainly target a high performance architecture, the selected point is that of minimum delay. Hence, the delay of the FMA is $5.4 ns$ with an area equals to $155099.881932 \mu m^2$.

Figure

6.1 shows the percentage of the delay in each block on the critical path. While, Figure 6.2 shows the area profiling of the FMA.

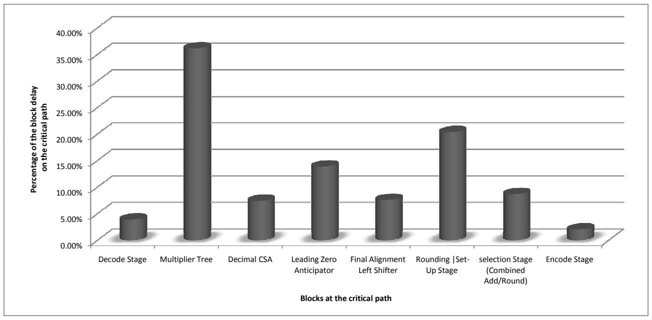


Figure 6.1: Delay of Each Block on the Critical Path

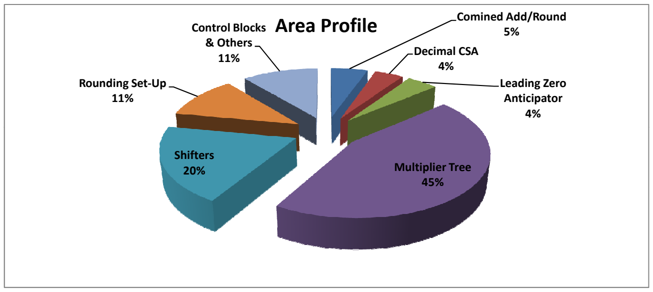


Figure 6.2: Area Profile for the Building Blocks of the FMA

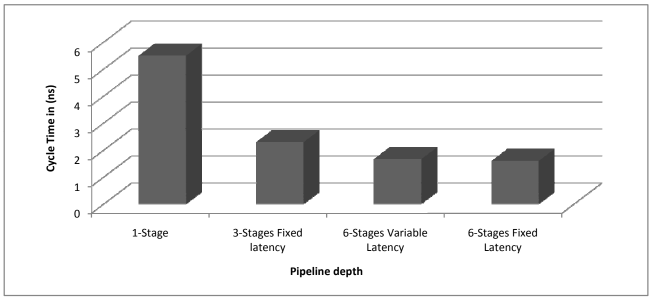


Figure 6.3: Area

6.2.2 Pipelined Arithmetic Unit

The single stage pipeline can operate on a frequency 182 MHz. The fixed latency 3-stage pipeline can have a minimum cycle time of 2.3 ns. The bottleneck is the first stage which contains the multiplier tree. The fixed latency 6-stage pipeline can operate up to 625 MHz. It is also constrained by the carry save adder trees in the reductionb part of the multiplier tree. It is placed at the second stage. The variable latency 6-stage pipleine can reach to 600 MHz with the addition and multiplication perfomed on 5 cycles only. Figure 6.3 shows the cycle time of each pipeline depth.

6.3 Comparison

6.3.1 FMA

Table 6.3.1 compares the availabe decimal FMA implementation with the one proposed. Although, both designs have a shortage in a complete and correct functionality, our propsal gives the best performance. It gives 17% improvement in the delay of the proposal in [39] and 7% improvement relative to [24].

Design	Normalized Delay	Normalized Area	Functionality
Decimal FMA [24]	1.08	4.47 (Binary & Decimal)	Incomplete
Decimal FMA [39]	1.2	0.8	~30% error in test vectors of [79]
Proposed FMA	1	1	Passed all test vectors

Table 6.2: Comparison of Different FMA designs

6.4 Future Work

Our future work will be focused on the following issues:

- Increading the pipeline depth to operate at higher frequencies suitable for general purpose processors.

- Extending the FMA and the floating-point arithmetic unit to work on 128-bit precision.

- Extending the functionality of the floating-point arithmetic unit to include all the decimal floating point operations in the standard.

- Including the floating-point arithmetic unit in the CU-Ultra SPARC Processor (Cairo University Ultra-Spac).

Bibliography

[1] T. Dantzig, *Number, the Language of Science*. The Macmillan Corporation, 1930.

[2] P. E. Ceruzzi, *A History of Modern Computing*. The MIT Press, 2003.

[3] M. ibn Musa Al-Khawarizmi, *The Keys of Knowledge*. around 830 C.E.

