HIGH-SPEED CO-PROCESSORS BASED ON REDUNDANT NUMBER SYSTEMS

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy

in the Department of Electrical and Computer Engineering

University of Saskatchewan

Saskatoon, Saskatchewan, Canada

By

AMIR KAIVANI

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering

University of Saskatchewan

57 Campus Drive

Saskatoon, Saskatchewan

Canada

S7N 5A9

ABSTRACT

There is a growing demand for high-speed arithmetic co-processors for use in applications with computationally intensive tasks. For instance, Fast Fourier Transform (FFT) co-processors are used in real-time multimedia services and financial applications use decimal co-processors to perform large amounts of decimal computations.

Using redundant number systems to eliminate word-wide carry propagation within interim operations is a well-known technique to increase the speed of arithmetic hardware units. Redundant number systems are mostly useful in applications where many consecutive arithmetic operations are performed prior to the final result, making it advantageous for arithmetic co-processors. This thesis discusses the implementation of two popular arithmetic co-processors based on redundant number systems: namely, the binary FFT co-processor and the decimal arithmetic co-processor.

FFT co-processors consist of several consecutive multipliers and adders over complex numbers. FFT architectures are implemented based on fixed-point and floating-point arithmetic. The main advantage of floating-point over fixed-point arithmetic is the wide dynamic range it introduces. Moreover, it avoids numerical issues such as scaling and overflow/underflow concerns at the expense of higher cost. Furthermore, floating-point implementation allows for an FFT co-processor to collaborate with general purpose processors. This offloads computationally intensive tasks from the primary processor.

The first part of this thesis, which is devoted to FFT co-processors, proposes a new FFT architecture that uses a new Binary-Signed Digit (BSD) carry-limited adder, a new floating-point

BSD multiplier and a new floating-point BSD three-operand adder. Finally, a new unit labeled as Fused-Dot-Product-Add (FDPA) is designed to compute $AB \pm CD \pm E$ over floating-point BSD operands.

The second part of the thesis discusses decimal arithmetic operations implemented in hardware using redundant number systems. These operations are popularly used in decimal floating-point co-processors. A new signed-digit decimal adder is proposed along with a sequential decimal multiplier that uses redundant number systems to increase the operational frequency of the multiplier. New redundant decimal division and square-root units are also proposed.

The architectures proposed in this thesis were all implemented using Hardware-Description-Language (Verilog) and synthesized using Synopsys Design Compiler. The evaluation results prove the speed improvement of the new arithmetic units over previous pertinent works. Consequently, the FFT and decimal co-processors designed in this thesis work with at least 10% higher speed than that of previous works. These architectures are meant to fulfill the demand for the high-speed co-processors required in various applications such as multimedia services and financial computations.

ACKNOWLEDGMENTS

All research was sponsored by the Electrical and Computer Engineering department at the University of Saskatchewan and the Natural Science and Engineering Research Council (NSERC) of Canada. All the toolkits and standard cell libraries used in this research were provided by CMC Microsystems, Canada.

I would like to thank my supervisor Dr. Seok-Bum Ko, who provided inspiration and advice during my PhD program at the University of Saskatchewan. I would also like to thank Dr. Li Chen with whom I had my VLSI course in the first year of my PhD program. Special thanks to the committee members, Dr. Joseph E. Salt, Dr. Francis M. Bui and Dr. FangXiang Wu who helped improve the quality of the research and the thesis. In the end, I would like to thank my friends in the lab, specially L. Han who helped me a lot during the first two years of my PhD program.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

BCD	Binary Coded Decimal
BSD	Binary Signed Digit
CLA	Carry Look-ahead Adder
CSA	Carry Save Adder
DIF	Decimation in Frequency
DIT	Decimation in Time
DSP	Digital Signal Processing
DSSD	Decimal Septa Signed Digit
FA	Full Adder
FDPA	Fused Dot Product Add
FFT	Fast Fourier Transform
FMA	Fused Multiply Add
GSD	Generalized Signed Digit
HA	Half Adder
LZA	Leading Zero Anticipation
LZD	Leading Zero Detection
MCM	Multiple Constant Multiplier
MMCM	Merged Multiple Constant Multiplier
MSD	Most Significant Digit
PP	Partial Product
PPA	Partial Product Accumulation
PPG	Partial Product Generation
PPR	Partial Product Reduction
PRC	Partial Remainder Computation
QDS	Quotient Digit Selection
RDS	Root Digit Selection
RNS	Residue Number System
WBP	Weighted Binary Position

LIST OF PUBLICATIONS

Journals:

1. **Kaivani, A.** and S. Ko, "Floating-Point Butterfly Architecture Based on Binary Signed-Digit Representation," *IEEE Transactions on Very Large Scale Integration*, to appear, 2015.

2. **Kaivani, A.**, L. Han and S. Ko, "Improved design of high-frequency sequential decimal multipliers," *IET Electronics Letters*, Vol. 50, No. 7, pp. 558-560, 2014.

3. Han, L., **A. Kaivani** and S. Ko, "Area Efficient Sequential Decimal Fixed-point Multiplier," *Journal of Signal Processing Systems*, Vol. 75, No. 1, pp. 39-46, 2014.

4. **Kaivani, A.** and S. Ko, "Decimal Division Algorithms: The Issue of Partial Remainders," *Journal of Signal Processing Systems*, Vol. 73, No. 2, pp. 181-188, 2013.

5. **Kaivani, A.** and S. Ko, "Decimal SRT Square Root: Algorithm and Architecture," *Circuits, Systems and Signal Processing*, Vol. 32, No. 5, pp.2137-2150, 2013.

Conferences:

1. **Kaivani, A.** and S. Ko, "High-Speed FFT Processors Based on Redundant Number Systems," *IEEE International Symposium on Circuits and Systems (ISCAS'14)*, pp. 2237-2240, 2014.

2. **Kaivani, A.** and S. Ko, "Decimal signed digit addition using stored transfer encoding," *26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'13)*, pp.1-4, 2013.

3. **Kaivani, A.**, L. Chen and S. Ko, "High-frequency sequential decimal multipliers," *IEEE International Symposium on Circuits and Systems (ISCAS'12)*, pp. 3045-3048, 2012.

CHAPTER 1
INTRODUCTION & BACKGROUNDS

The need for high-speed processing greatly exceeds what general-purpose processors can handle. This is where, for instance, arithmetic co-processors can be used to offload computationally intensive tasks from the primary processor. Fast Fourier Transform (FFT) co-processors are used in real-time multimedia services and financial applications use decimal co-processors to perform large amounts of decimal computations.

A co-processor receives data from a general-purpose processor to execute time-consuming operations. After the co-processor is done processing the data, the results are sent back to the general-purpose processor. This approach saves time and offloads computationally intensive tasks from primary processors, therefore achieving higher overall performance. Of various available co-processors, the binary Fast Fourier Transform (FFT) co-processor and the decimal floating-point co-processor have received a lot of attention recently.

FFT circuitry consists of several consecutive multipliers and adders over complex numbers. Until recently, most FFT architectures used fixed-point arithmetic only, before FFTs based on floating-point operations became prominent [1, 2]. Using the IEEE-754-2008 standard [3] for floating-point arithmetic allows FFT co-processors to collaborate with general purpose processors.

Despite the fact that binary computer arithmetic improves processing speed and reduces hardware complexity, decimal computer arithmetic has recently been revived. The advantage of decimal computer arithmetic over its binary counterpart is that decimal arithmetic is capable of mirroring human computations (i.e., radix-10) and representing fractions precisely where binary

cannot (e.g., 0.2) [8]. Some applications, such as finance and banking, cannot tolerate a loss of precision; this is where decimal computer arithmetic is useful.

Decimal computer arithmetic can be implemented in hardware or software. The software implementation of decimal arithmetic operations with binary logic devices was widely used until IBM revealed an all-hardware implementation of decimal processors such as the POWER6 decimal processor [9]. Additionally, the IEEE 754-2008 [3] standard for floating-point arithmetic now supports the decimal hardware implementation. Hardware decimal arithmetic is used where high-speed computations are performed on large amounts of data.

1.1 Fast Fourier Transform (FFT)

N-point FFT computation is described in Eqn. 1.1, where $d(k)$ is the input and $W_N = e^{-2\pi i/N}$ is the complex twiddle factor.

$$X[n] = \sum_{k=0}^{N-1} d(k) \cdot W_N^{nk}; \quad n = 0,1, \dots, N-1 \tag{1.1}$$

As a result of Eqn. 1.1, the outputs are as follows:

$$\begin{aligned}
&X[0] = d(0)W_N^0 + d(1)W_N^{0\times1} + \cdots + d(N-1)W_N^{0\times(N-1)} \\
&X[1] = d(0)W_N^0 + d(1)W_N^{1\times1} + \cdots + d(N-1)W_N^{1\times(N-1)} \\
&\vdots \\
&X[k] = d(0)W_N^0 + d(1)W_N^{k\times1} + \cdots + d(N-1)W_N^{k\times(N-1)} \\
&\vdots \\
&X[N-1] = d(0)W_N^0 + d(1)W_N^{(N-1)\times1} + \cdots + d(N-1)W_N^{(N-1)\times(N-1)}
\end{aligned} \tag{1.2}$$

Consequently, implementation of Eqn. 1.1 requires N^2 complex multiplication plus $N(N-1)$ complex additions. Given that each complex multiplication (addition) includes four (zero) real multiplications and two (two) additions, in overall $4N^2$ real multiplications and $2N(2N-1)$ real additions are required to implement Eqn. 1.1.

Cooley and Tukey [10] proposed an efficient algorithm which makes hardware realization of Eqn. 1.1 much easier. This algorithm, as shown in Eqn. 1.3, decomposes Eqn. 1.1 into even and odd indices. Therefore, the *N*-input FFT computation is simplified to the computation of two (*N*/2)-input FFT (see Fig. 1.1). Continuing this decomposition leads to 2-input FFT block also known as "butterfly" unit.

$$\begin{aligned} X[n] &= \sum_{k=0}^{\frac{N}{2}-1} d(2k)\cdot W_N^{2nk} + \sum_{k=0}^{\frac{N}{2}-1} d(2k+1)\cdot W_N^{n(2k+1)} \\ &= \sum_{k=0}^{\frac{N}{2}-1} d(2k)\cdot W_N^{2nk} + W_N^n \sum_{k=0}^{\frac{N}{2}-1} d(2k+1)\cdot W_N^{2nk} \\ \xrightarrow{W_N^{2nk}=e^{\frac{-2\pi i}{N}2nk}=e^{\frac{-2\pi i}{N/2}nk}=W_{N/2}^{nk}} X[n] &= \sum_{k=0}^{\frac{N}{2}-1} d(2k)\cdot W_{N/2}^{nk} + W_N^n \sum_{k=0}^{\frac{N}{2}-1} d(2k+1)\cdot W_{N/2}^{nk} \\ \xrightarrow{W_N^{k+N/2}=-W_N^k} X\left[n+\frac{N}{2}\right] &= \sum_{k=0}^{\frac{N}{2}-1} d(2k)\cdot W_{N/2}^{nk} - W_N^n \sum_{k=0}^{\frac{N}{2}-1} d(2k+1)\cdot W_{N/2}^{nk} \end{aligned} \tag{1.3}$$

There are two FFT architectures, decimation in time (DIT) and decimation in frequency (DIF). The former, shown in Fig. 1.1, consists of a complex multiplication followed by complex add/sub operations, while the latter requires the add/sub operations prior to the multiplication.

There are other FFT algorithms in the literature such as radix-3 [11], radix-4 [12], mixed radix [13], split radix [14], convolution-based (e.g., Winograd [15] and Bluestein [16]), to name

a few. However, the radix-2 Cooley-Tukey algorithm is known as the most efficient algorithm for hardware implementation due to its simplicity.

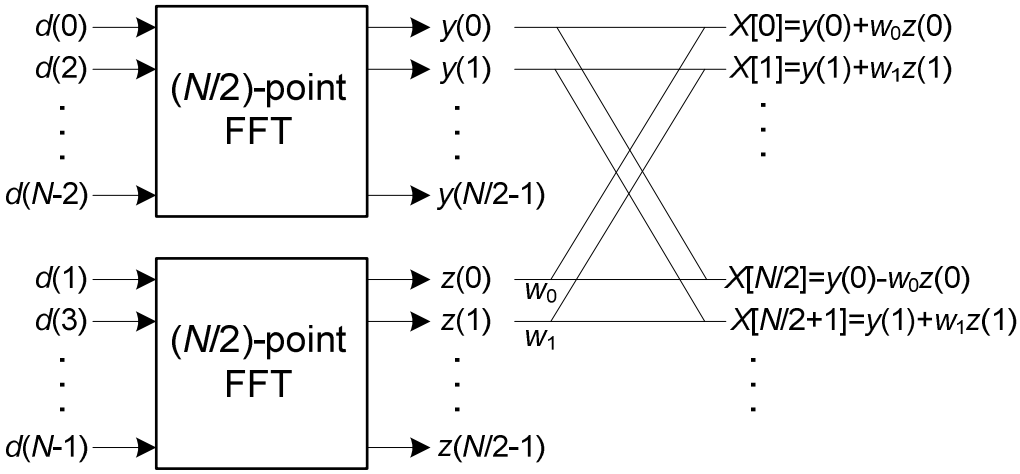


Figure 1.1: Implementation of an *N*-point FFT

Generally, *N*-point DIT FFT implementation using Cooley-Tukey algorithm can be summarized as:

- Number of stages = $\log_2 N$
- Number of twiddle factors = $N/2$
- Number of butterfly units at each stage = $N/2$
- Difference between indices of the upper and lower leg of a butterfly unit = $2^{stage-1}$

As an example, Fig. 1.2 illustrates the implementation of an 8-input DIT FFT.

1.2 Butterfly unit

Butterfly unit is actually a fused-multiply-add/sub (FMA) over complex operands. Fig. 1.3 depicts a DIT butterfly unit which consists of a complex multiplier, a complex adder and a complex subtractor.

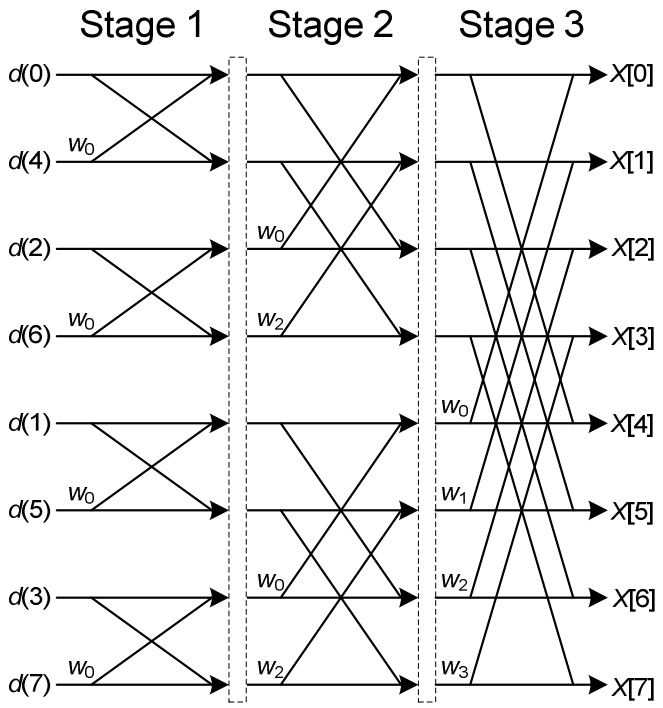


Figure 1.2: 8-input FFT using DIT butterfly

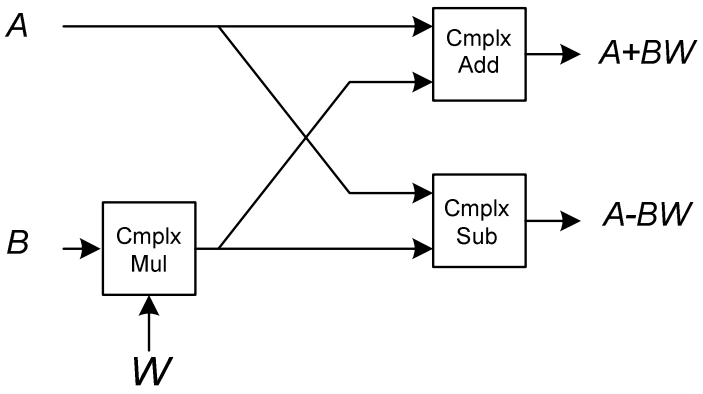


Figure 1.3: Butterfly Architecture (DIT)

A complex number A consists of one real and one imaginary component such that $A = d_{Re} + id_{Im}$. Complex addition/subtraction $\langle S_{Re}, S_{Im}\rangle = \langle A_{Re}, A_{Im}\rangle \pm \langle B_{Re}, B_{Im}\rangle$ includes two additions/subtractions over the real and imaginary components i.e., $S_{Re} = A_{Re} \pm B_{Re}$ and $S_{Im} = A_{Im} \pm B_{Im}$. Complex multiplication $\langle P_{Re}, P_{Im}\rangle = \langle A_{Re}, A_{Im}\rangle \times \langle B_{Re}, B_{Im}\rangle$ is performed as shown in Eqn. 1.4.

$$P_{Re} = A_{Re}B_{Re} - A_{Im}B_{Im}; \quad P_{Im} = A_{Re}B_{Im} + A_{Im}B_{Re} \tag{1.4}$$

According to Eqn. 1.4, a complex multiplication requires four multiplications and two additions/subtractions. However, Golub algorithm introduces another equation (Eqn. 1.5) for the computation of the real component P_{Re} which, in some cases, leads to lower cost [19].

$$P_{Re} = (A_{Re} + A_{Im})(B_{Re} - B_{Im}) + A_{Re}B_{Im} - A_{Im}B_{Re} \quad (1.5)$$

Given that $A_{Re}B_{Im}$ and $A_{Im}B_{Re}$ are already computed for P_{Im}, Golub algorithm requires three multiplications and five additions/subtractions. This leads to a lower cost than the conventional method, assuming a multiplier costs at least 3 times more than an adder. Therefore, there are two methods to implement a butterfly unit: 1) conventional 2) Golub's approach.

Fig. 1.4 shows the implementation of a DIT butterfly with expanded complex numbers using the conventional approach. Accordingly, it consists of four multipliers and six adders/subtractors. It should be noted that, given the constant values of twiddle factors (*W*), the multipliers are constant and can be implemented via a series of shifters and adders in lieu of the multiplier tree. Fig. 1.5 shows the implementation of a DIT butterfly unit based on the Golub's approach. Accordingly, it consists of three multipliers and nine adders/subtractors.

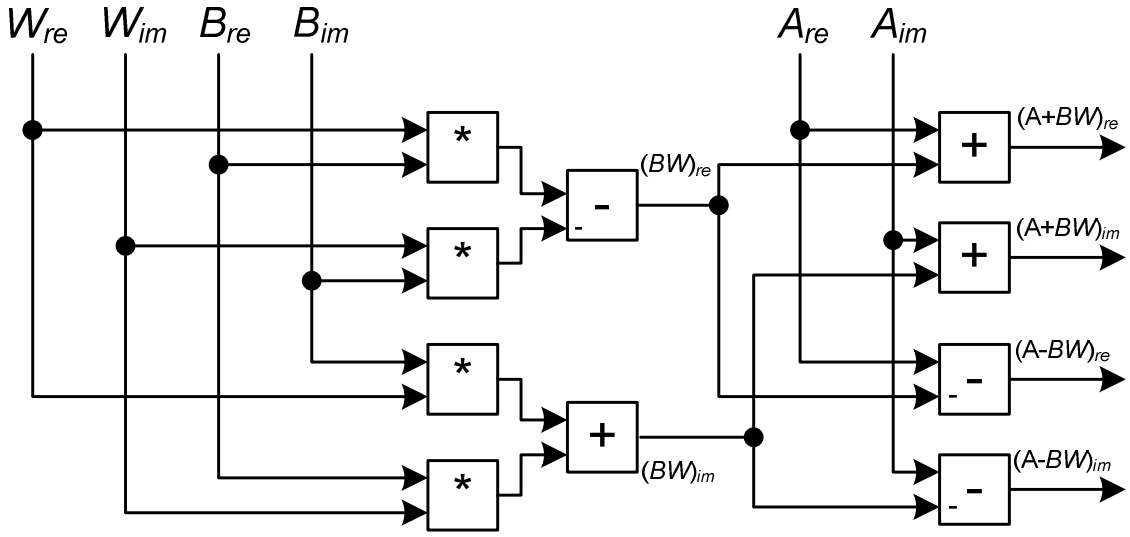


Figure 1.4: DIT Butterfly architecture using conventional approach

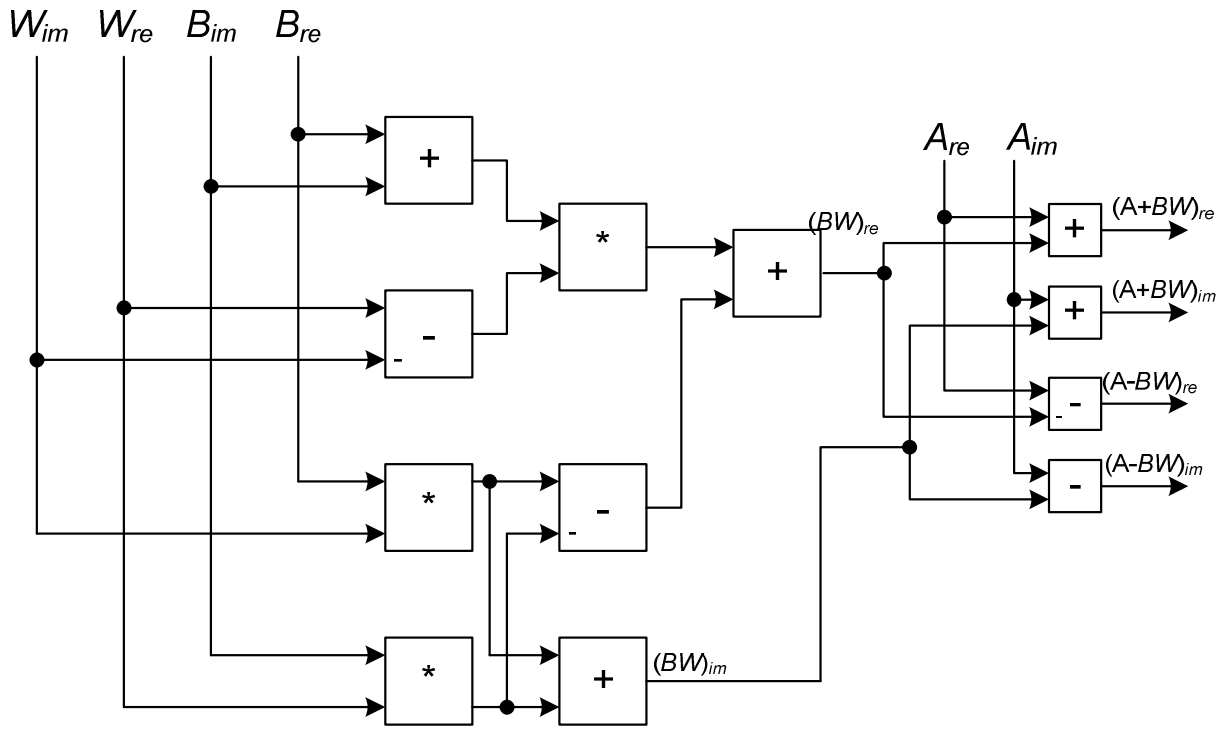


Figure 1.5: DIT Butterfly architecture using Golub's approach

1.3 Floating-Point Arithmetic

Floating-point arithmetic is being used, and preferred over fixed-point, in many applications due to the fact that it provides a large range of numbers and a high degree of precision. It is also common to be used in a variety of Digital Signal Processing (DSP) applications because it relieves the designer of numerical issues e.g., scaling, overflow, and underflow. A floating-point number, as represented in Eqn. 1.6, consists of four components; namely, *sign*, *significand*, *base* and *exponent*.

$$(-1)^{sign} \times significand \times (base)^{exponent} \tag{1.6}$$

The above four components for a single precision floating-point number, according to IEEE 754-2008 standard [3], take values as follows:

- $sign \in \{0,1\}$
- $significand$ is a 24 bit number $\in [1, 2 - 2^{23}]$
- $base = 2$
- $exponent$ is an 8 bit integer $\in [-126, 127]$

Given that *base* is always equal to two, it is only required to store the other three components. With the intention of storing these components efficiently, the following modifications are done:

1. It is always $(exponent + bias)$ stored. Given $bias = 127$, the stored value belongs to $[1, 254]$. The values 0 and 255 are reserved for special values.
2. Keeping $significand$ normalized (i.e., the most significant bit is always 1), allows for storing only 23 bits of the significand. The most significant bit is known as a hidden bit.

Therefore, a single precision floating-point number covers $[2^{-126}, 2^{128})$ and any number smaller than 2^{-126} is a *denormalized* number. Consequently, 0 cannot be represented in a normalized range and a special code is assigned to represent 0 i.e., $exponent + bias = 0$ and $significand = 0$.

IEEE 754-2008 standard [3] imposes special codes for the following special values:

- $\pm\mathbf{0}$: $exponent + bias = 0 \quad and\ significant = 0$
- $\boldsymbol{Denormalized\ number}$: $exponent + bias = 0 \quad and\ significand \neq 0$
- $\pm\infty$: $exponent + bias = 255\ and\ significand = 0$
- $\boldsymbol{Not\ a\ Number}$ $(\boldsymbol{NaN})$: $exponent + bias = 255\ and\ significant \neq 0$

Floating-point addition/subtraction consists of the following:

a) *Exponent difference*: Determine exponent difference Δ and the smaller exponent.

b) *Alignment shift*: Shift right (Δ positions) the significand of the number having a smaller exponent. The largest exponent is the result's exponent.
c) *Addition/subtraction over significands*: Determine and perform the actual operation; may need to swap the operands.
d) *Normalization shift*: Shift right 1 bit, in case of addition overflow. Detect the number of leading zeros and shift left, in case of subtraction, such that there is a new hidden 1 to the left of the binary point.
e) *Rounding*: Use extra bits (Round, Guard and Sticky) to round the result. This may lead to post normalization.
f) *Exponent adjustment*: Adjust the exponent to compensate for the shifts in d) and e).

In order to improve the speed of floating-point addition, dual-path architecture is usually used which separates the slow shifts (b and d) into different paths. This, however, requires parallel significand addition/subtraction modules.

Floating-point multiplication consists of the following:

a) *Multiplication of the significands*: fixed-point multiplication over the significands
b) *Addition of exponents*: The exponent of the result is determined by this addition.
c) *Normalization*: This involves leading-zero-detection (LZD) and shifting.
d) *Rounding*: Round the significand of the product according to standard rounding methods.
e) *Exponent adjustment*: The shifts in *d*) calls for this exponent adjustment

In floating-point fused multiply add (FMA) operation, *a* and *b* of the multiplication can be performed in parallel with *a* and *b* of the addition. Consequently, there is no need to use dual path architecture [17]. Moreover, common steps of addition and multiplication (i.e., normalization, rounding and exponent adjustment) are usually combined to save area and time.

1.4 Redundant Number Systems

Carry propagation is known as the main decelerator in digital arithmetic operations. Carry digits are the consequence of the fact that the value of the *i*th digit of the final result (e.g., sum in addition) can depend on the values of the operands in *i*th position and the less significant positions (0 to *i*–1).

There are some techniques to reduce the carry propagating latency such as Residue Number Systems (RNS) [7] and redundant number systems.

A number system, defined by radix r and digit-set $[\alpha, \beta]$, is redundant if and only if $\beta - \alpha + 1 > r$ [7]. Moreover, the redundancy index of that digit-set is defined as $\rho = \beta - \alpha + 1 - r$. Amongst available digit-sets for a specific redundant number system, minimally and maximally redundant digit-sets are the most popular ones due to their unique features. A minimally redundant digit-set is the one with $\rho = 1$; while a maximally redundant digit-set has the maximum cardinality represented with minimum number of bits. For example, the Binary signed-digit (BSD) representation with digit-set [-1, 1] is a redundant number system.

In addition to radix and digit-set values, the encoding used to represent a digit-set has a great impact on the performance and efficiency of a redundant number system. For example, Table 1.1 shows some encodings for *r*=10 and digit-set [0,10].

Assuming that the transfer digits are $t_i \in [-\lambda, \mu]$, the $(\lambda - \alpha \leq w_i \leq \beta - \mu)$ condition must be satisfied to guarantee no carry-propagation in Step *c* of Algorithm 1.1. In other words the maximum (minimum) value of a transfer digit plus interim sum must fit into digit-set $[\alpha, \beta]$.

Table 1.1: Encodings for a Decimal Digit

Encoding Weights	Dot notation
8-4-2-1	● ● ● ●
8-4-2-1-1	● ● ● ● ●
4-2-2-1-1	● ● ● ● ●

Algorithm 1.1: Carry-free addition

Inputs: Two redundant numbers $X: x_{n-1} \cdots x_1 x_0$ and $Y: y_{n-1} \cdots y_1 y_0$.

Output: Addition result represented in redundant format $S: s_n s_{n-1} \cdots s_1 s_0$.

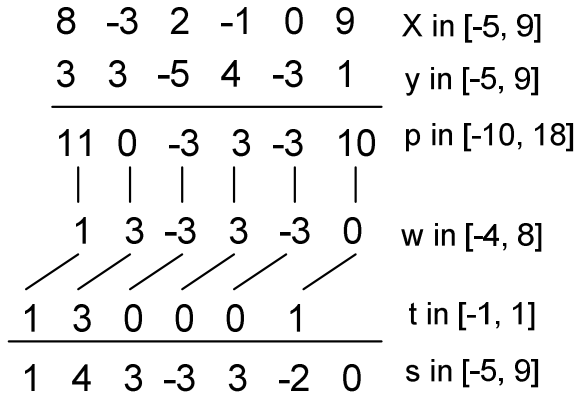
Perform followings for $0 \leq i < n$, in parallel.

a) Compute position sums $p_i = x_i + y_i$.

b) Divide p_i into a transfer digit t_{i+1} and interim sum w_i such that $w_i = p_i - rt_{i+1}$.

c) Compute the final result as $s_i = w_i + t_i$. ■

Example 1.1: Carry-free addition as $s = x + y$, in redundant digit-set [-5, 9].



It is proven in [18] that carry-free addition is possible iff one of the following conditions is satisfied:

- $r \geq 3, \rho \geq 3$
- $r \geq 3, \rho = 2, \alpha \neq 1, \beta \neq 1$

Therefore, carry-free addition is not applicable to binary signed-digit representation with digit-set $[-1,1]$. In this case a carry-limited addition is available [7].

Algorithm 1.2: Carry-limited addition

Inputs: Two redundant numbers $X: x_{n-1} \cdots x_1 x_0$ and $Y: y_{n-1} \cdots y_1 y_0$.

Output: Addition result represented in redundant format $S: s_n s_{n-1} \cdots s_1 s_0$.

Perform following steps for $0 \leq i < n$, in parallel.

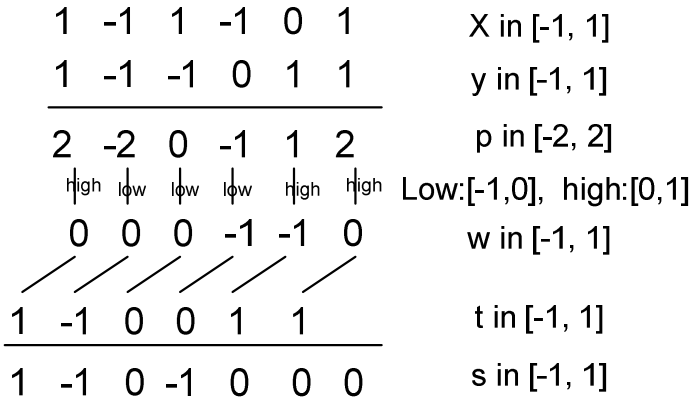
a) Compute position sums $p_i = x_i + y_i$.

b) Compare p_i to a constant to determine whether $e_{i+1} = low$ or $high$ (e_{i+1} is a binary range estimate for t_{i+1}).

c) Given e_i, divide p_i into a transfer digit t_{i+1} and interim sum w_i such that $w_i = p_i - r t_{i+1}$.

d) Compute the final result as $s_i = w_i + t_i$. ■

Example 1.2: Carry-limited addition as *s* = *x* + *y*, in binary signed-digit [-1, 1].



It should be noted that in Example 1.2 $p = -1(1)$ is kept intact when the incoming carry is in [0,1] ([−1,0]), so as to guarantee no further carry-propagation.

1.5 Motivation of Research

Carry propagation decelerates digital arithmetic operations. Redundant number systems are the most popular technique to overcome this challenge. A number system, defined by radix *r* and digit-set [α, β], is redundant if and only if $\beta - \alpha + 1 > r$.

Redundant number systems eliminate word-wide carry propagation within interim operations. However, the conversion from a redundant format to a non-redundant one requires carry-propagation. This makes redundant number systems mostly useful in applications where many consecutive arithmetic operations are performed prior to the final result making it a suitable technique to use in computer arithmetic co-processors. This thesis discusses the implementation of the binary floating-point FFT co-processor, and the decimal arithmetic co-processor.

Floating-point FFT circuitry consists of several consecutive multipliers and adders over complex numbers. The main advantage of floating-point over fixed-point arithmetic is the wide dynamic range it introduces. The main drawback of floating-point operations is their slowness compared with their fixed-point counterparts. One way to speed up floating-point arithmetic is merging several operations in a single floating-point unit to save delay, area and power consumption [2]. Previous works on floating-point FFT architectures have used this technique [1, 4, 5] to design a dot-product unit, to compute $(A + B) \times (C + D)$, so as to gain performance improvement.

Using redundant number systems [6] is another well-known way of improving the speed of floating-point arithmetic units, where there is no word-wide carry propagation within interim operations. The conversion from non-redundant to redundant format is a carry-free operation; however, the reverse conversion requires carry propagation [7].

Decimal computer arithmetic is inherently slower than its binary counterpart, since it is working based on a non-power-of-two radix. Likewise for the FFT co-processor, redundant number systems are very helpful in increasing the speed of the decimal arithmetic co-processor.

1.6 Objectives of the Thesis

The main objective of this thesis is to design a high-speed co-processor based on redundant number systems. In particular, the architectures of the two of most commonly used co-processors will be redesigned based on redundant number systems to achieve improved performance.

The first part of this thesis investigates the advantages and costs of designing high-speed floating-point FFT architectures using redundant number systems. New architectures are proposed and compared to previous works.

The second part is devoted to proposing decimal arithmetic co-processors with architectures based on redundant number systems comparing them with previous works. A complete decimal arithmetic unit is designed accordingly, with four basic decimal arithmetic operations: addition, subtraction, multiplication and division. An architecture based on redundant number systems is also proposed for computing decimal square-root.

1.7 Novelties

Although there are other works on the use of redundant floating-point number systems, they are not optimized for FFT architectures that require both a redundant floating-point multiplier and an adder. The novel techniques used in the new floating-point FFT architecture include:

- All significands are represented in binary signed-digit (BSD) format and the corresponding carry-limited adder is designed.
- Design of floating-point constant multipliers for operands with BSD significands.
- Design of floating-point three-operand adders for operands with BSD significands.

- Design of floating-point Fused-Dot-Product-Add units (i.e., $AB \pm CD \pm E$) for operands with BSD significands.

The novel techniques used in the new decimal floating-point arithmetic units include:

- Design of a decimal redundant adder based on signed-digit and stored transfer encoding.
- Design of a high-speed sequential decimal multiplier based on unconventional representations.
- Design of a decimal divider with redundant representation of the quotient and partial remainders.
- Design of a high-speed decimal square-root based on redundant number representation.

1.8 Organization of the Thesis

The rest of the thesis is divided into two main parts. Part I discusses the details of the new floating-point FFT co-processor designed based on redundant number systems. Previous works on floating-point FFT co-processors are presented in Chapter 2. In Chapter 3 (partially published in [63, 64]), the new FFT architectures are explained in detail. These architectures are designed based on the new floating-point redundant multiplier and the new Binary signed-digit (a redundant representation) three-operand adder. Chapter 4 includes the evaluation results of the new floating-point arithmetic units used in the new FFT co-processor. These results are compared with those of previous works.

Part II of this thesis presents the details of the new decimal arithmetic units designed based on redundant number systems. Chapter 5 (partially published in [65]) presents the details of the new redundant decimal adder. The new sequential decimal multiplier, based on an unconventional redundant representation, is explained in Chapter 6 (partially published in [66,

67, 68]). In Chapter 7 (partially published in [69]), a new decimal divider that represents quotient and partial remainders in a redundant format is proposed. Chapter 8 (partially published in [70]) presents the new decimal square-root unit. Finally, Chapter 9 is devoted to the conclusions and future works.

Part I

FFT Co-Processors

CHAPTER 2
PREVIOUS WORKS ON FFT PROCESSORS

This chapter discusses the previous works related to floating-point FFT processors which includes butterfly architectures and complex multipliers.

2.1 Floating-Point Complex Multipliers [19]

This work [19] focuses on the design of a floating-point complex multiplier with the help of fused floating-point arithmetic.

Conventional and Golub's method are two approaches based on which the paper [19] designs floating-point complex multipliers. As is also mentioned in previous section, Golub's method requires fewer multipliers but more adders than the conventional approach. Authors opt for conventional method, due to the fact that the latency and cost of floating-point adders are close to those of floating-point multipliers. Consequently, Golub's method is not recommended for floating-point implementations.

The paper [19] proposes a fused-dot-product unit which performs two multiplications and then adds the products i.e., $AB + CD$. With this unit, efficiency is achieved due to the elimination of the interim rounding and normalization of the fused-dot-product unit [19]. A conventional complex multiplier is then designed using two fused-dot-product units.

Floating-point fused-add-subtract is another unit designed in the paper [19]. This unit computes both floating-point sum and difference of two input operands. It combines the common parts of floating-point addition and subtraction and hence significantly lower power/area cost.

The third floating-point unit discussed in [19] is a fused-dot-product-add/subtract unit, which is a combination of the two previous units. This new unit has four inputs (A, B, C, D) and two outputs $X = AB + CD$ and $Y = AB - CD$. A Golub's complex multiplier is then designed using the latter two units.

Taking advantage of the aforementioned fused units, the paper [19] concludes that "Golub's method results in a modest increase in complexity and power consumption and a large increase in delay relative to the conventional method. This is true even when fused implementations are used" [19].

2.2 Floating-Point Fused Butterfly Arithmetic [1]

This work [1] improves the performance of butterfly unit and hence FFT processors by proposing two fused floating-point operations, namely, Dot-Product (Fig. 2.1) and Add-Subtract (Fig. 2.2).

The Dot-Product unit, compute *AB+CD*, consists of the following operations:

- Two floating-point multipliers, perform in parallel
- Alignment Shift
- (4:2) Compressor
- Carry-propagating adder, performs in parallel with Leading-Zero-Detector (LZD)
- Normalization & Rounding

The fused Add-Subtract unit, compute *A±B*, consists of the following operations:

- Exponent difference block, perform in parallel with MUXes to select significands
- Alignment Shift
- A subtraction and an addition perform in parallel
- Exponent adjustment, perform in parallel with rounding and normalization
- MUXes to select Add/Subtract results

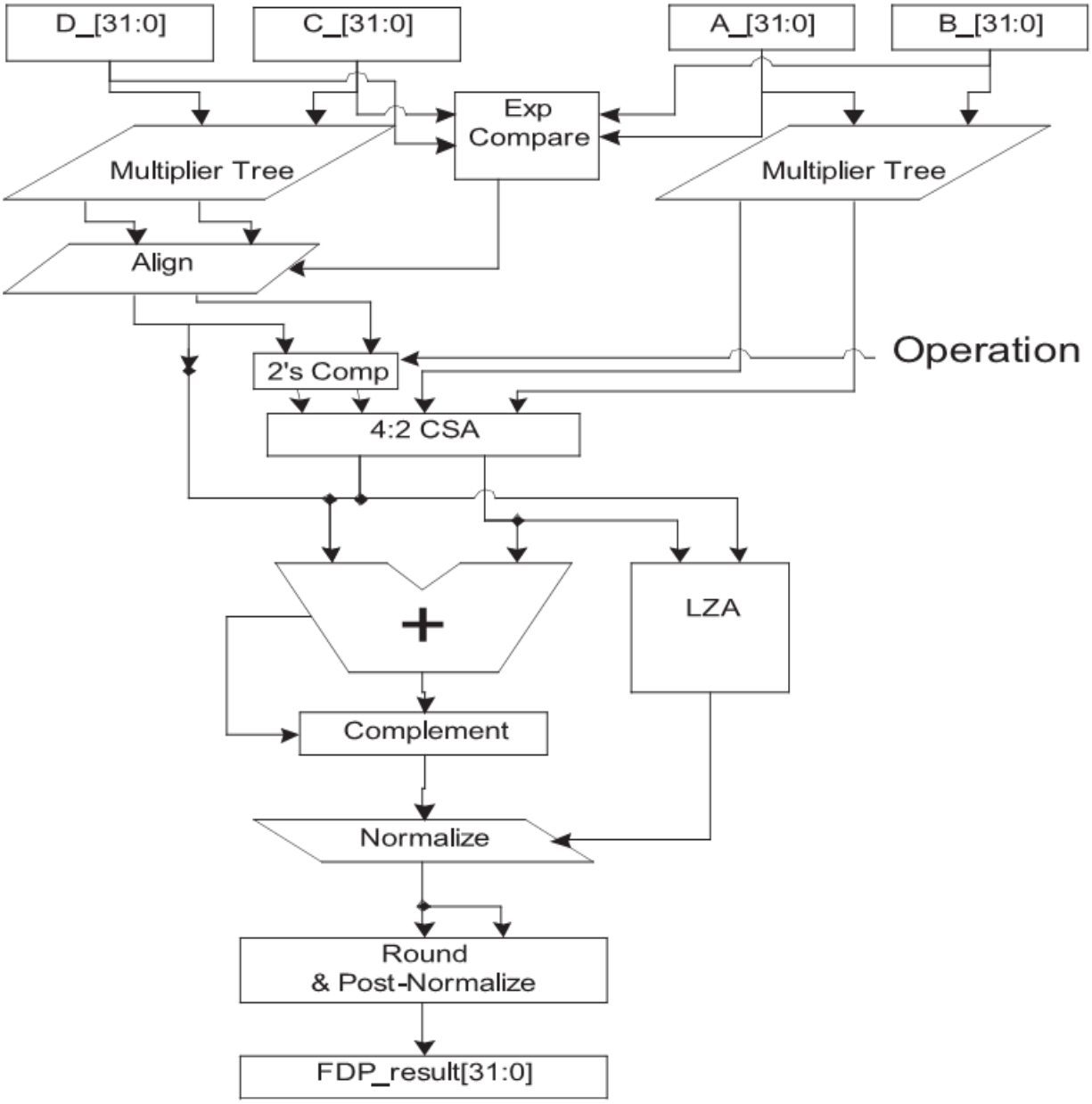


Figure 2.1: Floating-point Dot-Product unit [1]; Result = AB±CD

Next a radix-2 (Fig. 2.3) butterfly (Decimation in Frequency) unit is designed using Dot-Product and fused ADD-Subtract modules. It has been shown that the proposed radix-2 butterfly architecture requires 35% lower area cost and is 15% faster than its discrete (not fused) implementation.