[4] A. G. Bromley, "Charles Babbage's analytical engine, 1838," *Annals of the History of Computing*, vol. 4, pp. 196 –217, july-sept. 1982.

[5] H. Goldstine and A. Goldstine, "The electronic numerical integrator and computer (eniac)," *Annals of the History of Computing, IEEE*, vol. 18, pp. 10 –16, spring 1996.

[6] A. W. Burks, H. H. Goldstine, and J. von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," tech. rep., Institution for Advanced Study, Princeton, 1946.

[7] W. Bouchholz, "Fingers or fists ? (the choice of decimal or binary representation)," *Communications of the ACM*, vol. 2, pp. 3–11, 1959.

[8] M. Cowlishaw, "Decimal floating-point: algorism for computers," in *Computer Arithmetic, 2003. ARITH-16 2003. 16th IEEE Symposium on*, pp. 104 – 111, june 2003.

[9] A. Vazquez, *High Performance Decimal Floating Point Units*. PhD thesis, Universidade de Santiago de Compostela, 2009.

[10] C. Baugh and B. Wooley, “A two’s complement parallel array multiplication algorithm,” *Computers, IEEE Transactions on*, vol. C-22, pp. 1045 – 1047, dec. 1973.

[11] L. Dadda, “Some schemes for parallel multipliers,” *Alta Frequenza*, vol. 34, pp. 349–456, 1965.

[12] P. M. Kogge and H. S. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence equations,” *Computers, IEEE Transactions on*, vol. C-22, pp. 786 –793, aug. 1973.

[13] R. H. Larson, “High-speed multiply using fused input carry-save adder,” *IBM technology Disclosure Bulletin*, vol. 16, pp. 2053–2054, 1973.

[14] L. Rubinfield, “A proof of the modified Booth’s algorithm for multiplication,” *Computers, IEEE Transactions on*, vol. C-24, pp. 1014 – 1015, oct. 1975.

[15] *INTEL, 8080/8085 Floating-Point Arithmetic Library User’s Manual*. Intel Corporation, 1979.

[16] A. Heninger, “Zilog’s Z8070 floating point processor,” *Mini Micro Systems*, vol. 40, pp. 1–7, 1983.

[17] “IEEE standard for radix-independent floating-point arithmetic,” *ANSI/IEEE Std 854-1987*, pp. 0–1, 1987.

[18] “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std 754-2008*, pp. 1–58, 29 2008.

[19] "Telco Benchmark for a telephone company billing application."

[20] L.-K. Wang, C. Tsen, M. Schulte, and D. Jhalani, "Benchmarks and performance analysis of decimal floating-point applications," in *Computer Design, 2007. ICCD-25 2007. 25th International Conference on*, pp. 164 –170, oct. 2007.

[21] H. Fahmy, R. Raafat, A. Abdel-Majeed, R. Samy, T. ElDeeb, and Y. Farouk, "Energy and delay improvement via decimal floating point units," in *Computer Arithmetic, 2009. ARITH-19 2009. 19th IEEE Symposium on*, pp. 221 –224, june 2009.

[22] C. Webb, "IBM z10: The next-generation mainframe microprocessor," *Micro, IEEE*, vol. 28, pp. 19 –29, march-april 2008.

[23] E. Schwarz and S. Carlough, "Power6 decimal divide," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pp. 128 –133, july 2007.

[24] P. K. Monsson, "Combined binary and decimal floating-point unit," Master's thesis, Technical University of Denmark, 2008.

[25] Sun MicroSystems, "Sun BigDecimal Library."

[26] "IBM decnumber Library."

[27] "Intel decimal floating-point math library."

[28] J. Thompson, N. Karra, and M. Schulte, "A 64-bit decimal floating-point adder," in *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pp. 297 – 298, feb. 2004.

[29] L.-K. Wang and M. Schulte, "Decimal floating-point adder and multifunction unit with injection-based rounding," in *Computer Arithmetic,*

2007. ARITH-18 2007. 18th IEEE Symposium on, pp. 56 –68, june 2007.

[30] L.-K. Wang and M. Schulte, “A decimal floating-point adder with decoded operands and a decimal leading-zero anticipator,” in *Computer Arithmetic, 2009. ARITH-19 2009. 19th IEEE Symposium on*, pp. 125 –134, june 2009.

[31] A. Vazquez and E. Antelo, “Conditional speculative decimal addition,” in *the 7th Conference of Real Numbers Comput.*, 2006.

[32] K. Yehia, H. Fahmy, and M. Hassan, “A redundant decimal floating-point adder,” in *Signals, Systems and Computers, 2010. ASOLIMAR-44 2010. 44th Asilomr Conference on*, pp. 1144 –1147, nov. 2010.