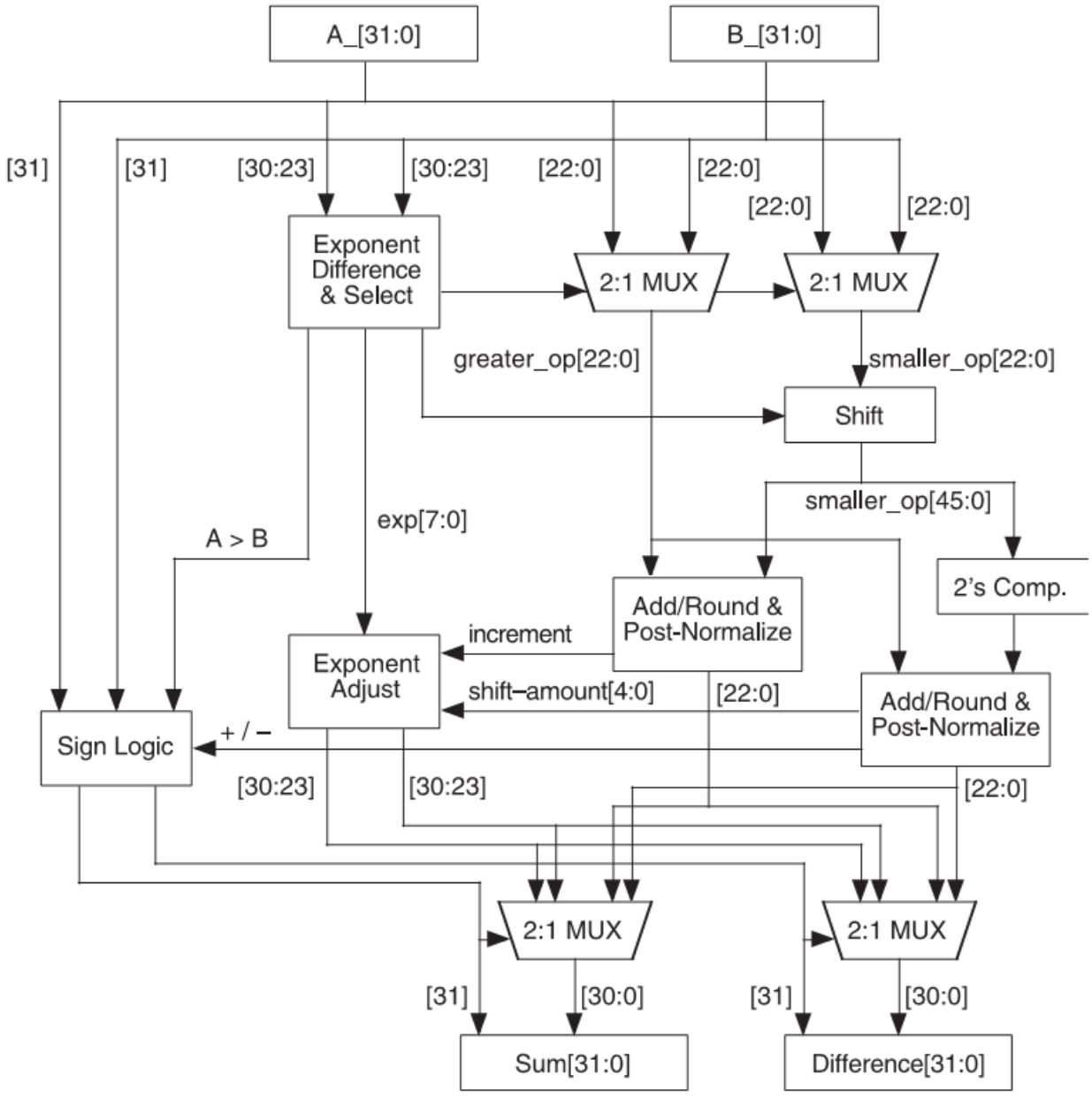


Figure 2.2: Floating-point fused Add-Subtract unit [1]

Moreover, a radix-4 butterfly (Decimation in Time) unit is designed using Dot-Product and fused Add-Subtract modules. It has been shown that the proposed radix-4 butterfly architecture requires 26% lower area cost and is 13% faster than its discrete (not fused) implementation.

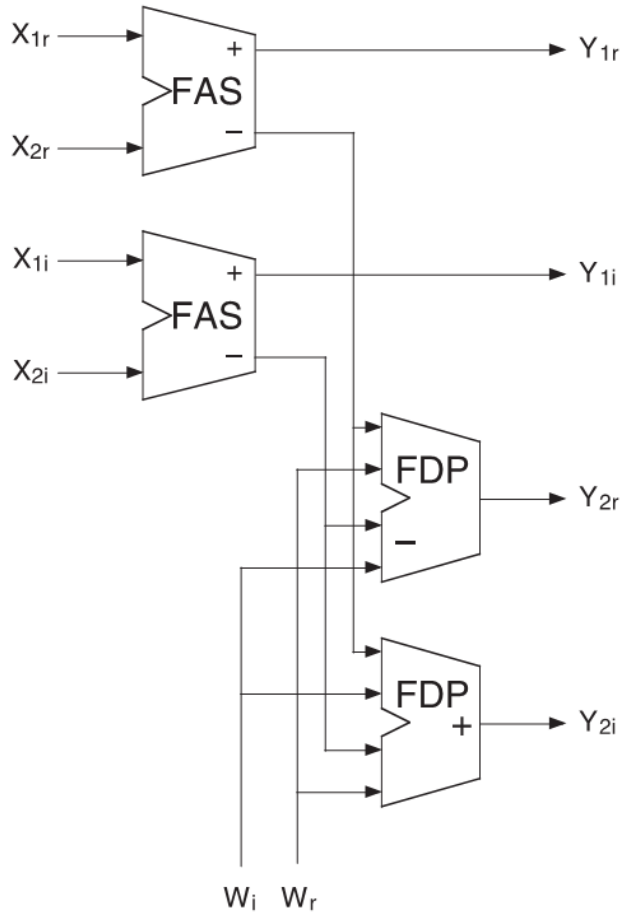


Figure 2.3: Radix-2 floating-point DIF butterfly [1]

In overall, the paper [1] shows that using fused operations to implement butterfly units (and FFT) leads to faster modules with lower area cost. This advantage is mainly achieved because of removing some extra rounding & normalization operations e.g., those of floating-point multipliers.

2.3 Floating-Point Fused Butterfly with Merged Multipliers [5]

Use of merged constant multipliers increases the performance of floating-point fused butterfly units. As mentioned before, given the constant values of twiddle factors, the conventional multipliers can be replaced by simple shift-add operations. In [5], it has been shown

that (for a 1024-point FFT) more than 28% percent of shift-coefficients of the shifters between real and imaginary parts of the twiddle factors are the same. For example, the shift coefficients of W_{re}^{798} are 1, 5, 8 and 11 while those of W_{im}^{798} are 0, 5, 9 and 11. Therefore, two coefficients, in this case (5 and 11), are the same [5]. Taking advantage of the same shift-coefficients reduces the power consumption of the butterfly units. Fig. 2.4 shows the butterfly unit of [5] using the merged multipliers.

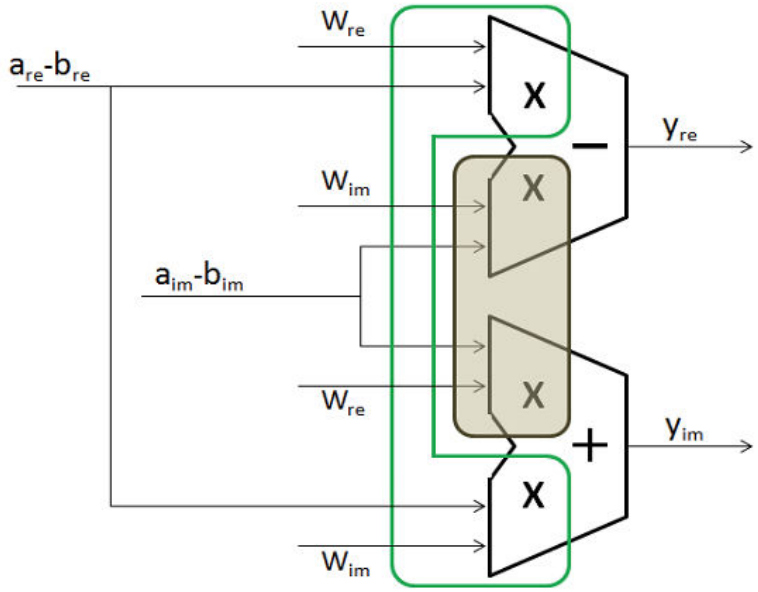


Figure 2.4: Butterfly unit using merged multipliers [5]

2.4 Improved Floating-Point Dot-Product Unit [2]

This work [2] is a modification to floating-point dot-product unit. The major improvements include:

- Reducing the shift amount in the alignment step
- Performing early normalization so as to reduce latency
- A four-input Leading-Zero-Detector is used over redundant operands to reduce critical path delay
- The dual path algorithm is also presented for speed improvement

Figs. 2.5 and 2.6 show the single path and dual path implementations of the enhanced dot-product of this work, respectively.

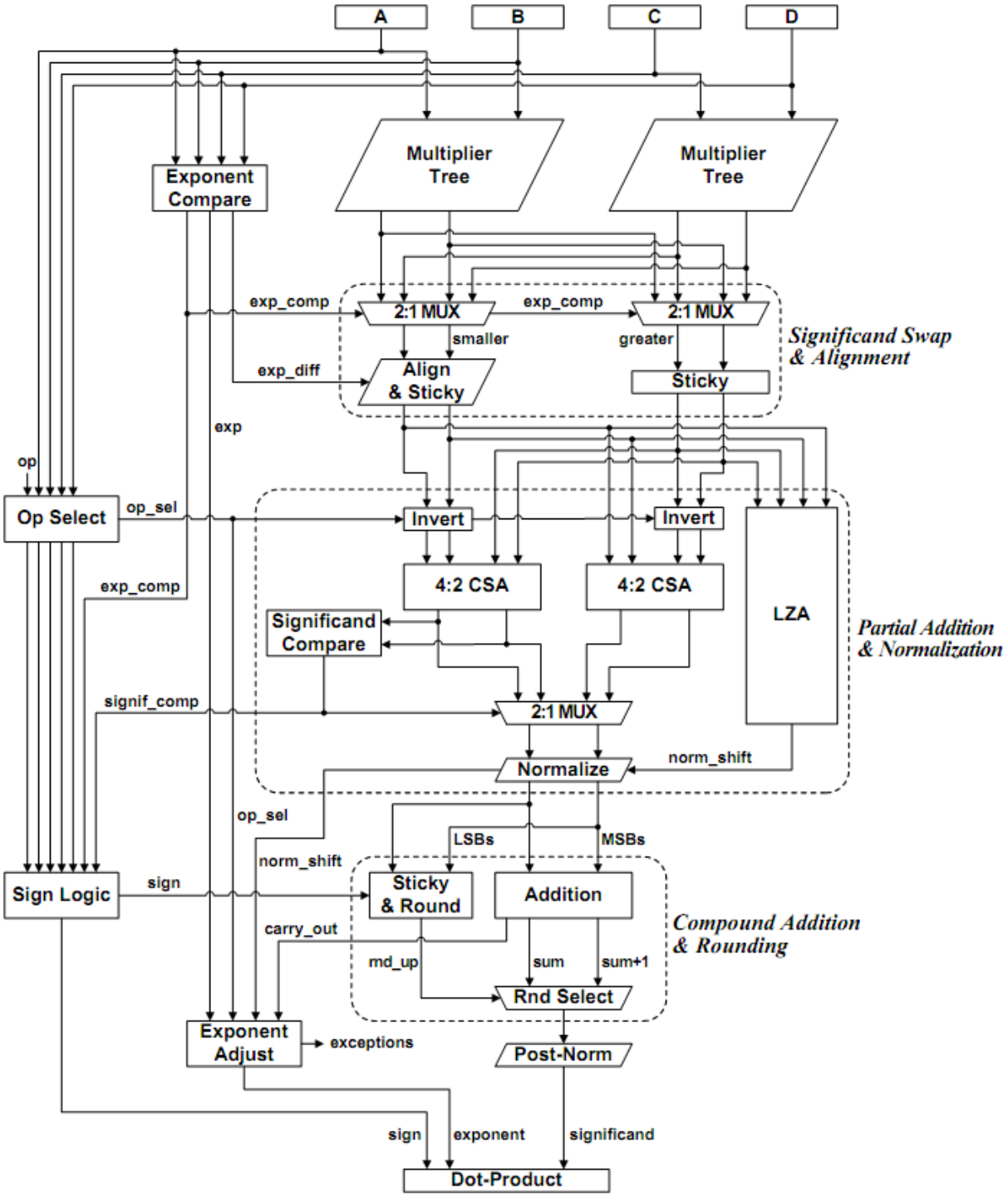


Figure 2.5: Enhanced floating-point dot-product unit (Single path) [2]

It has been shown that the enhanced single (dual) path dot-product unit consumes 25% (19%) lower area and 16% (26%) lower latency compared to the conventional method. This dot-product unit can be used in the design of a high-performance butterfly unit. Replacing the dot-product unit of [1] with this faster one, leads to a high-speed butterfly unit.

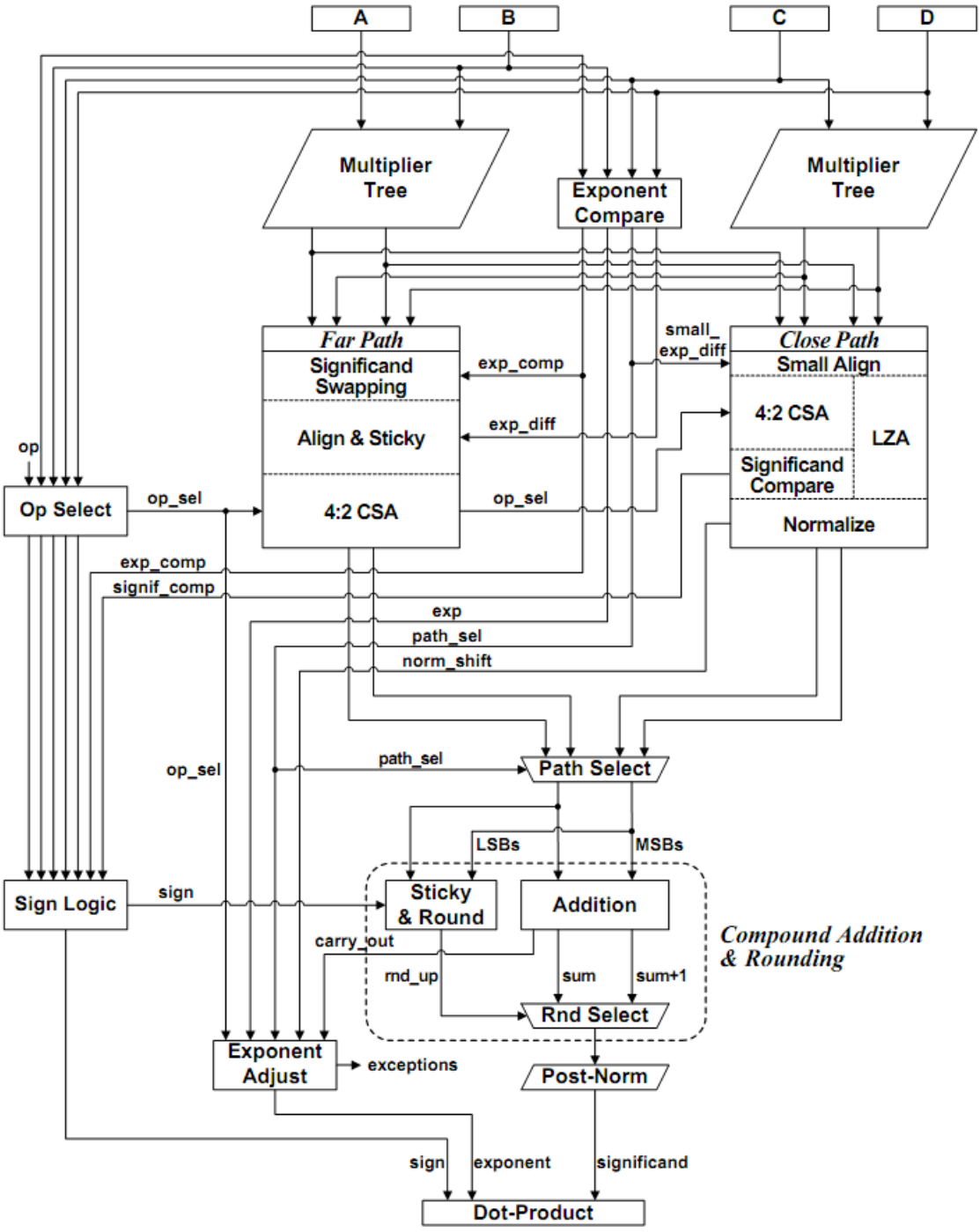


Figure 2.6: Enhanced floating-point dot-product unit (Dual path) [2]

2.5 Summary of Previous FFT Architectures

The first paper [19] is just about complex multipliers which is used in butterfly units. It concludes that the conventional architecture is more suitable for floating-point operands than Golub's method. The reason lies within the fact that the latency and cost of floating-point adders are close to those of floating-point multipliers. Consequently, Golub's method is not recommended for floating-point implementations.

The two major works on floating-point FFT/butterfly architecture are [5, 1]. The former introduces architectures based on multiple-constant multipliers (MCM) and merged multiple-constant multiplier (MMCM), amongst which the fastest design reported to have the latency of 4.08ns with the area consumption of $97{,}302\mu\text{m}^2$, in 45nm CMOS technology.

The other work [1] designs a floating-point butterfly unit using novel dot-product blocks. This work, simulated based on 45nm CMOS technology, has the latency of 4.00ns with area cost of about $47{,}489\mu\text{m}^2$. The dot-product unit of this design is reported to have a latency of about 2.72ns with $16{,}104\mu\text{m}^2$ area cost.

The most recent work [2] has proposed a very fast floating-point dot-product unit which can be used in the design of a high-performance butterfly unit. Replacing the dot-product unit of [1] with this faster one, leads to a high-speed butterfly architecture.

In a nutshell, a butterfly unit designed based on the combination of [1, 2] is the fastest architecture for both single-path and dual-path designs; while the architecture of [1] consumes the lowest area.

CHAPTER 3
THE PROPOSED FFT ARCHITECTURE[1]

The proposed FFT architecture is based on the Cooley-Tukey algorithm which is the most efficient FFT algorithm for hardware implementation. As is mentioned in the previous section, an *N*-point DIT FFT implementation using Cooley-Tukey algorithm has $\log_2 N$ stages, each of which consists of $N/2$ butterfly units. Moreover, $N/2$ twiddle factors are required, which can be pre-computed and stored a look-up table.

Therefore, the butterfly unit is the major building block of an FFT processor. Having the butterfly architecture, one can design the *N*-point FFT block using $\log_2 N$ stages, each of which consists of $N/2$ butterfly units working in parallel. Therefore, the latency of each stage is equal to that of a butterfly unit plus those of registers. Consequently, the details of the proposed butterfly architecture will be discussed in the following sub-sections.

3.1 The Proposed Butterfly Architecture

The proposed butterfly is actually a complex Fused-Multiply-Add followed by a complex addition with floating point operands. Expanding the complex numbers, Fig. 3.1 depicts the required modules for a butterfly unit. A naive approach to implement Fig. 3.1 is to cascade floating-point operations i.e., floating-point multiplication followed by two cascaded floating-point addition/subtraction. A more efficient approach is to merge the floating-point multiplication with the first floating-point addition/subtraction. This method leads to a floating-point Fused-Multiply-Add followed by a floating-point addition/subtraction.

[1] Published @ 1) IEEE Transactions on VLSI, 2) ISCAS'14

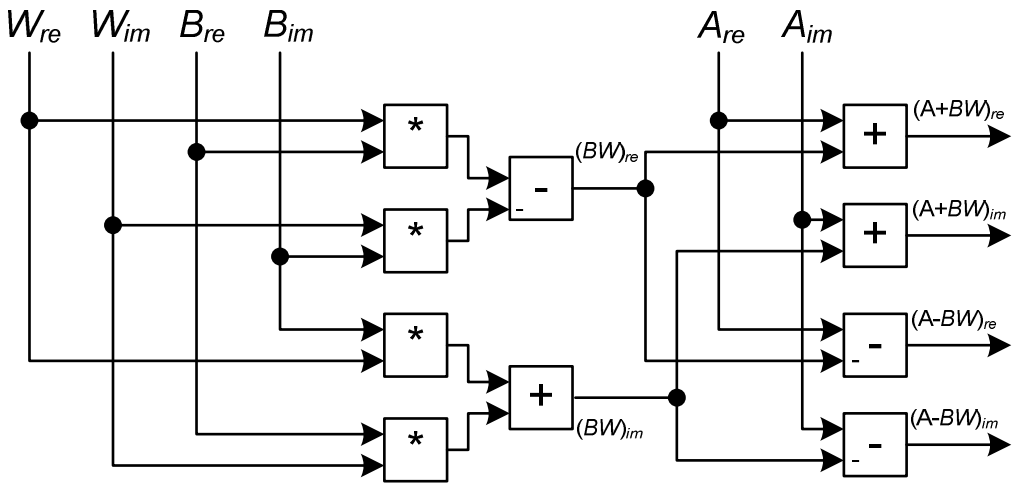


Figure 3.1: Butterfly Architecture with Expanded Complex Numbers

Using the floating-point Fused-Multiply-Add, as is discussed in Section 1.5, would save time, area and power. However, the butterfly function cannot be directly implemented by Fused-Multiply-Add (i.e., A+BC). In order to circumvent this problem a Dot-Product unit is required, which is an extension to Fused-Multiply-Add operation.

A Dot-Product unit computes AB+CD or AB-CD. This unit is capable of saving more time, area and power than Fused-Multiply-Add. The reason lies in the fact that a Dot-Product unit combines more floating-point operands and hence eliminating more intermediate Normalization, Rounding and Leading-Zero-Detection. Combining floating-point operations, although seems interesting and an easy way of saving time, area and power, leads to precision loss which need to be taken care of, meticulously.

For example, in a Fused-Multiply-Add unit the combination of the multiplication with the addition removes the intermediate Rounding, Normalization and Leading-Zero-Detection after the multiplication. However, it is important to pass wider operands (more number of bits) to the floating-point addition such that the required precision can be recovered at the end of the addition.

For a Dot-Product unit the loss of precision is more critical, given that the Rounding, Normalization and Leading-Zero-Detection of two multiplications are now removed and left to be dealt with at the end of the addition. Consequently, more bits have to be passed to the floating-point addition so as to be able to recover the required precision. Fig. 3.2 depicts a butterfly architecture implemented using Dot-Product units.

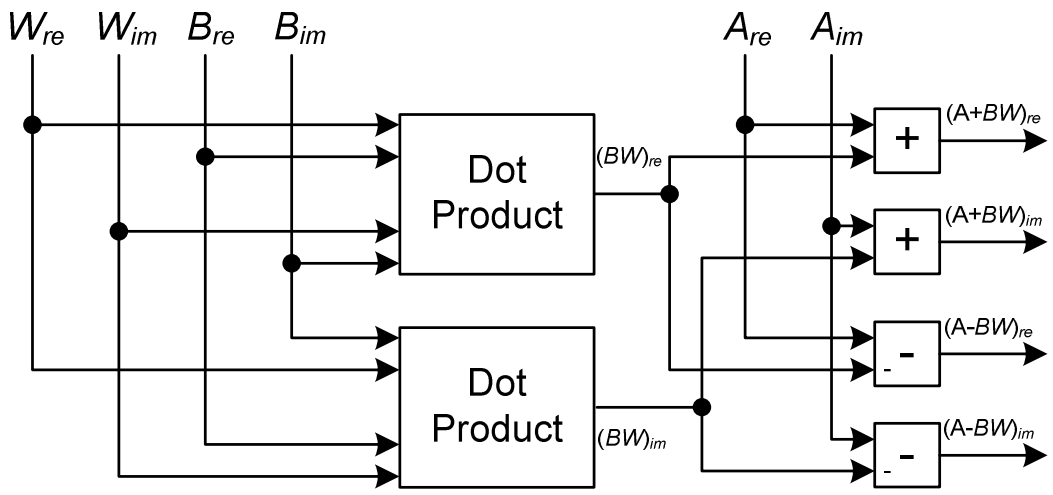


Figure 3.2: Butterfly Architecture with Dot-Product Units

According to Fig. 3.2, the constituent operations of a floating-point butterfly unit are a floating-point Dot-Product (e.g., $B_{re}W_{im} + B_{im}W_{re}$) followed by a floating-point addition/subtraction. Extending the concept of combining floating-point operations even further leads to the proposed Fused-Dot-Product-Add (FDPA) operation (e.g., $B_{re}W_{im} + B_{im}W_{re} + A_{im}$) over floating-point operands. Fig. 3.3 shows the butterfly unit implemented using the Fused-Dot-Product-Add unit.

It should be noted that in Fig. 3.3 the FDPA units provide two outputs (dubbed - and +). The - output computes $B_{re}W_{im} + B_{im}W_{re} - A_{im}$ while the + output is the result of $B_{re}W_{im} + B_{im}W_{re} + A_{im}$.

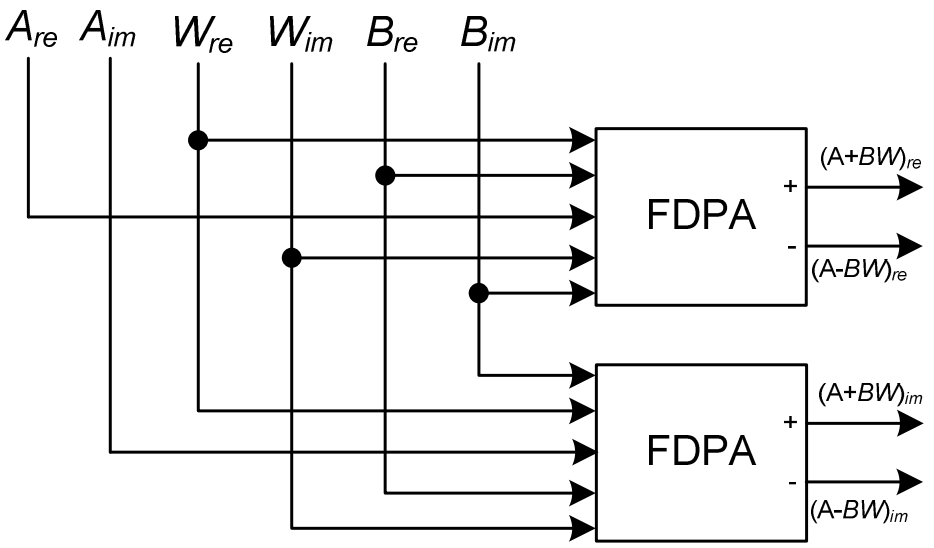


Figure 3.3: Butterfly Architecture with Floating-point Fused-Dot-Product-Add

It will be shown later that given the sign embedded representation of the floating-point operands of the FDPA, generating the second output is done with no extra latency and almost no extra area cost.

The Fused-Dot-Product-Add unit can be implemented in two ways:

a) Combining a floating-point Dot-Product unit with a floating-point addition.
b) Combining a floating-point multiplication with a floating-point three-operand addition.

Approach a) is a direct extension of Fig. 3.2 where the intermediate Rounding, Normalization and Leading-Zero-Detection (after Dot-Product) are eliminated and left to be dealt with after the addition. Fig. 3.4 illustrates the proposed floating-point FDPA unit implemented using Dot-Product units. This makes the required circuitry to guarantee the correct precision even more complicated, given that now three consecutive operations are combined (i.e., Multiplication and two additions). Moreover, even more number of bits are required to be passed between operations so as to reach the desired precision.

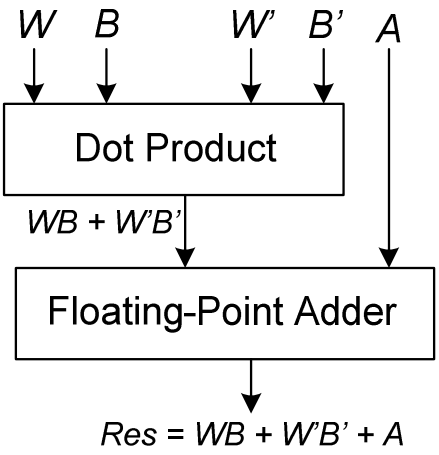


Figure 3.4: Floating-point Fused-Dot-Product-Add (FDPA) with Dot-Product Units

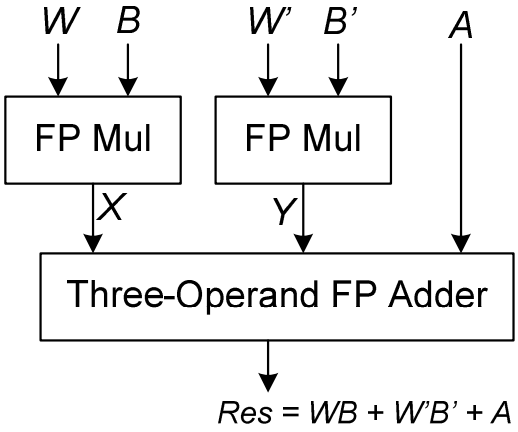


Figure 3.5: Floating-point Fused-Dot-Product-Add (FDPA) with 3-Operand Adder

Approach b) sees the floating-point fused-dot-product-add unit as a combination of a floating-point multiplier followed by a floating-point three operand addition. This eliminates the Rounding, Normalization and Leading-Zero-Detection after the multiplication as well as the one inside the floating-point three-operand adder. Fig. 3.5 illustrates the proposed floating-point fused-dot-product-add (FDPA) unit implemented using two floating-point multiplications followed by a three-operand floating-point addition.

3.1.1 Data Representation

The representation of the floating-point operands has a significant impact on the number of bits (weighted positions) required to be passed to the next operation. There is a need to find the most efficient representation for the exponent and significands of the floating-point operands. Moreover, a redundant representation, at least for the significands, is desired so as to eliminate intermediate carry-propagation.

Having a redundant representation for a floating-point operation creates some other challenges specially with the rounding and normalization. For instance, in a redundant floating-point adder designed by Fahmy and Flynn [20] a radix-16 redundant representation is used.

Assuming a maximally redundant radix-16 signed digit representation, with [-15, 15] as the digit set, each digit can be represented by a 5-bit 2's complement number. However, this leads to an invalid value (i.e., -16). Therefore, the first challenge would be designing the floating-point adder such that -16 cannot be generated in the output.

Determining the rounding position is another challenge, because the binary position for inserting the round value should be determined based on the non-redundant value of the significand (i.e., IEEE format). In other words, rounding the redundant representation and then converting it to non-redundant format must lead to the same value if it is converted first and then the result is rounded.

Leading-Zero-Detection is another challenge, which is somehow related to the rounding challenge. In the process of converting to non-redundant representation, a -1 value might be propagated to the most-significant position and turn it into zero. This would also change the correct rounding position. For higher radix-representations (e.g., radix-16) this may lead to a rounding position shifted as far as four digits.

In the proccess of choosing a redundant representation for the significand, it should be noted that higher radix representation leads to lower number of extra bit to store a significand; however, it leads to a small carry-propagation inside each digit. For example, radix-16 redundant representation requires a 4-bit carry-propagation inside a digit. Moreover, a higher radix representation requires a more complicated digit adder.

According to the above discussion, the followings present the data representation that will be used for the proposed butterfly architecture. The exponents of all inputs are represented in two's complement, after subtracting the bias. The value of the bias is determined by the IEEE format used.

The significands of A_{re}, A_{im}, B_{re} and B_{im} are represented in binary signed digit (BSD). Within the BSD representation (shown in Fig. 3.6 for A_{re}) every binary position takes values of $\{-1,0,1\}$ represented by one negative-weighted bit (*negabit*) [6] and one positive-weighted bit (*posibit*). Negabits (Posibits) are shown in white (black) dots and capital (small) letters.

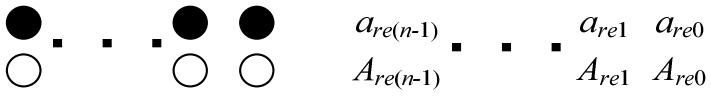


Figure 3.6: Dot and symbolic notation of the significand of A_{re}

The significand of W is stored in the modified Booth encoding [7] in which every binary position takes a value of $\{-1,0,1\}$ where there is at least one 0 in two adjacent positions. Therefore, only multiples of $\pm B$ and $\pm 2B$ are required which can be computed easily by shift and negation. This leads to a simpler partial product generation phase in the multiplier. The conversion to the modified Booth encoding is actually a radix-4 digit-set conversion i.e., from [0, 3] to [-2, 2].

For converting an n-bit binary number y $(y_{n-1}y_{n-2}\dots y_0)$ to a modified Booth representation z the first step is to put every 2 bits of y into a group such that

$$v_j = 2y_{2j+1} + y_{2j} \quad j = \frac{n}{2} - 1, \dots, 0 \tag{3.1}$$

The second step is to divide v_j into a radix-4 transfer t_{j+1} and an interim sum w_j; such that

$$v_j = 4t_{j+1} + w_j \tag{3.2}$$

Finally, the third step is to generate z_j by adding the same weighted transfers and interim sums such that

$$z_j = t_j + w_j \tag{3.3}$$

It should be noted that the third step has to be done in a carry-free manner i.e., the addition must not produce any further carry. This is guaranteed if the second step is performed such that

$$-2 \le w_j \le 1 \; and \; 0 \le t_j \le 1 \tag{3.4}$$

This is achieved if the following rules hold during the conversion:

$$w_j = \begin{cases} v_j & if \; v_j \le 1 \\ v_j - 4 & if \; v_j \ge 2 \end{cases} \qquad t_{j+1} = \begin{cases} 0 & if \; v_j \le 1 \\ 1 & if \; v_j \ge 2 \end{cases} \tag{3.5}$$

The details of the proposed redundant floating-point multiplier and the proposed redundant three-operand floating-point adder are presented below.

3.2 The Proposed Redundant Floating-Point Multiplier

Floating-point multiplication, as is discussed in Chapter 2, consists of operations on the exponents and those on the significands. The former is just a simple addition of the exponent; although, there may be a need for exponent adjustment in the normalization and rounding phase.

The latter, however, is the most time-consuming part of a floating-point multiplier. Multiplication over significands consists of three major steps called: 1) Partial Product Generation, 2) Partial Product Reduction and 3) Final Addition.

The proposed multiplier, likewise other parallel multipliers, consists of two major steps, namely, partial product generation (PPG) and partial product reduction (PPR). However, contrary to conventional multipliers, the proposed multiplier keeps the product in redundant format and hence there is no need for the final carry-propagating adder.

3.2.1 Partial Product Generation (PPG)

The partial product generation, in a 2's complement representation of the multiplicand and multiplier, consists of arrays of AND operation such that each bit of the multiplier is ANDed to the whole bits of the multiplicand. This is not the case if the operands are represented in redundant format and/or Booth encoding. For example, if the multiplier is represented in the modified Booth encoding, the partial product generation looks like the circuitry shown in Fig. 3.7 [17].

Partial product generation of a redundant multiplier is even more complicated, since the cardinality of the multiplier's digit-set is more than the radix. Generating the multiples of the multiplicand is easy (shift and negation) for ±2x and ±1x; however, ±3x and ±5x (if exists) involve an addition. Fig. 3.8 shows how these multiples are generated. Consequently, higher redundancy factor (see Chapter 2) leads to more complicated partial product generation; although it provides faster redundant addition.

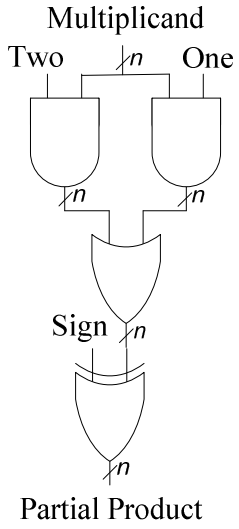


Figure 3.7: Partial Product Generation of the modified Booth encoding

$$
\begin{array}{rl}
X: & x_{n-1} \ldots x_1\ x_0 \\
2X: & x_{n-1} \ldots x_1\ x_0\ 0 \\
4X: & x_{n-1} \ldots x_1\ x_0\ 0\ 0 \\
X: & x_{n-1} \ldots x_1\ x_0 \\
+\ 2X: & x_{n-1} \ldots x_1\ x_0\ 0 \\
\hline
3X: & y_n\ y_{n-1} \ldots y_1\ y_0 \\
X: & x_{n-1} \ldots x_1\ x_0 \\
+\ 4X: & x_{n-1} \ldots x_1\ x_0\ 0\ 0 \\
\hline
5X: & z_{n-1}\ z_n\ z_{n-1} \ldots z_1\ z_0
\end{array}
$$

Figure 3.8: Generating the Multiples of the Multiplicand

The PPG step of the proposed multiplier is completely different from that of the conventional one because of the representation of the input operands (B, W, B', W'). Moreover, given that W_{re} and W_{im} are constants, the multiplications over significands can be computed via a series of shifters and adders.

For example, multiplying B1 by 113 $(1110001)_2$ can be done by the following shift and add steps [7]:

Step *a*: B2 = Shift B1 left for 1 bit
Step *b*: B3 = R2 + B1
Step *c*: B6 = Shift B3 left for 1 bit
Step *d*: B7 = B6 + B1
Step *e*: B112 = Shift B7 left for 4 bits
Step *f*: B113 = B112 + B1

In order to speed-up the above operation one may use the following sets of operations; however, it requires a hardware to be able to perform shift and add, simultaneously.

Step *a*: B3 = (Shift B1 left for 1 bit) + B1
Step *b*: B7 = (Shift B3 left for 1 bit) + B1
Step *c*: B113 = (Shift B7 left for 4 bits) + B1

It should be noted that in order to perform the above steps, there is also a need for a barrel shifter to be able to shift an operand for various number of bits. For example, Steps *a* & *b* require 1 bit shifts however, Step *c* requires a 4-bit shift. The other sequence would be the one shown below, where there is a need for both addition and subtraction.

Step *a*: B8 = Shift B1 left for 3 bits
Step *b*: B7 = B8 - B1
Step *c*: B112 = Shift B7 left for 4 bits
Step *d*: B113 = B112 + B1

Having the ability of performing subtraction is mostly useful when the multiplier has a series of consecutive 1s in its binary representation. In this case one may need to perform lots of addition operations however, with a subtraction operation only one subtraction and one addition are required.

With the intention of reducing the number of adders, the significand of W is stored in modified Booth encoding [7] in which every binary position takes a value of $\{-1,0,1\}$ where there is at least one 0 in two adjacent positions. Therefore, $\left\lceil \frac{n}{2} \right\rceil$ add/sub is sufficient to compute an n-by-n multiplication.

Given the modified Booth representation of W_{re} and W_{im} one partial product (PP), selected from multiplicand B, is generated per two binary positions of the multiplier W, as shown in Table 3.1. Note that each binary position (e.g., W_i) consists of two bits $W_i^- W_i^+$ to represent $-1, 0$ or 1.

Table 3.1: Generation of i^{th} partial product

$W_{i+1}^- W_{i+1}^+$	$W_i^- W_i^+$	$\|W_{i+1}^- W_{i+1}^+ W_i^- W_i^+\|$	PP_i
0 0	0 0	0	0
0 0	0 1	1	B
0 0	1 1	-1	$-B$
0 1	0 0	2	$2 \times B$
1 1	0 0	-2	$-2 \times B$

Fig. 3.9 illustrates the required circuitry for the generation of PP_i based on Table 3.1. It should be noted that given the binary-signed-digit (BSD) representation of multiplicand B, the value of $-B$ ($-2B$) is generated through a simple NOT over all bits of B ($2B$).

Moreover, $2B$ is generated via a 1-bit left shift over B. Note that each partial product consists of (n+1) digits (i.e., binary positions), each of which has a negabit and a posibit.

3.2.2 Partial Product Reduction (PPR)

Partial product reduction phase is actually a multi-operand addition of the partial products generated in the partial product generation phase.

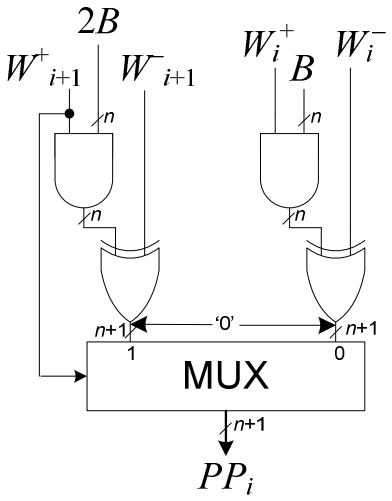


Figure 3.9: Generation of the i^{th} partial product

In general, there are two approaches to reduce the partial products; 1) reduction by rows and 2) reduction by columns. The former uses adders which could be either redundant adders or carry-propagating adders. Let's say 8 partial products are reduced using the adders (i.e., reduction by rows). Fig. 3.10 depicts the required steps. It takes 3 steps to reduce the 8 operands while each step has the latency of one adder. The total number of adders used in this reduction method is seven.

Generalizing this approach for p operands (i.e., [p:1] reduction block) is described next. Reduction of p operands can be divided into two parts, each of which reduces p/2 operands (i.e., [p/2:1] reduction block); and then add the outputs together.

Fig. 3.11 shows the reduction of p operands using reduction blocks for p/2 operands. Each of those [p/2:1] modules could be further divided into two sub modules. Continuing this approach leads to a [2:1] reduction block which is known as a carry-propagating adder. Consequently, log(p) steps are required to reduce p operands to 1.

Reduction by columns is done by modules called counters or compressors. These modules take p bits, all at the same weighted position, and generate q bits of adjacent weights. In

other words, the number of 1s in p bits is counted and represented in a q-bit number. That is why these modules are called [p:q] counters. Therefore, the following relation between q and p holds [17].

$$2^q - 1 \geq p \rightarrow q = \lceil \log_2(p+1) \rceil \tag{3.6}$$

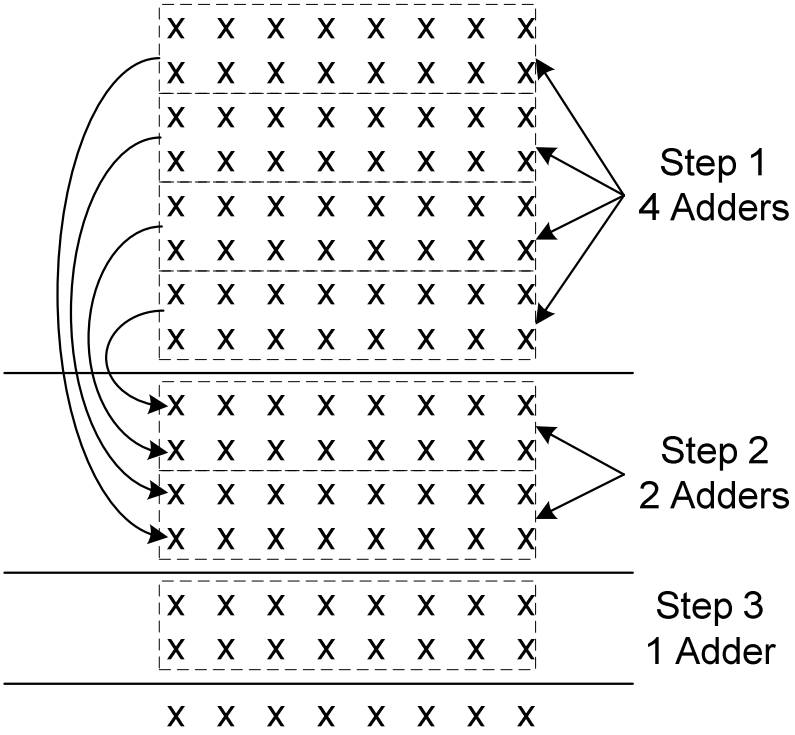


Figure 3.10: Partial Product Reduction by Rows

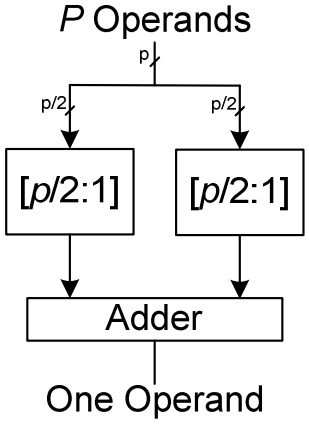


Figure 3.11: [*p*:1] reduction block based on [*p*/2:1] blocks

A [3:2] counter is simply a full-adder which is implemented using the following logical expressions:

$$q_1 = p_2p_1 + p_2p_0 + p_1p_0,\ q_0 = p_2 \oplus p_1 \oplus p_0 \tag{3.7}$$

Larger counters can be built using full-adders and half-adders (i.e., [2:2] counter). For example, Fig. 3.12 depicts a [7:3] counter implemented by full-adders. The numbers next to each wire show the weight of that input/output. It should be noted that in a counter all same weighted outputs can be added together using a full-adder. The final output is a 3-bit number which counts the number of 1s in the input p.