[33] A. Vazquez, E. Antelo, and P. Montuschi, “A new family of high.performance parallel decimal multipliers,” in *Computer Arithmetic, 2007. ARITH-18 2007. 18th IEEE Symposium on*, pp. 195 –204, june 2007.

[34] G. Jaberipur and A. Kaivani, “Improving the speed of parallel decimal multiplication,” *Computers, IEEE Transactions on*, vol. 58, pp. 1539 –1552, nov. 2009.

[35] M. Erle, M. Schulte, and B. Hickmann, “Decimal floating-point multiplication via carry-save addition,” in *Computer Arithmetic, 2007. ARITH-18 2007. 18th IEEE Symposium on*, pp. 46 –55, june 2007.

[36] B. Hickmann, A. Krioukov, M. Schulte, and M. Erle, “A parallel IEEE p754 decimal floating-point multiplier,” in *Computer Design, 2007. ICCD-25 2007. 25th International Conference on*, pp. 296 –303, oct. 2007.

[37] R. Raafat, A. Abdel-Majeed, R. Samy, T. ElDeeb, Y. Farouk, M. Elkhouly, and H. Fahmy, "A decimal fully parallel and pipelined floating point multiplier," in *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pp. 1800 –1804, oct. 2008.

[38] A. Vazquez, E. Antelo, and P. Montuschi, "Improved design of high-performance parallel decimal multipliers," *Computers, IEEE Transactions on*, vol. 59, pp. 679 –693, may 2010.

[39] R. Samy, H. Fahmy, R. Raafat, A. Mohamed, T. ElDeeb, and Y. Farouk, "A decimal floating-point fused-multiply-add unit," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pp. 529 –532, aug. 2010.

[40] L.-K. Wang and M. Schulte, "Decimal floating-point division using Newton-Raphson iteration," in *Application-Specific Systems, Architectures and Processors, 2004. Proceedings. 15th IEEE International Conference on*, pp. 84 – 95, sept. 2004.

[41] H. Nikmehr, B. Phillips, and C.-C. Lim, "Fast decimal floating-point division," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, pp. 951 –961, sept. 2006.

[42] A. Vazquez, E. Antelo, and P. Montuschi, "A radix-10 SRT divider based on alternative bcd codings," in *Computer Design, 2007. ICCD-25 2007. 25th International Conference on*, pp. 280 –287, oct. 2007.

[43] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, "Leading-zero anticipatory logic for high-speed floating point addition," *Solid-State Circuits, IEEE Journal of*, vol. 31, pp. 1157 –1164, aug 1996.

[44] A. Vazquez, J. Villalba, and E. Antelo, "Computation of decimal transcendental functions using the CORDIC algorithm," in *Computer Arithmetic, 2009. ARITH-19 2009. 19th IEEE Symposium on*, pp. 179 –186, june 2009.

[45] J. Harrison, "Decimal transcendentals via binary," in *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pp. 187 –194, june 2009.

[46] D. Chen, Y. Zhang, Y. Choi, M. H. Lee, and S.-B. Ko, "A 32-bit decimal floating-point logarithmic converter," in *Computer Arithmetic, 2009. ARITH-19 2009. 19th IEEE Symposium on*, pp. 195 –203, june 2009.

[47] R. Tajallipour, D. Teng, S.-B. Ko, and K. Wahid, "On the fast computation of decimal logarithm," in *Computers and Information Technology, 2009. ICCIT '09. 12th International Conference on*, pp. 32 –36, dec. 2009.

[48] A. Y. Duale, M. H. Decker, H.-G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal floating-point in z9: An implementation and testing perspective," *Research and Development, IBM Journal of*, vol. 51, pp. 217 –227, jan. 2007.

[49] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "IBM POWER6 microarchitecture," *Research and Development, IBM Journal of*, vol. 51, pp. 639 –662, nov. 2007.

[50] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. Dilullo, and M. Lanzerotti, "Design of the Power6 microprocessor," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 96 –97, feb. 2007.

[51] S. Oberman and M. Flynn, "Design issues in division and other floating-point operations," *Computers, IEEE Transactions on*, vol. 46, pp. 154 –161, feb 1997.

[52] R. Jessani and M. Putrino, "Comparison of single- and dual-pass multiply-add fused floating-point units," *Computers, IEEE Transactions on*, vol. 47, pp. 927 –937, sep 1998.

[53] C. Hinds, "An enhanced floating point coprocessor for embedded signal processing and graphics applications," in *Signals, Systems and Computers, 1999. ASOLIMAR-33 1999. 33rd Asilomr Conference on*, vol. 1, pp. 147 –151 vol.1, 1999.