Having multiple counters working in parallel leads to a multi-column counter which can be used to reduce several columns. In a multi-column counter, there are multiple counters each of which performing on a single weighted position and passes the carries to the next higher weighted column. For example, Fig. 3.13 shows a [7:2] compressor which passes the carries to the next higher weighted position and receives input carries from the lower weighted position. It should be noted that this module is called [7:2] compressor, because 2 bits are not enough to count 7 bits.

Therefore, partial product reduction phase of a multiplier can be taken care of by multi-column compressors. These modules take p bits of a single column (same weights) and reduce it into two bits per column. For example, a 54-by-54 bit multiplier generates 54 partial products. In order to reduce these partial products one could design a [54:2] compressor to reduce them to only two operands. Then a carry-propagating adder, over the two operands, generates the final product. Large compressors/counters can be built based on smaller ones. For instance, Fig. 3.15 shows a [16:2] compressor based on [4:2] compressors (Fig. 3.14). The PPR step of the proposed

multiplier is based on the reduction by row approach, however, it is completely different from that of the conventional method.

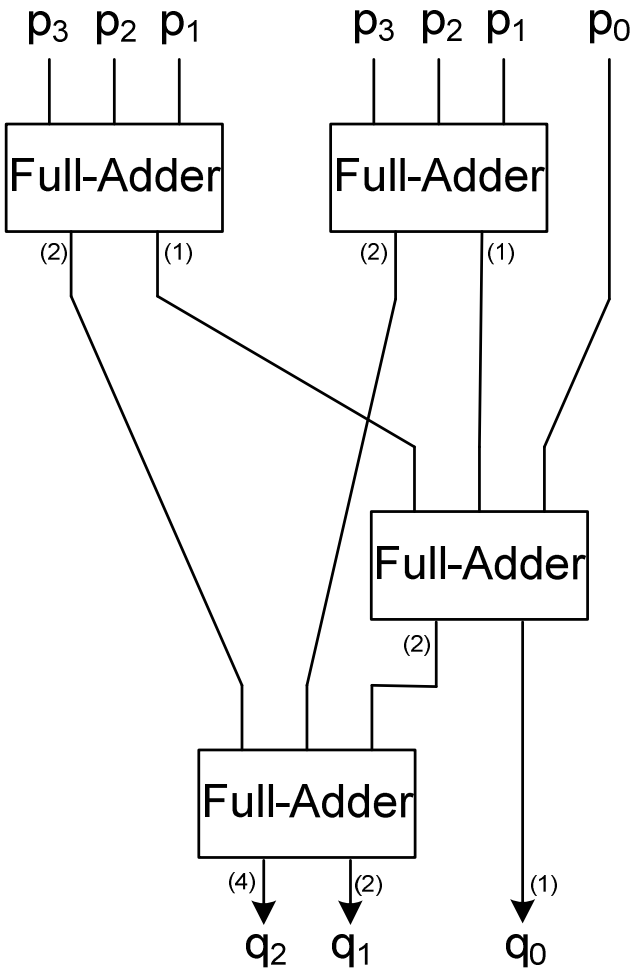


Figure 3.12: [7:3] Counter by Full-Adders

Given that partial products are all represented in a redundant encoding i.e., Binary-Signed-Digit, there is a need for an adder/counter that works on BSD digits. This carry-limited addition circuitry is shown in Fig. 3.16, where capital (small) letters symbolize negabits (posibits). The white dots are logical NOT operators required over negabit signals [6]. The critical path delay of this adder consists of three full-adders.

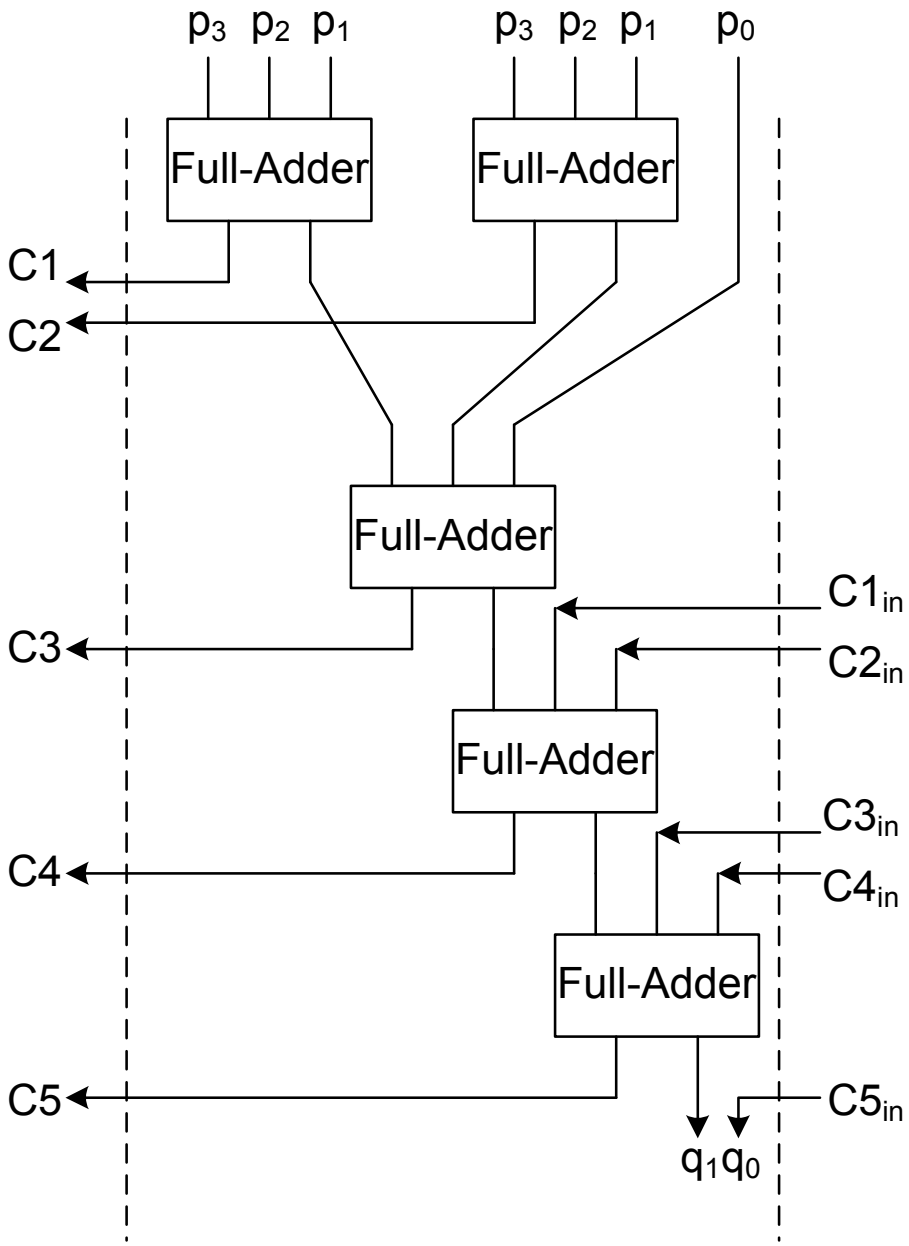


Figure 3.13: [7:2] Compressor by Full-Adders

Since the BSD adder is actually a carry-limited adder, reducing the partial product using this adder can be deemed as a reduction-by-column or reduction-by-row approach. In either case, the major constituent of the PPR step is the proposed carry-limited addition over the operands represented in BSD format.

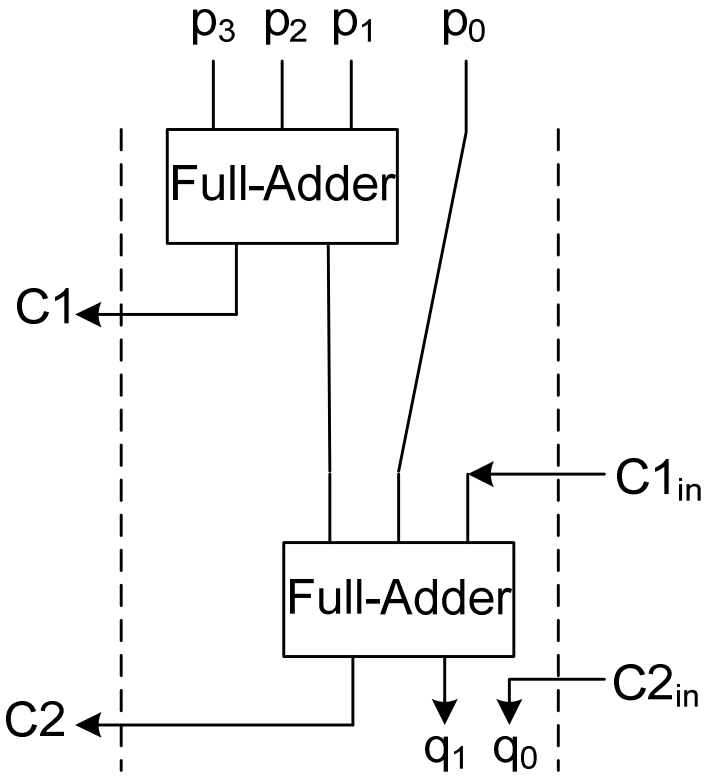


Figure 3.14: [4:2] Compressor by Full-Adders

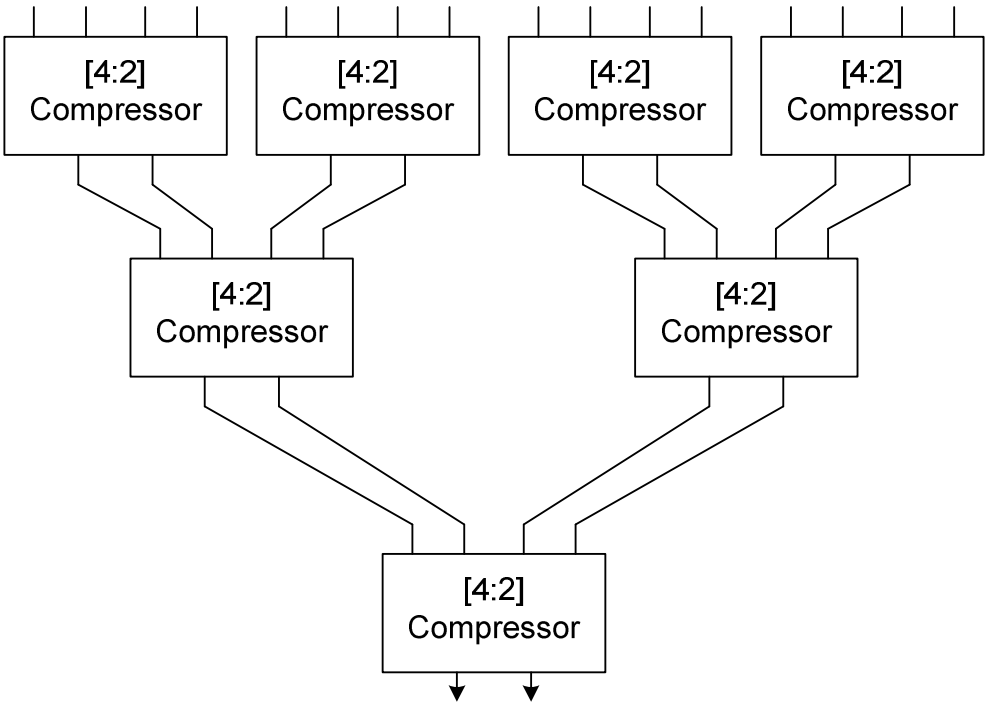


Figure 3.15: [16:2] Compressor by [4:2] Compressors

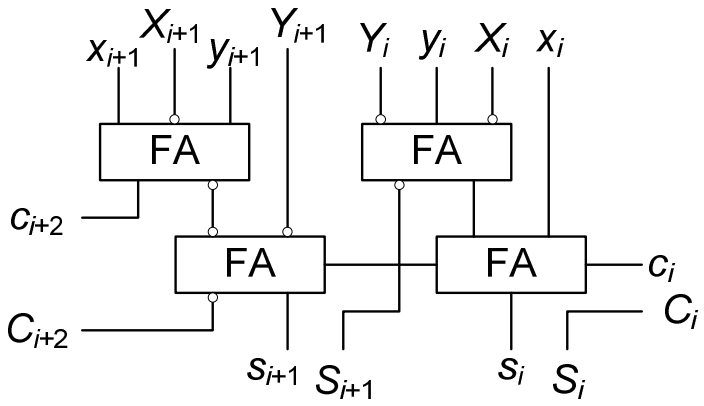


Figure 3.16: The proposed BSD adder (two digit slice)

Once the partial products are generated, in PPG step, carry-limited adders (Fig. 3.16) are used to generate the final product. Therefore, for an *n*-by-*n* multiplier, $\left\lceil \frac{n}{2} \right\rceil + 1$ (1 for the hidden bit of *W* encoded in modified Booth) partial products are generated and $\lceil \log n \rceil - 1$ levels of BSD adders are required to produce the product.

Since each partial product (PP_i) is (*n*+1)-digit (*n*,…,0) which is either *B* (*n*-1,…,0) or 2*B* (*n*,…,1), the length of the final product may be more than 2*n*. For example, a 24-digit multiplier (compliant with IEEE single precision format) leads to a 51-digit product. Assuming that the sign-embedded significands of inputs *A* and *B* (24 bits) are represented in Binary-Signed-Digit; while that of *W* is represented in modified Booth encoding (25 bits). The last partial product has 24-(binary position) width (instead of 25), given that the most significant bit of *W* is always 1 (hidden bit).

The multiplication over significands is implemented using the partial product generation (PPG) unit of Fig. 3.9 and BSD adders of Fig. 3.16. The reduction of the partial products is done in four levels, as shown in Fig. 3.17, using twelve BSD adders.

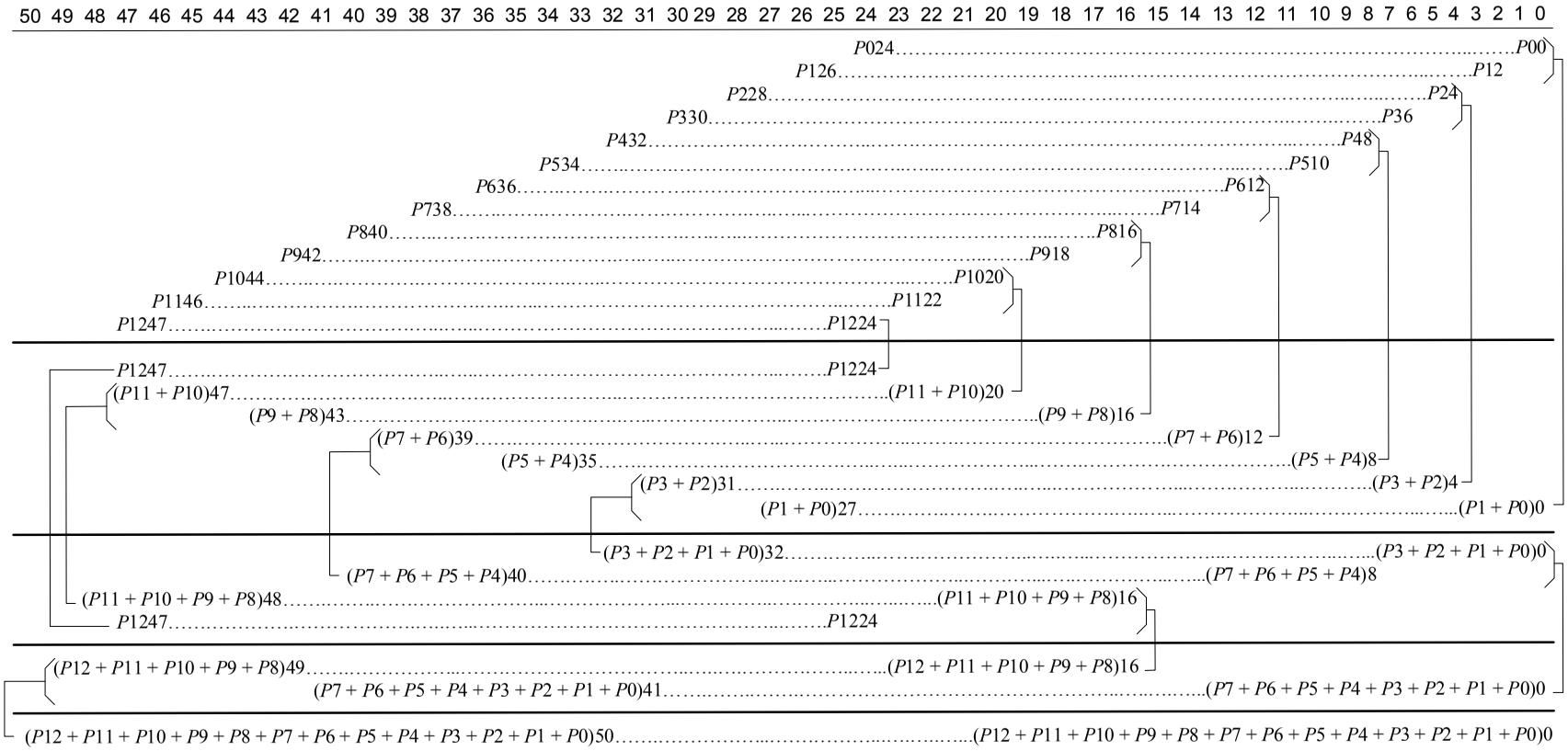


Figure 3.17: Partial Product Reduction Tree

The numbers in the first row of Fig. 3.17 show the bit positions of each partial product. For example, P024 symbolizes bit position 24 of first partial product (PP0). Note that the last partial product has 24-(binary position) width (instead of 25), given that the 26th bit of *W* is always 1 (hidden bit).

Now that the product is generated, it is required to determine how many bits are required to be passed to the three-operand adder so as to meet the precision requirements. As is shown in Fig. 3.17, the final product is a 51-bit number represented in Binary-Signed-Digit encoding. Given the normalized single precision formats of the inputs (*B* is in ±[1, 2) and *W* in [1, 2)), the final product is in ±[1, 4).

If the multiplier's operands were represented in standard IEEE format (i.e., each with 24 bits), the final product would fit into 48 binary positions (47…0). Consequently, positions 45 down to 0 would be fractions (see Fig. 3.18). However, the product out of the proposed BSD multiplier has 51 binary positions. Given that the value of this number is the same as that of the standard product, the 45 least significant positions are fractions.

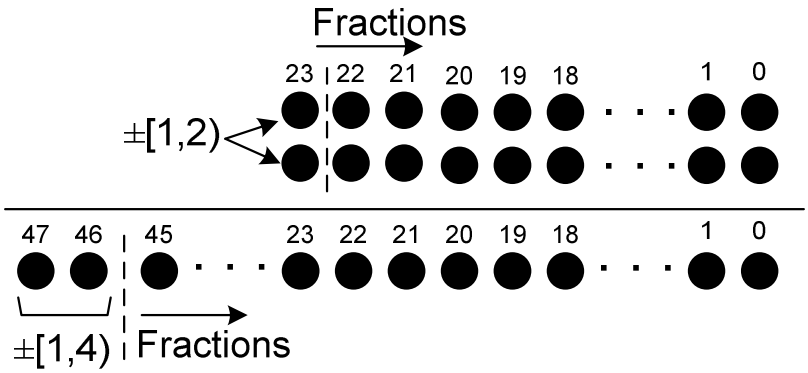


Figure 3.18: Final Product Format with Standard Non-redundant Operands

Similar to standard binary representation, Guard (G) and Round (R) positions are sufficient for correct rounding. Therefore, only 23+2 fractional binary positions of the final product are required to guarantee the final error less than 2^{-23} (required by IEEE Standard).

Selecting 25 binary positions out of 46 fractional positions of the final product dismisses positions 0 to 20. However, the addition of the next step would produce carries to Guard and Round positions. Nevertheless, because of the carry-limited BSD addition, contrary to standard binary addition, only positions 20 and 19 may produce such carries.

It should be noted that each position of a BSD number has the value of {-1, 0, 1}, while converting this representation to the standard non-redundant format a -1 carry could be propagated to the most-significant digit (MSD) and may turn the MSD into 0. In this case, the rounding position (determined based on the number of detected zeros) might be one position to right of the least-significant digit. Therefore, again positions 20 and 19 are enough to be passed to the three-operand adder.

In overall, positions 0 to 18 of the final product are not used and hence a simpler PPR tree is possible. Fig. 3.19 shows the required digits passed to the three-operand adder. Fig. 3.20 illustrates the proposed redundant floating-point multiplier.