[54] Y. Voronenko and M. Puschel, "Automatic generation of implementations for dsp transforms on fused multiply-add architectures," in *Acoustics, Speech, and Signal Processing, 2004. Proceedings. ICASSP 2004. IEEE International Conference on*, vol. 5, pp. V – 101–4 vol.5, may 2004.

[55] E. Linzer and E. Feig, "Implementation of efficient FFT algorithms on fused multiply- add architectures," *Signal Processing, IEEE Transactions on*, vol. 41, p. 93, jan 1993.

[56] A. Robison, "N-bit unsigned division via n-bit multiply-add," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 131 – 139, june 2005.

[57] E. Hokenek, R. Montoye, and P. Cook, "Second-generation RISC floating point with multiply-add fused," *Solid-State Circuits, IEEE Journal of*, vol. 25, pp. 1207 –1213, oct 1990.

[58] R. K. Montoye, E. Hokenek, and S. L. Runyon, "Design of the IBM RISC System/6000 floating-point execution unit," *Research and Development, IBM Journal of*, vol. 34, pp. 59 –70, jan. 1990.

[59] F. P. O'Connell and S. W. White, "POWER3: The next generation of PowerPC processors," *Research and Development, IBM Journal of*, vol. 44, pp. 873 –884, nov. 2000.

[60] T. Lang and J. Bruguera, "Floating-point fused multiply-add with reduced latency," in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 145 – 150, 2002.

[61] G. Even and P.-M. Seidel, "A comparison of three rounding algorithms for IEEE floating-point multiplication," *Computers, IEEE Transactions on*, vol. 49, pp. 638 –650, jul 2000.

[62] S. Oberman, H. Al-Twaijry, and M. Flynn, "The SNAP project: design of floating point arithmetic units," in *Computer Arithmetic, 1997. Proceedings., 13th IEEE Symposium on*, pp. 156 –165, jul 1997.

[63] M. Santoro, G. Bewick, and M. Horowitz, "Rounding algorithms for IEEE multipliers," in *Computer Arithmetic, 1989., Proceedings of 9th Symposium on*, pp. 176 –183, sep 1989.

[64] J. Bruguera and T. Lang, "Floating-point fused multiply-add: reduced latency for floating-point addition," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 42 – 51, june 2005.

[65] M. Xiao-Lu, “Leading zero anticipation for latency improvement in floating-point fused multiply-add units,” in *ASIC, 2005. ASICON-6 2005. 6th International Conference On*, vol. 1, pp. 53 – 56, oct. 2005.

[66] G. Li and Z. Li, “Design of a fully pipelined single-precision multiply-add-fused unit,” in *VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on*, pp. 318 –323, jan. 2007.

[67] L. Huang, L. Shen, K. Dai, and Z. Wang, “A new architecture for multiple-precision floating-point multiply-add fused unit design,” in *Computer Arithmetic, 2007. ARITH-18. 18th IEEE Symposium on*, pp. 69 –76, june 2007.

[68] Z. Qi, Q. Guo, G. Zhang, X. Li, and W. Hu, “Design of low-cost high-performance floating-point fused multiply-add with reduced power,” in *VLSI Design, 2010. VLSID ’10. 23rd International Conference on*, pp. 206 –211, jan. 2010.

[69] H. He, Z. Li, and Y. Sun, “Multiply-add fused float point unit with on-fly denormalized number processing,” in *Circuits and Systems, 2005. 48th Midwest Symposium on*, pp. 1466 –1468 Vol. 2, aug. 2005.

[70] M. Erle and M. Schulte, “Decimal multiplication via carry-save addition,” in *Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on*, pp. 348 – 358, june 2003.

[71] M. A. Erle, E. M. Schwarz, and M. J. Schulte, “Decimal multiplication with efficient partial product generation,” in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 21 – 28, june 2005.

[72] R. Kenney and M. Schulte, “High-speed multioperand decimal adders,” *Computers, IEEE Transactions on*, vol. 54, pp. 953 – 963, aug 2005.

[73] T. Lang and A. Nannarelli, “A radix-10 combinational multiplier,” in *Signals, Systems and Computers, 2006. Conference Record of the Fourteens Asilomar Conference on*, pp. 313 – 317, 28 - november 1, 2006.

[74] I. D. Castellanos and J. E. Stine, “Decimal partial product generation architectures,” in *Circuits and Systems, 2008. 51st Midwest Symposium on*, pp. 962 –965 Vol. 2, aug. 2008.