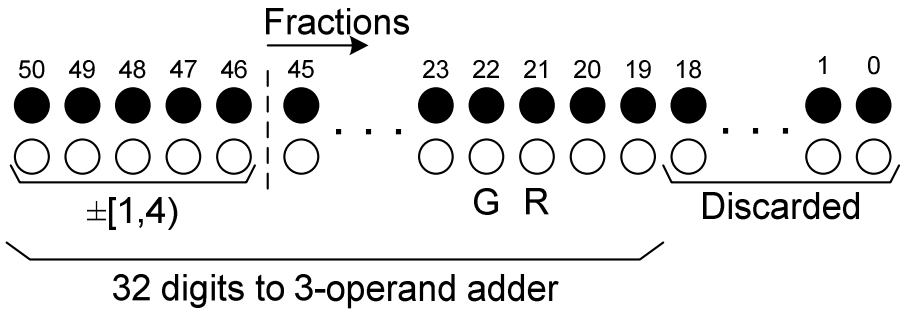


Figure 3.19: Redundant Product of the Proposed Multiplier

The exponents of the input operands are taken care of in the same way as is done in conventional floating-point multipliers; i.e., adding the exponents to get the product's exponent. However, normalization and rounding are left to be done in the next block of the butterfly architecture (i.e., three-operand adder).

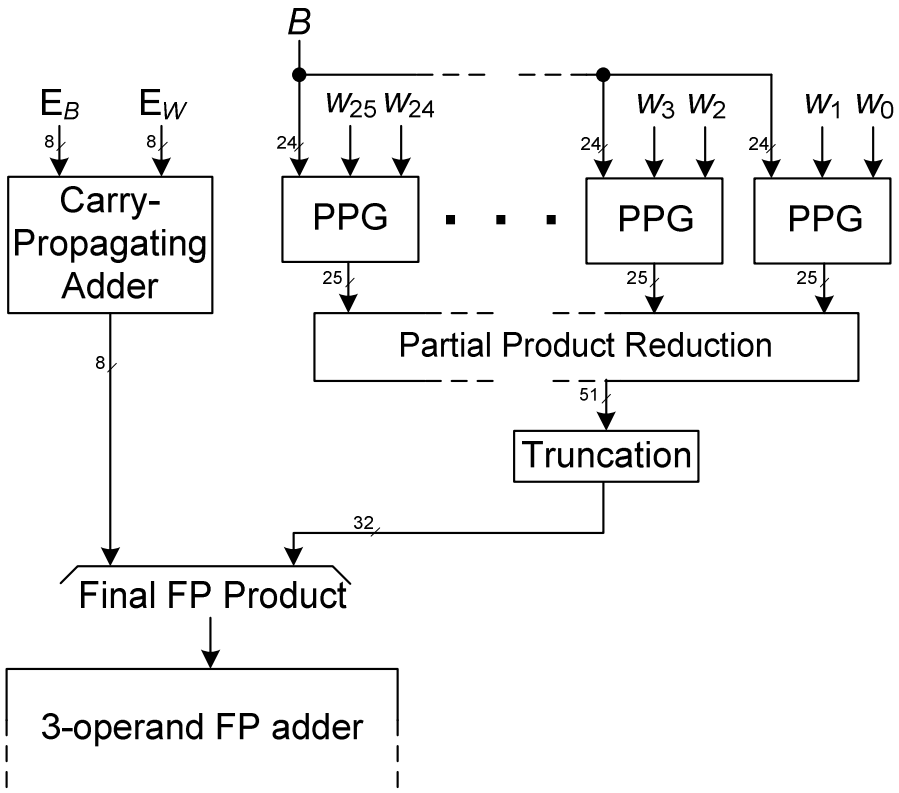


Figure 3.20: The Proposed Redundant Floating-Point Multiplier

3.3 The Proposed Three-Operand Redundant Floating-Point Adder

The proposed three-operand floating-point adder (computing $Res = X + Y + A$) accepts three operands as inputs:

- X and Y: The products of the redundant floating-point multipliers, each of which with a 32-digit significand
- A: The floating-point input with a 24-digit significand

The straightforward approach to perform a three-operand floating-point addition is to cascade two floating-point adders, as shown in Fig. 3.21.

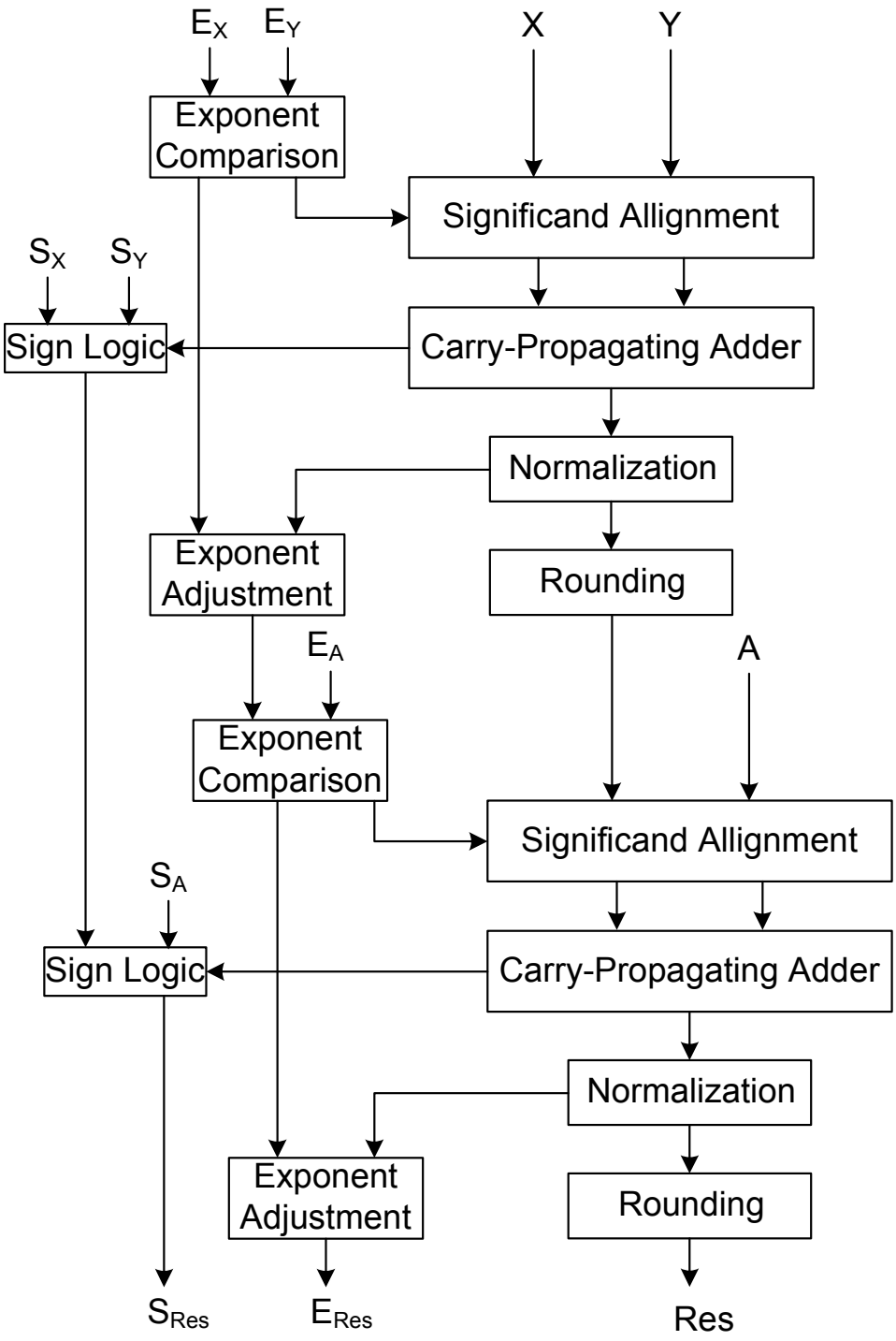


Figure 3.21: Straightforward Three-Operand Floating-Point Adder

The followings discuss the details of each building block in the straightforward three-operand floating-point adder:

- Exponent Comparison: It is a simple 8-bit subtractor. The output is the difference Δ (could be negative or positive) between the two exponents, passed to the significand alignment block and the exponent adjustment block.
- Significand Alignment: This block consists of a multiplexer and a barrel shifter (shown in Fig. 3.22). The multiplexer selects the significand with the smaller exponent and passes it to the barrel shifter. The barrel shifter shifts the selected significand for Δ bits to the right. The other significand goes to the output intact.

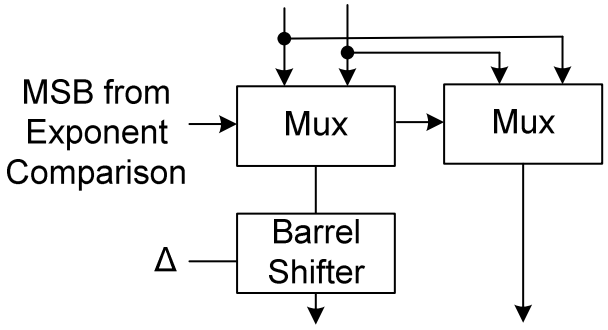


Figure 3.22: Significand Alignment Block

- Carry-Propagating Adder: Any kind of carry-propagating adder (e.g., carry-look-ahead, ripple-carry, carry-skip etc.) could be used here.
- Sign Logic: This block determines the sign of the addition which consists of simple XOR gates.
- Normalization: This block, as shown in Fig. 3.23, consists of two phases: 1) leading-zero-detection and 2) barrel shifter. The leading-zero-detector detect the most significant 1 and counts the number of leading 0s. This value is passed to the barrel shifter to shift the significand to the left. The amount of shift value is also passed to Exponent Adjustment.

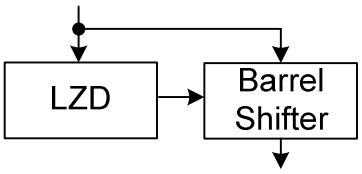


Figure 3.23: Normalization Block

- Exponent Adjustment: This block subtracts the amount left-shift bits (received from Normalization block) from the exponent.
- Rounding: This block is responsible to round the significand according to the standard rounding modes [3]. Four rounding modes are described below assuming that the value *x* is between two floating-point values F1 and F2 (as shown in Fig. 3.24).

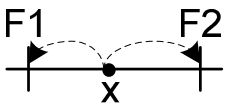


Figure 3.24: Value *x* between two floating-point values F1 and F2

1. Round to nearest (tie to even):

$$Round(x) = \begin{cases} F1 & |x - F1| < |x - F2| \\ F2 & |x - F1| > |x - F2| \\ even(F1, F2) & |x - F1| = |x - F2| \end{cases}$$

2. Round toward zero:

$$Round(x) = \begin{cases} F1 & x \geq 0 \\ F2 & x < 0 \end{cases}$$

3. Round toward $+\infty$:

$$Round(x) = F2$$

4. Round toward $-\infty$:

$$Round(x) = F1$$

The straightforward three-operand floating-point adder suffers from high latency, power and area consumption. A better way to implement this module is to use fused three-operand floating-point adders [21, 22], shown in Fig. 3.25.

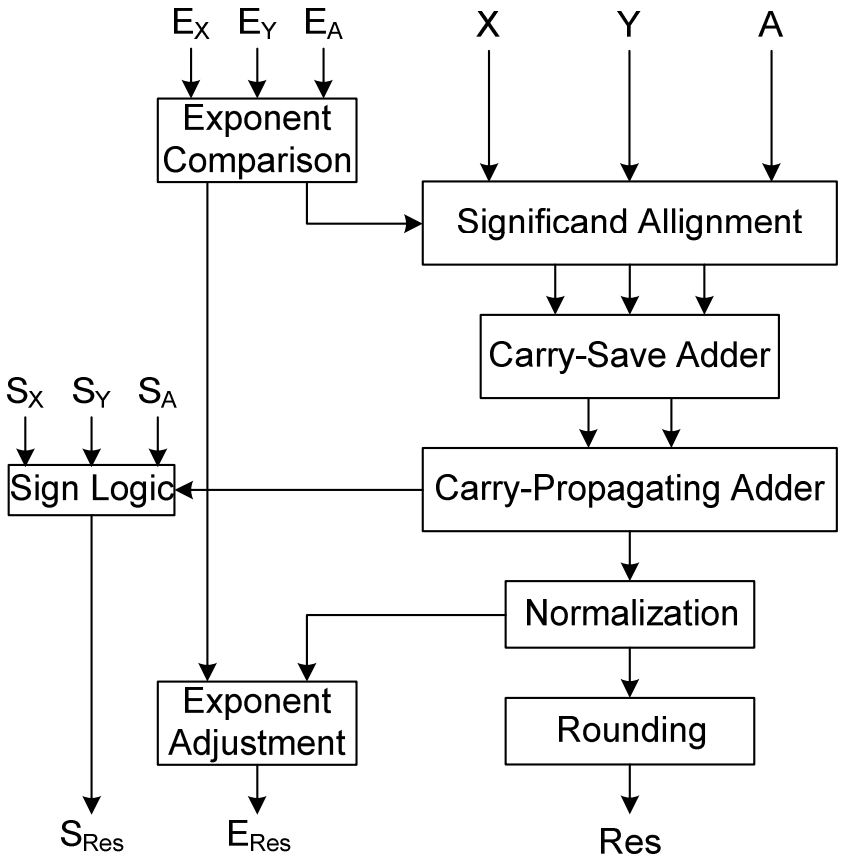


Figure 3.25: Conventional Fused Three-Operand Floating-Point Adder

The constituent blocks of this architecture are almost the same as those of the straightforward architecture, except for the extra carry-save adder. However, the functionality of some of these blocks are totally different from those of the straightforward approach.

The differences are:

- *Exponent Comparison*: This block is meant to determine the biggest exponent among the three inputs. Fig. 3.26 shows a straightforward implementation of this block. The biggest exponent is determined and the difference between the largest

exponent and other exponents (Δ_1 and Δ_2) is computed and sent to the significand alignment block.

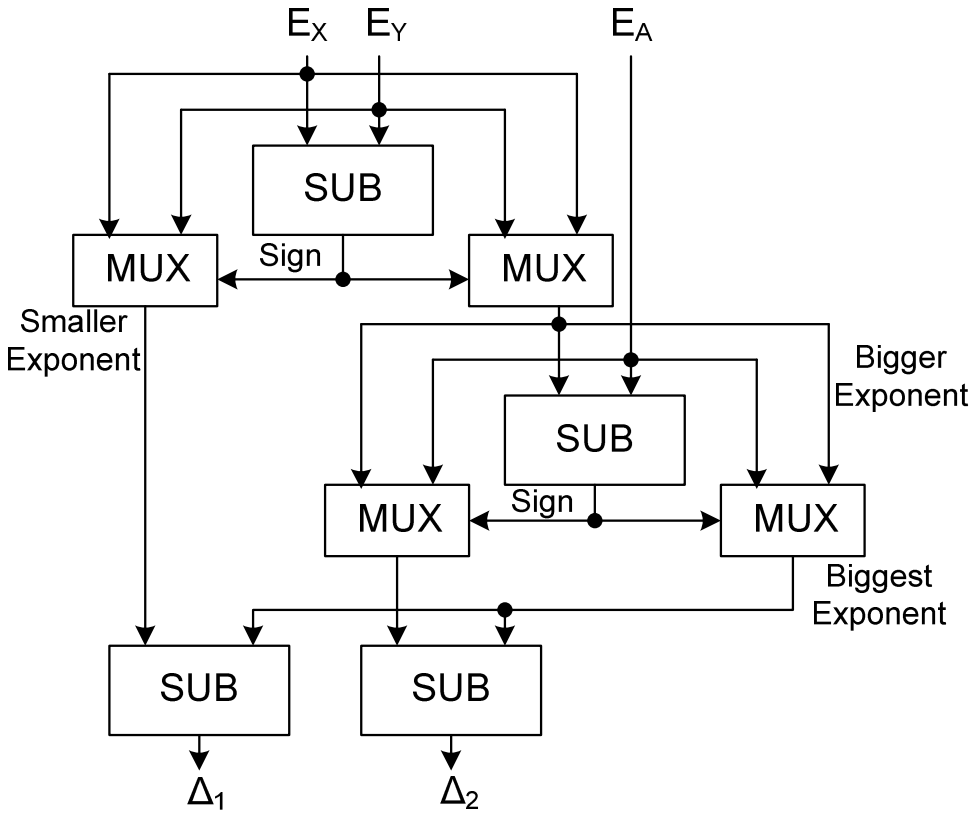


Figure 3.26: Exponent Comparison with Three Inputs

- *Significand Alignment*: This block receives two Δ signals and two select signals from the exponent comparison block and shifts the selected significands accordingly. Fig. 3.27 depicts the details of the significand alignment block.

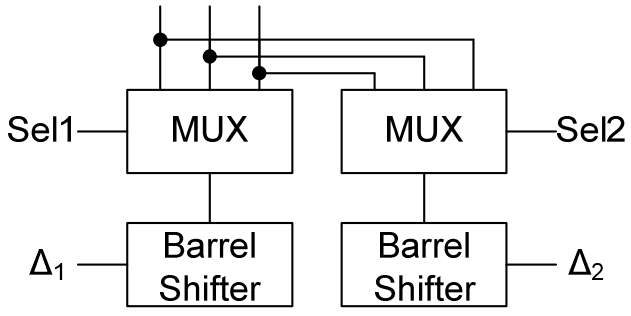


Figure 3.27: Significand Alignment of the Fused Three-Operand Floating-Point Adder

- *Carry-Save Adder*: It consists of multiple full-adders working on parallel. So the overall latency of this block is equal to that of a single full-adder.

The rest of the blocks in the conventional fused three-operand adder is the same as that of the straightforward approach. The normalization part, which consists of the leading-zero detector (LZD), produces the normalized significand; and the rounding block generates the final sum. The exponent of the final sum is the largest exponent, adjusted according to the amount of the normalization shift.

In the proposed three-operand floating-point adder, a new alignment block is implemented and CSA-CPA are replaced by the proposed Binary Signed-Digit adders. Moreover, sign logic is eliminated. The details of the proposed three-operand floating-point adder are presented below.

The exponent comparison and significand alignment of the proposed architecture is almost the same as that of the fused three-operand adder. The only difference is that the significand alignment block, in the proposed design, does not wait for the exponent comparison block to finish and part of significand alignment operation is overlapped with the exponent comparison.

Moreover, the addition part also overlaps with the significand alignment and exponent comparison. The only blocks that wait for other blocks to finish before they start their operations are Normalization, Rounding and exponent adjustment. Therefore, having multiple blocks working partially in parallel makes the proposed three-operand floating-point adder faster than previous works. In essence, the exponent comparison, first, selects the bigger exponent between E_X and E_Y (called E_{Big}) using a binary subtractor ($\Delta = E_X - E_Y$); and the operand with the

smaller exponent (X or Y) is shifted $\|\Delta\|$-bit to the right. Next, a Binary Signed-Digit adder computes the addition result ($SUM = X + Y$), using the aligned X and Y.

Adding third operand (i.e., $SUM + A$) requires another alignment. This second alignment is done in a different way of the previous one so as to reduce the critical path delay of the three-operand adder. First, the value of $\Delta_A = E_{Big} - E_A + 30$ is computed which shows the amount of right shifts required to be performed on A (with the initial position of 30 digits shifted to the left). This initial 30-digit wired shift (i.e., 30-digit extension) eliminates the need for any left shift of the third operand significand (A). This reduces the complexity of the second barrel shifter, thereby saving latency and area consumption.

Fig. 3.28 illustrates the alignments implemented in the proposed three-operand floating-point adder. In case that A is not shifted (i.e., the result after alignment is A), the two zero digits (shown in shaded in Fig. 3.28) are used as Guard and Round digits.

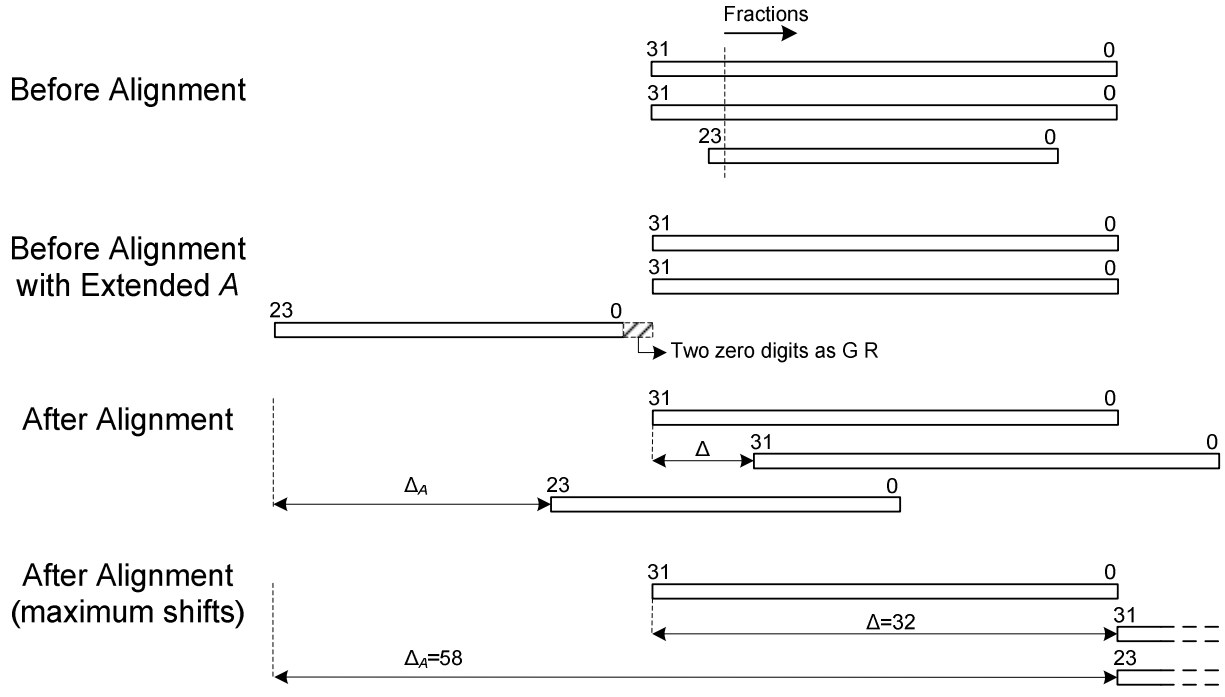


Figure 3.28: The Proposed Three-Operand Alignment Scheme

Next, a BSD adder adds the aligned third significand (58-digit) to *SUM* (33-digit) generated from the first BSD adder. Since the input operands have different number of digits, this adder is a 58-digit BSD adder in which some positions consist of the digit adder with $y_i, Y_i, y_{i+1}, Y_{i+1}$ assigned to '0'. Fig. 3.29 depicts the Binary Signed-Digit adder with some inputs assigned to '0'.