[75] J. Bruguera and T. Lang, “Leading-one prediction with concurrent position correction,” *Computers, IEEE Transactions on*, vol. 48, pp. 1083 –1097, oct 1999.

[76] A. Vazquez and E. Antelo, “A high-performance significand BCD adder with IEEE 754-2008 decimal,” in *Computer Arithmetic, 2009. ARITH-19 2009. 19th IEEE Symposium on*, pp. 135 – 144, june 2009.

[77] V. G. Oklobdzija, “An algorithmic and novel design of a leading zero detector circuit: Comparison with logic synthesis,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, pp. 124 – 128, march 1994.

[78] O. Leary, X. Zhao, R. Gerth, C. Johan, and H. Seger, “Formally verifying IEEE compliance of floating-point hardware,” *Intel Technology Journal*, vol. 40, pp. 200–200, Jan 1999.

[79] A. Sayed-Ahmed, H. Fahmy, and M. Hassan, “Three engines to solve verification constraints of decimal floating-point operation,” in *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record*

of the Forty Fourth Asilomar Conference on, pp. 1153 –1157, nov. 2010.

[80] M. Aharoni, S. Asaf, R. Maharik, I. Nehama, I. Nikulshin, and A. Ziv, "Solving constraints on the invisible bits of the intermediate result for floating-point verification," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, pp. 76 – 83, june 2005.

[81] M. Aharoni, R. Maharik, and A. Ziv, "Solving constraints on the intermediate result of decimal floating-point operations," in *Computer Arithmetic, 2007. ARITH-18 2007. 18th IEEE Symposium on*, pp. 38 –45, june 2007.

[82] B. Fu, A. Saini, and P. Gelsinger, "Performance and microarchitecture of the i486 processor," in *Computer Design: VLSI in Computers and Processors, 1989. ICCD '89. Proceedings., 1989 IEEE International Conference on*, pp. 182 –187, oct 1989.

[83] G. Gerwig, H. Wetter, E. M. Schwarz, J. Haess, C. A. Krygowski, B. M. Fleischer, and M. Kroener, "The IBM eServer z990 floating-point unit," *Research and Development, IBM Journal of*, vol. 48, pp. 311 –322, may 2004.

[84] C. Hinds and D. Lutz, "A small and fast leading one predictor corrector circuit," in *Signals, Systems and Computers, 2005. ASOLIMAR-39 2005. 39th Asilomr Conference on*, pp. 1181 – 1185, 28 - november 1, 2005.

[85] E. Hokenek and R. K. Montoye, "Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit," *Research and Development, IBM Journal of*, vol. 34, pp. 71 –77, jan. 1990.

[86] T. R. S. J. J. J. Bradley, B. L. Stoffers and M. Widen, “Apparatus for performing simplified decimal multiplication by stripping leading zeroes,” 1986.

[87] D. Jacobsohn, “A suggestion for a fast multiplier,” *Electronic Computers, IEEE Transactions on*, vol. EC-13, p. 754, dec. 1964.

[88] R. Kenney and M. Schulte, “Multioperand decimal addition,” in *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pp. 251 – 253, feb. 2004.

[89] V. S. Negi and S. A. Tague, “Data processor having units carry and tens carry apparatus supporting a decimal multiply operation,” 1984.

[90] N. Quach and M. Flynn, “Leading one prediction : Implementation, generalization, and application,” tech. rep., Stanford University, 1991.

[91] R. Rogenmoser and L. O. Donnell, “Method and apparatus to correct leading one prediction,” 2002.

[92] J.-L. Sanchez, H. Mora, J. Mora, F. J. Ferrandez, and A. Jimeno, “An iterative method for improving decimal calculations on computers,” *Mathematical and Computer Modeling*, vol. 50, pp. 869–878, 2009.

[93] M. Schmookler and K. Nowka, “Leading zero anticipation and detection-a comparison of methods,” in *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pp. 7 –12, 2001.

[94] E. M. Schwarz, J. S. Kapernick, and M. F. Cowlishaw, “Decimal floating-point support on the IBM System z10 processor,” *Research and Development, IBM Journal of*, vol. 53, pp. 4:1 –4:10, jan. 2009.

[95] L.-K. Wang, *Processor Support for Decimal Floating Point Arithmetic*. PhD thesis, Univ. Wisconsin Madison, 2007.

[96] L.-K. Wang, M. Schulte, J. Thompson, and N. Jairam, “Hardware designs for decimal floating-point addition and related operations,” *Computers, IEEE Transactions on*, vol. 58, pp. 322 –335, march 2009.

[97] “IBM test suit-FPgen.”

[98] *The iAPX 286 Programmer’s Reference Manual*. Intel Corporation, 1985.