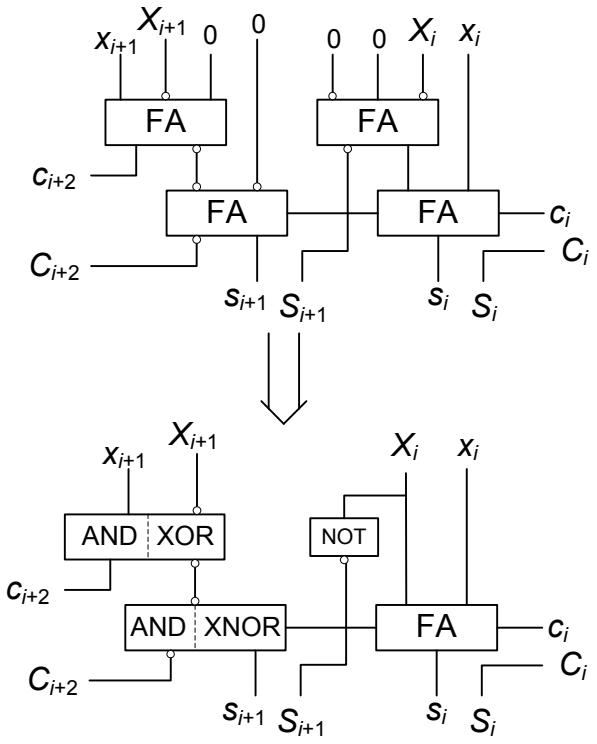


Figure 3.29: BSD Adder with some inputs assigned to zero

The next steps are normalization and rounding which are done using conventional methods for BSD representation [23, 24]. Normalization of redundant operands is more complicated than that of non-redundant operands. The first step to normalize a redundant-represented number is to detect and eliminate the leading non-zero digits of no significance [24]. In other words, there is a need to eliminate non-zero digits whose total values are zero. Fig. 3.30 shows an example of this situation for Binary Signed-Digit representation.

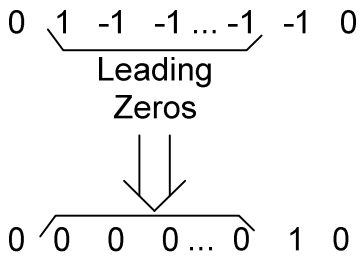


Figure 3.30: Non-zero Digits of No Significance

The next step is to detect the number of leading zeros and then shift the operand to the left accordingly. The leading-zero detector (LZD) can be implemented using a divide-and-conquer approach in which, first, a 2-bit LZD is designed and then larger LZD is built using the basic 2-bit LZD. Detecting number of leading zeros is the same as detecting the leftmost 1 in the bit string. Table 3.2 shows the combinations of two bits and how the leading 1 is detected [25].

Table 3.2: 2-bit Leading-Zero Detection

Pattern	Position	'1' is detected?
1X	0	Yes
01	1	Yes
00	X	No

Table 3.2, basically, says if any '1' is detected and if so it is in position 0 or 1. This unit can be used to build larger LZD. Fig. 3.31 shows a 4-bit LZD implemented using 2-bit LZDs. *D* signals represent if any '1' is detected and *P* signals represent the position of the detected '1'. Using the same approach, 4-bit LZD detectors can be used to build an 8-bit LZD and so on. At the end, the value of *P* shows the number of leading zeros. This value is passed to the barrel shifter to perform left shifts on the operand.

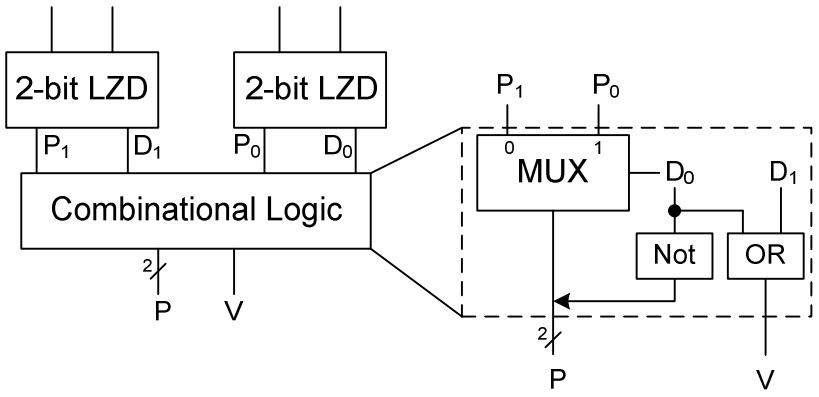


Figure 3.31: 4-bit LZD Implemented Using 2-bit LZD

It should be noted that one could replace the two-input LZD with a four-input leading-zero-anticipation (LZA) [2] for speed-up; but at the cost of more area consumption. The concept behind this LZA is to anticipate the number of leading zeros even before performing the addition. Consequently, for a non-redundant representation, instead of performing LZD on a single number, LZA works on two numbers that are supposed to be added together.

For the operands represented in redundant format, using LZA is more complicated, because the single redundant number consists of more bits. For example, a BSD representation is actually two numbers (one negative- and one positive-weighted). Therefore, LZA has to be performed on four numbers (two negative- and two positive-weighted). This increases the area and latency of the normalization block. That is why it is desired to use LZA and stick to the simpler LZD approach. The rounding part simply determines the round value, based on the Guard and Round positions, and adds it to the digit in the rounding position.

As discussed before the rounding position is usually the least-significant position of the output operand; however, in redundant representation, this position might be moved 1 bit to the right due to the propagating of a -1 value during the conversion to non-redundant representation. It should be noted that adding the round value is a carry-limited operation, thanks to Binary Signed-Digit representation, and hence, can be done in constant time.

The proposed three-point floating-point adder is implemented as shown in Fig. 3.32 in which new alignment and addition blocks are introduced. Moreover, given the sign-embedded representation of the significands (i.e., BSD) there is no need for a *sign logic*.

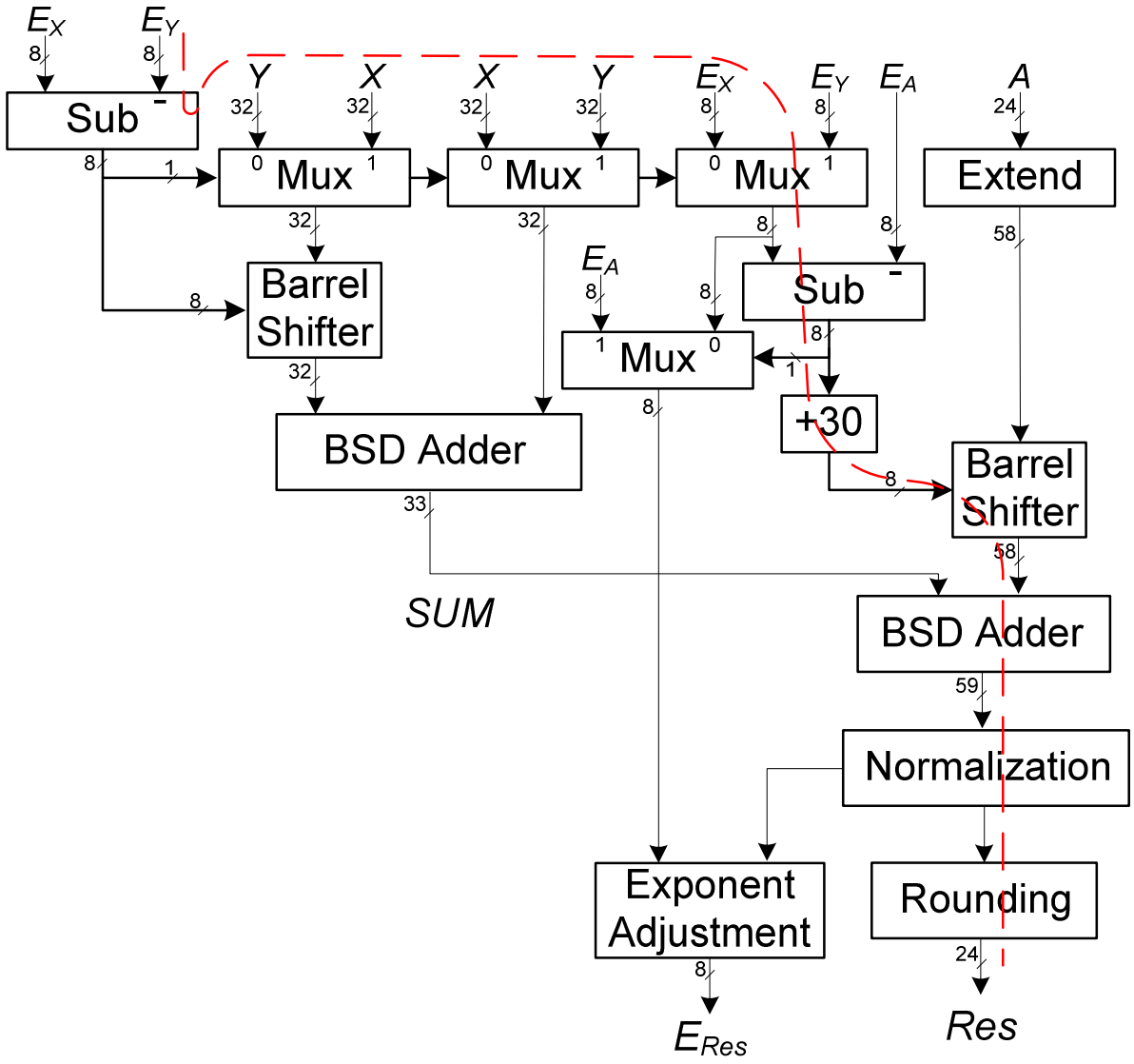


Figure 3.32: The proposed floating-point three-operand addition (critical path is shown in red line)

A comparison of the proposed design with the conventional one is shown in Table 3.3. The critical path of the three-operand adder (as shown in Fig. 3.32) consists of:

- Two 8-bit carry-propagating subtractors (0.25*ns* each)
- A MUX (0.07*ns*)

- A +30 block ($0.17ns$)
- A barrel shifter ($0.29ns$)
- The final BSD adder ($0.16ns$)
- Normalization and Rounding ($0.75ns$)
- Registers ($0.22ns$).

Table 3.3: Comparison of Floating-Point Three-Operand Adders

	Conventional [21]	**Proposed**
Exponent Comparison & Significand Alignment	$3 \times (8\ bit\ Sub)$ $3 \times MUX$ $3 \times Shifter$ *Combinational Logic*	$2 \times (8\ bit\ Sub)$ $4 \times MUX$ $2 \times Shifter$ *+30 Block*
Significand Addition	CSA + CPA	$2 \times BSD\ Adder$
Critical Path	Sub + MUX+ Shifter + Comb. + CSA + CPA	Sub + MUX + Sub + (+30) + Shifter + BSD Adder
Sign Logic	Yes	No
Latency (90*nm* CMOS)	$2.7ns$	$2.16ns$

3.4 Conversion to and from BSD Representation

The conversion to / from BSD representation, and the way it influences the proposed butterfly architecture, are discussed in this sub-section.

The conversion to any redundant format is a carry-free operation, however, the reverse conversion requires carry-propagation [7]. For binary signed-digit (BSD) representation, the forward conversion is straightforward. Assuming the non-redundant sign magnitude representation of a significand as $F: f_s f_{n-1} \dots f_1 f_0$, where f_s is the sign bit, the conversion to n-digit BSD format ($\{R_{n-1} r_{n-1}\} \dots \{R_1 r_1\}\{R_0 r_0\}$) is shown in Fig. 3.33.

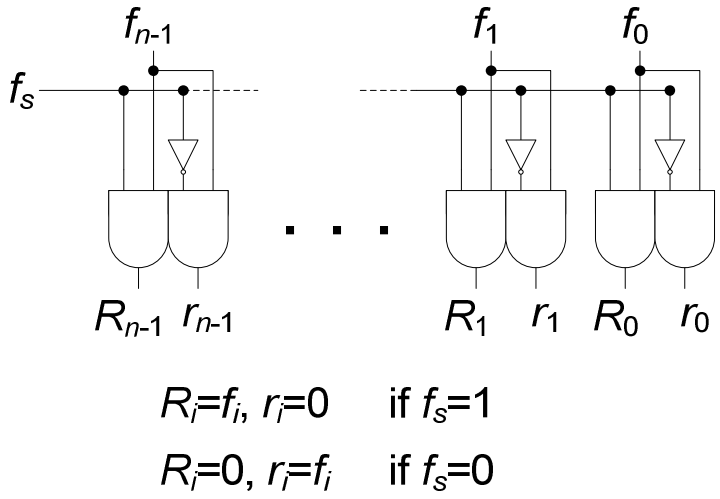


Figure 3.33: Conversion to BSD representation

The reverse conversion is a simple carry-propagating subtractor computing $F = r - R$.

Given that the proposed butterfly architecture is meant to be used in a Fast Fourier Transform (FFT) unit, the reverse conversion is done in the very last iteration of the FFT unit. There might be a need for one more step in the end to convert BSD result to non-redundant representation. This step (i.e., a carry-propagating addition), if not fused by other floating-point conversion operations, adds an extra cycle to the whole FFT unit.

It should be noted that the output of the proposed butterfly unit is represented in redundant format and is fed to the next stage (in FFT) without being converted to non-redundant representation.

Next chapter discusses the performance evaluation of the arithmetic units proposed in this section to be used in the FFT co-processor.

CHAPTER 4

EVALUATIONS AND COMPARISON OF FFT ARCHITECTURES

The evaluation results of the proposed architecture, in terms of latency and area, are presented and compared with previous pertinent works, in this chapter. The proposed design is synthesized by Synopsys Design Compiler using the STM 90nm CMOS standard library [26] for 1.00 VDD and 25°C temperature in which a FO4 latency is 45ps and the area of a NAND2 is $4.4\mu m^2$.

The critical path delay of the proposed butterfly architecture, equal to that of the Fused Dot-Product Add (FDPA), consists of a constant multiplier, a three-operand FP adder plus registers (Table 4.1). It is worth mentioning that the critical path delay of the three-operand adder (including those of the termination phase and register), shown in Table 3.3, is equal to 2.16*ns*. However, since the inputs *X* and *Y*, coming from multipliers, are in the critical path of the FDPA, a different path of the three-operand adder is shown in Table 4.1.

Table 4.1: Critical Path Delay of the Proposed Floating-Point Butterfly Architecture

Module	Components	Delay (*ns*)
Multiplication (1.04 *ns*)	PPG	0.19
	PPR (4 levels of BSD adders)	0.85
Three-operand Addition (0.58 *ns*)	Mux	0.04
	Barrel Shifter	0.20
	BSD Adder 1	0.18
	BSD Adder 2	0.16
Termination (0.75 *ns*)	LZD	0.21
	Normalization & Rounding	0.54
Register		0.22
Total		2.59

The total consumed area of the proposed butterfly unit is evaluated as $375{,}347\mu m^2$ of which $8{,}337\mu m^2$ is for registers. The dynamic and leakage power consumption of this design are about 90.6 mw and 7.6 mw, respectively.

The two major works on floating-point butterfly architecture are in [1, 5]. The work in [5] introduces architectures based on *multiple-constant multipliers* (MCM) and *merged multiple-constant multiplier* (MMCM), amongst which the fastest design reported to have the latency of $4.08ns$ with the area consumption of $97{,}302\mu m^2$, in $45nm$ CMOS technology.

The other work [1] designs a FP butterfly unit using novel dot-product blocks. This work, simulated based on 45nm CMOS technology, has the latency of $4.00ns$ with the area cost of about $47{,}489\mu m^2$. The dot-product unit of this design is reported to have a latency of about 2.72ns with $16{,}104\mu m^2$ area cost.

A recent work [2] has proposed a very fast FP dot-product unit which can be used in the design of a high-performance butterfly unit. Replacing the dot-product unit of [1] with this faster one, leads to a high-speed butterfly architecture. Table 4.2 shows the comparison of the proposed butterfly architecture with those of the previous works. As a result, the proposed design, simulated in $90nm$ (vs $45nm$), is yet much faster than those of previous works. Moreover, scaling the area of the proposed design to $45nm$ technology results in the value of about $\left(\frac{\sqrt{375\,347}}{2}\right)^2 = 93{,}836\mu m^2$ which is almost equal to that of [5].

For the sake of fair comparison the new design is also synthesized using 45nm Nangate Open Cell Library with 1.25 VDD. The area of the proposed design is $56{,}338\mu m^2$ with wire load model: "5K_hvratio_1_1" and the delay constraint set to 3.15ns (i.e., equal to that of the fastest previous works).

Table 4.2: Comparison of the Floating-Point Butterfly Architectures

		Technology	Delay (*ns*)	Area (μm^2)
[5]		45 nm	4.08	97,302
[1]		45 nm	4.00	47,489
[1] + [2]	**Single Path**	45 nm	3.42	60,857
	Dual Path	45 nm	3.15	62,857
Proposed		90 nm	2.59	375,347
		Scaled Area		93,836

Having the butterfly unit synthesized, the area and delay of an *N*-point FFT architecture based on the proposed butterfly unit is evaluated below. As discussed in Chapter two, an *N*-point FFT unit is implemented in $\log_2 N$ stages each of which consists of $N/2$ butterfly units (working in parallel). Therefore,

Latency (90nm technology): $2.59ns \times log_2 N$

(+1 cycle for the conversion to non-redundant format)

Area (90nm technology): $375347\mu m^2 \times \frac{N \log_2 N}{2}$

(+LUT[32×N/2] for storing Twiddle factors)

Although floating-point and fixed-point architectures are not meant to be compared together, a fixed-point butterfly architecture is also designed and synthesized based on the proposed Fused Dot-Product Add (FDPA) unit. Table 4.3 illustrates the critical path delay of the proposed fixed-point butterfly unit.

Floating-point arithmetic has advantages over fixed-point among which, wider dynamic range is of interest in FFT implementation. Contrary to fixed-point representation, where dynamic range is linearly proportional to the number of bits, the dynamic range in floating-point representation grows exponentially with increasing bit width. For example, wider dynamic range increases SNQR (6dB per bit) with lower number of bits.

Table 4.3: Critical Path Delay of the Proposed Fixed-Point Butterfly Architecture (16-bit)

Module	Delay (*ns*)
Multiplication	0.8
Three Operand Adder	0.4
Register	0.2
Total	1.4

One may increase the number of bits in fixed-point representation to cover the range provided by floating-point arithmetic. In this case, for example, 277 bits are required to cover the range of a single-precision floating-point representation (i.e., $2^{277} \cong 2^{23} \times 2^{254}$). Implementing a 277-bit multiplier using the proposed 16-bit multipliers, based on Karatsuba multiplication [27], would lead to a critical path delay of:

16-bit multiplication + 4×(3-operand adder) = 0.8+1.6 = 2.4ns.

The latency of the 277-bit carry-free adder is the same as that of a 16-bit and is equal to 0.4ns. Therefore, overall latency of a 277-bit would be approximately equal to 2.4+0.4+0.2 = 3.0ns VS 2.59ns of the proposed floating-point butterfly unit. Although the fixed-point and floating-point latencies are almost equal, a fixed-point FFT requires post processing operations such as scaling and overflow/underflow concerns.

Furthermore, use of IEEE-754-2008 standard [3] for floating-point arithmetic allows for an FFT co-processor in collaboration with general purpose floating-point processors. This offloads computationally intensive tasks from the processors.

In overall the proposed high-speed floating-point (FP) butterfly architecture is much faster than those of previous works but at the cost of higher area. In order to have a high-performance butterfly unit a new three-operand redundant floating-point adder is developed, and modified Booth encoding is used to speed-up the proposed constant BSD multiplier.

Part II

Decimal Co-Processors

CHAPTER 5
DECIMAL ADDITION[2]

Decimal addition is the most effective operation in a decimal processor, since all other operations are working based on the addition (or subtraction). Therefore, having an efficient decimal adder is of paramount importance in designing high-speed decimal arithmetic units.

Decimal digits are typically represented in binary-coded-decimal (BCD) encoding; in which each digit is represented by four bits. Decimal addition over this representation, conventionally, involves the following challenges:

- *Over-9 detection*: There is a need to detect the sum values over 9 to generate the output carry.
- *+6 Correction*: Generating a decimal carry may lead to +6 correction of the sum digit.

With the intention of overcoming the above difficulties, some alternative encodings can be used. For example, in the Excess-3 encoding [36], a decimal-digit d, represented in four bits, is stored as $d + 3$. Therefore, generating the decimal output carry does not require any over-9 detection.

A redundant representation of [30] with digit-set [-8, 9] does not require over-9 detection, because the maximum represented value in this digit-set is 9. In this representation the transfer digit is stored (does not go through any further addition) in the next higher significant decimal position which leads to a carry-free addition.

[2] Published @ 26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'13)

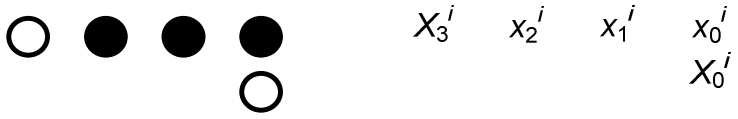
Amongst other efforts on designing a redundant decimal adder e.g., [28, 29, 30], the work done by Gorgin and Jaberipur [31], using the so called partitioning method, shows advantages in terms of both area and delay. In this method the Decimal Septa Signed Digit (DSSD) set $[-7,7]$, represented in four bits, used in the high-speed redundant decimal adder. The transfer bits t_0^{i+1} and T_0^{i+1} ($-1 \leq t_0^{i+1} + T_0^{i+1} \leq 1$) are generated by means of a fast combinational logic and then a semi-carry-propagating adder (i.e., two full-adders (FA), a half-adder (HA) and a NOR gate) produces the final sum. Typically, decimal redundant signed-digit addition (i.e., $X + Y + T_{in} = 10T_{out} + S$) is implemented via the following main steps, for $[-\alpha, \beta]$ as the digit-set.

- Generate the intermediate sum P and transfer set T_{out} such that $-\alpha + 1 \leq P \leq \beta - 1;\ X + Y = P + 10T_{out}$.
- Produce the final result as $S = P + T_{in}$.

It is an improved redundant decimal adder, based on the partitioning method, proposed here in this thesis in which $[-9,7]$ is used as the digit-set so as to allow using stored-carry representation [32] of the operands and hence a faster design.

5.1 The Proposed Redundant Decimal Adder

The main specification of the proposed adder is the stored-carry representation of a digit which leads to $[-9,7]$ as the digit-set of the proposed radix-10 adder. In the stored-carry representation, the carry is stored in the next higher digit and does not go through any further addition process. This leads to carry-free addition and increases the addition speed. Fig. 5.1 depicts a digit representation of the operands used in the proposed adder where black (white) dots symbolize the positive- (negative-) value bits.



(a) (b)

Figure 5.1: Digit representation (a) Dot notation (b) Symbolic notation

The proposed adder is designed based on Algorithm 5.1, reproduced from the conventional partitioning addition algorithm [31].

Algorithm 5.1 (The proposed addition):

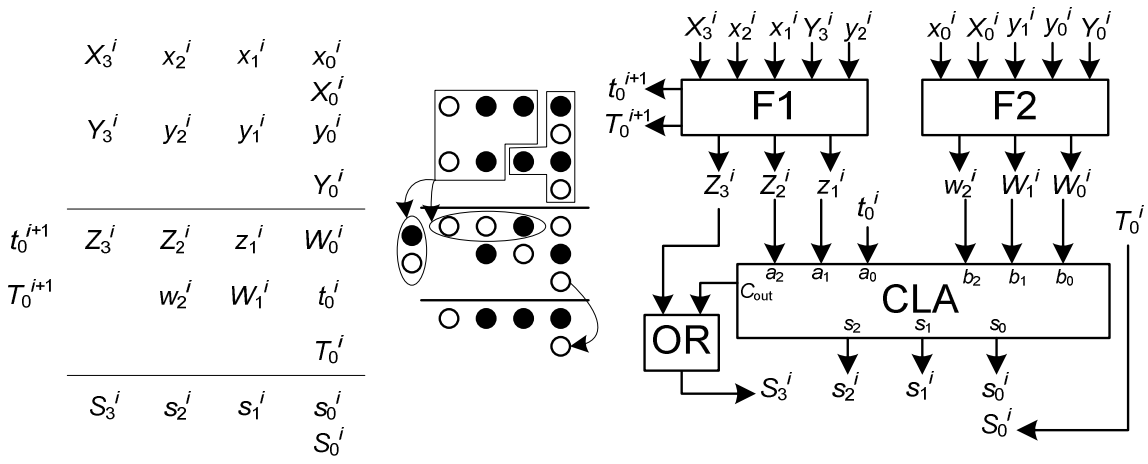
Inputs: Two *n*-digit numbers $X: X^{n-1} \dots X^1 X^0$, $Y: Y^{n-1} \dots Y^1 Y^0$, where a digit (e.g., i^{th}) is represented by five bits as $X^i: X_3^i x_2^i x_1^i x_0^i X_0^i$.

Output: One n-digit number $S: S^{n-1} \dots S^1 S^0$, where a digit (e.g., ith) is represented by five bits as $S^i: S_3^i s_2^i s_1^i s_0^i S_0^i$; and two transfer bits t_0^n, T_0^n.

Perform the following steps, for $0 \leq \text{i} \leq n-1$ in parallel.

a) Transfer bits t_0^{i+1}, T_0^{i+1} and high-portion of the interim sum Z_3^i, Z_2^i, z_1^i are generated by a combinational logic $\text{F1}(X_3^i, Y_3^i, x_2^i, y_2^i, x_1^i)$.
b) Low-portion of the interim sum w_2^i, W_1^i, W_0^i is generated by a logic $\text{F2}(y_1^i, y_0^i, Y_0^i, x_0^i, X_0^i)$.
c) Generate the final sum as $S_3^i s_2^i s_1^i s_0^i = Z_3^i Z_2^i z_1^i t_0^i + w_2^i W_1^i W_0^i$ and $S_0^i = T_0^i$. ■

Steps *a* and *b*, consist of combinational logics, are performed simultaneously to generate the whole interim sum and the transfer bits. Next, Step *c* computes the final sum through a fast 3-bit Carry-Look-ahead Adder (CLA) and an OR gate. The dot notation, symbolic notation and block diagram of the proposed adder are shown in Fig. 5.2. The details of the constituent blocks of this figure are presented in the following. The main idea here is that the output carry bits are generated based on only the most significant bits of a decimal digit (i.e., five most significant bits in this case). The total value of the rest of the bits are not large enough to produce any carry to the next digit. This reduces the complexity of the carry-generation logic, since it is designed based on only five bits (instead of ten).



(a) (b) (c)

Figure 5.2: The Proposed Adder (a) Symbolic Notation (b) Dot Notation (c) Block Diagram

Moreover, the negative-weighted carry does not go through any further addition while the positive-weighted carry serves as the least significant bit of the carry-propagating adder. The F1 and F2 blocks are responsible for Steps *a* and *b* of Algorithm 5.1. F1 is a five input five output logic and F2 generates three outputs from the five inputs. These blocks implement the logical expressions derived from the truth tables shown in Tables 5.1 and 5.2, respectively.

The 3-bit CLA is a fast adder which adds two 3-bit operands without any input carry. The logical expressions of the outputs of this CLA are presented in Eqn. 5.1. It should be noted that the negative-weighted inputs must be negated prior to be injected into the adder. Likewise, the negative-weighted outputs must be negated to represent the correct value.

$$\begin{aligned} s_0^i &= a_0 \oplus b_0 \\ s_1^i &= a_1 \oplus b_1 \oplus (a_0 b_0) \\ s_2^i &= a_2 \oplus b_2 \oplus (a_0 b_0 a_1 + a_0 b_0 b_1 + a_1 b_1) \\ C_{out} &= a_2 b_2 + [(a_2 + b_2)(a_0 b_0 a_1 + a_0 b_0 b_1 + a_1 b_1)] \end{aligned} \tag{5.1}$$

Finally, an OR gate is used to generate the most significant bit of the final sum S_3^i. The OR gate is good enough here to produce the most significant digit, since at least one of C_{out} or

Z_3^i must have zero value. It is worth mentioning that C_{out} is needed to be a negabit; therefore, a negative-weighted carry must be propagated through the 3-bit CLA. Consequently, the least significant bit of the CLA must take a zero-value negative-weighted bit (represented as '1'). In this case, there are two negabits and one posibit in the least-significant position of the CLA. Adding these bits together generates a positive-weighted sum bit and a negative-weighted carry to the next position. It is the same situation for the next two positions of the CLA. This guarantees that C_{out} is a negative-weighted bit.

Table 5.1: Truth Table of F1

X_3^i	Y_3^i	x_2^i	y_2^i	x_1^i	Value	T_0^{i+1}	t_0^{i+1}	Z_3^i	Z_2^i	z_1^i
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	2	0	0	0	0	1
0	0	0	1	0	4	0	1	1	0	1
0	0	0	1	1	6	0	1	0	1	0
0	0	1	0	0	4	0	1	1	0	1
0	0	1	0	1	6	0	1	0	1	0
0	0	1	1	0	8	0	1	0	1	1
0	0	1	1	1	10	0	1	0	0	0
0	1	0	0	0	–8	1	0	0	0	1
0	1	0	0	1	–6	0	0	1	0	1
0	1	0	1	0	–4	0	0	0	1	0
0	1	0	1	1	–2	0	0	0	1	1
0	1	1	0	0	–4	0	0	0	1	0
0	1	1	0	1	–2	0	0	0	1	1
0	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	2	0	0	0	0	1
1	0	0	0	0	–8	1	0	0	0	1
1	0	0	0	1	–6	0	0	1	0	1
1	0	0	1	0	–4	0	0	0	1	0
1	0	0	1	1	–2	0	0	0	1	1
1	0	1	0	0	–4	0	0	0	1	0
1	0	1	0	1	–2	0	0	0	1	1
1	0	1	1	0	0	0	0	0	0	0
1	0	1	1	1	2	0	0	0	0	1
1	1	0	0	0	–16	1	0	1	0	1
1	1	0	0	1	–14	1	0	0	1	0
1	1	0	1	0	–12	1	0	0	1	1
1	1	0	1	1	–10	1	0	0	0	0
1	1	1	0	0	–12	1	0	0	1	1
1	1	1	0	1	–10	1	0	0	0	0
1	1	1	1	0	–8	1	0	0	0	1
1	1	1	1	1	–6	0	0	1	0	1

Table 5.2: Truth Table of F2

y_1^i	y_0^i	Y_0^i	x_0^i	X_0^i	Value	w_2^i	W_1^i	W_0^i
0	0	0	0	0	0	0	0	0
0	0	0	0	1	–1	0	0	1
0	0	0	1	0	1	1	1	1
0	0	0	1	1	0	0	0	0
0	0	1	0	0	–1	0	0	1
0	0	1	0	1	–2	0	1	0
0	0	1	1	0	0	0	0	0
0	0	1	1	1	–1	0	0	1
0	1	0	0	0	1	1	1	1
0	1	0	0	1	0	0	0	0
0	1	0	1	0	2	1	1	0
0	1	0	1	1	1	1	1	1
0	1	1	0	0	0	0	0	0
0	1	1	0	1	–1	0	0	1
0	1	1	1	0	1	1	1	1
0	1	1	1	1	0	0	0	0
1	0	0	0	0	2	1	1	0
1	0	0	0	1	1	1	1	1
1	0	0	1	0	3	1	0	1
1	0	0	1	1	2	1	1	0
1	0	1	0	0	1	1	1	1
1	0	1	0	1	0	0	0	0
1	0	1	1	0	2	1	1	0
1	0	1	1	1	1	1	1	1
1	1	0	0	0	3	1	0	1
1	1	0	0	1	2	1	1	0
1	1	0	1	0	4	1	0	0
1	1	0	1	1	3	1	0	1
1	1	1	0	0	2	1	1	0
1	1	1	0	1	1	1	1	1
1	1	1	1	0	3	1	0	1
1	1	1	1	1	2	1	1	0

5.2 Evaluations and Comparisons of Decimal Redundant Adders

The evaluation of the proposed design and the comparison with the fastest previous work [31] is presented in this section. Both the proposed design and that of [31] are simulated via Verilog codes and synthesized by Synopsys Design Compiler using the STM 90nm CMOS standard library [26] for 1.00 VDD and 25°C temperature, where the FO4 latency is 45ps and the area of a NAND2 gate is 4.4μm^2.

5.2.1 Decimal Redundant Adder of [31]

The structure of this adder [31], shown in Fig. 5.3, is similar to that of Fig. 5.2. There are two critical paths for this adder as follows;

1) The combinational logic which generates the negative-weighted transfer bit plus two FAs, one HA and the NOR gate.
2) The combinational logic that generates *Z* plus an FA, an HA and the NOR gate.

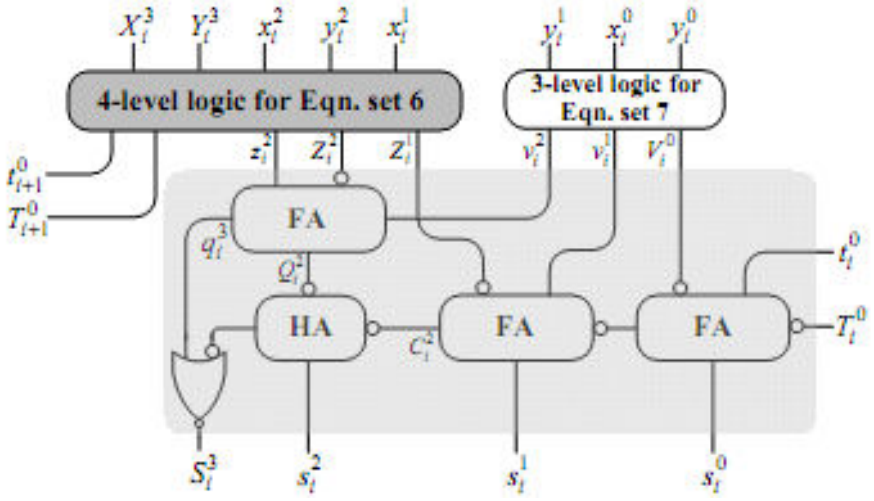


Figure 5.3: Decimal Redundant Adder of [31]

It should be noted that in the critical path 1) the positive-weighted transfer bit is generated much faster than the negative-weighted one and hence out of the critical path.

One might be thinking of speeding up the critical path 1), by replacing the cascaded FAs and HA by a fast CLA. However, this would not reduce the latency since in this case path 2) would be the critical path. The synthesis results show 23ns as the minimum latency of this adder with the area of $1{,}665\mu m^2$.

5.2.2 The Proposed Decimal Redundant Adder

In this adder some techniques are used to speed up both of the critical paths of [31]. Path 1) is improved by using a CLA and not feeding the negative-weighted transfer bit into this CLA (i.e., store it as the least significant negabit). Path 2) is improved by changing the representation of the intermediate sum and hence removing the FA of path 2) in [31]. However, the above improvements lead to the latency and area overhead to the combinational logic F2 which is not in the critical path delay. The evaluation results show 20ns as the minimum latency of this adder with the area of $2055\mu m^2$. The comparison of the proposed design with that of [31] is shown in Table 5.3, in terms of latency, area and power. According to Table 5.3 the proposed decimal signed-digit adder shows 15% delay advantage over the work of [31], but at the expense of 23% more area and 38% more power consumption.

Table 5.3: Comparison of Decimal Redundant Adders

	Delay (ns)	**Ratio**	**Area (μm^2)**	**Ratio**	**Power (μw)**	**Ratio**
Proposed	0.20	1	2055	1	600	1
[31]	0.23	1.15	1665	0.81	435	0.72

A decimal signed digit adder using the stored carry representation of the operands is proposed in this chapter. Using $[-9,7]$ as the digit-set simplifies the final addition (CLA) by removing the input carry.

CHAPTER 6

DECIMAL MULTIPLICATION[3]

The complexity of decimal arithmetic makes more challenges to be dealt with during the hardware implementation, compared to the binary counterparts. One of these challenges is the high area cost of the decimal operations, due to inefficient decimal encodings (non-power-of-two). Amongst the four basic operations i.e., addition, subtraction, multiplication and division; the latter two are considered as those with large area costs.

With the intention of reducing the high cost of these decimal operations the processor industry has opted for sequential realization [9, 33, 34]. Sequential hardware realization is at a disadvantage due to its high latency. For reducing the impact of this drawback, one can manipulate the cycle time (i.e., increasing the frequency) or the number of iterations (moving toward semi-parallel and parallel implementations). Recalling that the latter solution leads to the higher area cost, which as mentioned is not of interest to the processor industry, focusing on the cycle time seems to be more efficient.

This chapter focuses on the hardware realization of the decimal multiplication where a novel architecture is proposed to overcome the problem of slow sequential decimal multipliers via increasing the clock frequency. In this design, the cycle time is reduced to the latency of a binary half-adder (HA) plus that of a decimal multiply-by-two operation, which is in overall less than that of a decimal carry-save adder. The claimed improvement is supported by synthesizing and comparing the proposed design to those of the best previous pertinent works.

The proposed design includes the following novelties, with respect to previous works:

[3] Published @ 1) IET Electronics Letters, 2) ISCAS'12

- Designing a novel selection block to select appropriate easy-multiples of the multiplicand as a partial product.
- Proposing a novel approach for the partial product accumulation (PPA) with the intention of increasing the speed of this crucial step. The critical path delay of a conventional PPA consists of the delay of two decimal carry-save adders (CSA) while the latency of the proposed PPA is reduced to less than that of a mere decimal CSA by retiming the constituent parts of a decimal CSA.

6.1 Decimal Multiplication Overview

Decimal multiplication is a dyadic operation performed on two *n*-digit decimal numbers, called multiplicand X and multiplier Y, and computes the final 2*n*-digit product P; where X, Y and P (assumed to be represented in the standard BCD format) are shown in Eqn. 6.1. The capital (small) letters represent a decimal number (digit), while a subscript index indicates the position of the decimal number (digit).

$$X = \sum_{i=0}^{n-1} x_i \times 10^i ; \quad Y = \sum_{i=0}^{n-1} y_i \times 10^i ; \quad x_i, y_i, p_i \in [0,9]$$
$$P = \sum_{i=0}^{2n-1} p_i \times 10^i = X \times Y = X \sum_{i=0}^{n-1} y_i \times 10^i \tag{6.1}$$

According to Eqn. 6.1, for computing the final product one requires to perform $x_j \times y_i \times 10^{j+i}$, iteratively. The $\times 10^{j+i}$ operation is a mere (j + i)-digit left shift and $x_j \times y_i$ is known as the BCD digit multiplication [35], defined below.

Definition 6.1 (BCD digit multiplication): Given two BCD digits $0 \leq x_j, y_i \leq 9$ as inputs, compute the digit-product (two BCD digits) as $10p_{i,j}^{h} + p_{i,j}^{l} = y_i \times x_j$ and $p_{i,j}^{h} \in [0,8]$, $p_{i,j}^{l} \in [0,9]$ ■

Performing the BCD digit multiplication for all digits of the multiplicand (i.e., x_j for $0 \leq j < n$) with a single digit of the multiplier y_i generates, according to Eqn. 6.1, $P_i = y_i \times X$ which is traditionally called the (i^{th}) partial product.

Definition 6.2 (Partial product generation): Given an *n*-digit BCD number X and a BCD digit $0 \leq y_i \leq 9$, generate the (*n*+1)-digit partial product P_i as follows, where $p_{i,-1}^{h} = 0$.

$$P_i = \sum_{j=0}^{n} p_{i,j} \times 10^j = y_i \times X = \sum_{j=0}^{n-1} y_i \times x_j \times 10^j \overset{Def.1}{\Longrightarrow} 0 \leq p_{i,j} = p_{i,(j-1)}^{h} + p_{i,j}^{l} \leq 17 \quad \blacksquare$$

Observation 6.1: Regarding Definition 6.2, it is note that the cardinality of the digit-set of the partial products is greater than the radix (i.e., 18 > 10 in Definition 6.2) and hence redundant representations. Therefore, a carry-propagating digit-set conversion is required to have non-redundant partial products. ■

Computer arithmetic literature abounds with diverse hardware algorithms for decimal partial product generation (PPG), among which two algorithms have gained more popularity; namely PPG based on digit-multiplication (Algorithm 6.1) and PPG based on computing easy-multiples of the multiplicand (Algorithm 6.2) [36].

Definition 6.3 (Easy-multiples of the multiplicand): Given the multiplicand *X* represented in the standard BCD format, the mX ($m \in [0,9]$) is considered as an easy-multiple if and only if mX

i) is a non-redundant BCD number;

ii) can be computed from X in a constant time i.e., without the word-wide carry propagation. ■

Algorithm 6.1 (PPG based on digit-multiplication):

Inputs: A BCD digit y_i and an n-digit BCD number $X = \sum_{i=0}^{n-1} x_i \times 10^i$.

Output: A (n+1)-digit decimal number $P_i = y_i \times X = \sum_{j=0}^{n} p_{i,j} \times 10^j$.

Method:

For $0 \leq j < n$ do

I. Compute $10p_{i,j}^h + p_{i,j}^l = y_i \times x_j$ via BCD digit multiplication (Definition 6.1).

II. Compute $p_{i,j} = p_{i,(j-1)}^h + p_{i,j}^l$; where $p_{i,-1}^h = 0$.

III. Assign $p_{i,n} = p_{i,(n-1)}^h$. ■

Algorithm 6.2 (PPG based on easy-multiples of the multiplicand):

Inputs: A BCD digit y_i and an n-digit BCD number X.

Output: An (n+1)-digit decimal number $P_i = y_i \times X$.

Method:

I. Compute the required easy-multiples of the multiplicand in a constant time.

II. y_i selects $U_i, V_i \in \{\text{easy multiples}\}$ such that $P_i = U_i + V_i$. ■

In order to achieve the final product, according to Eqn. 6.1, one needs to generate and sum up the partial products for all digits of the multiplier (i.e., y_i for $0 \leq i < n$). This is known as partial product accumulation defined below.

Definition 6.4 (Partial product accumulation): Given *n* partial products (P_i for $0 \leq i < n$) as inputs, compute the 2*n*-digit final product $P = \sum_{i=0}^{2n-1} P_i \times 10^i$. ■

The sequential decimal multiplier, in essence, generates one partial product (e.g., P_i) per iteration, using either Algorithm 6.1 or 6.2; next the recurrence of Eqn. 6.2, performed in the i^{th} iteration, takes care of the PPA ($P[0] = 0$ and $P = P[n]$). In Eqn. 6.2, P[i] denotes the accumulated partial products after *i* iterations.

$$P[i+1] = P[i] + 10P_i \tag{6.2}$$

6.2 The Proposed Sequential Decimal Multiplier

The details of the proposed sequential decimal multiplier are presented in this section where the first part is devoted to elaborate on the architecture of the PPG step and next the proposed PPA architecture is described.

6.2.1 Partial Product Generation

The partial products, in the proposed design, are generated via Algorithm 6.2 with easy-multiples of X, 2X, 4X, 5X. It should be noted that although there are other set of easy-multiples (e.g., −X, −2X and 10X [37, 38]), the mentioned easy-multiples are used to avoid dealing with negative numbers in the PPA.

According to Algorithm 6.2, there is a need to compute the easy-multiples of the multiplicand. With the intention of lowering the area cost, the decimal 4-2-2-1 encoding is used for the multiples X, 2X, 4X, 5X as shown in Fig. 6.1, adapted from [38].

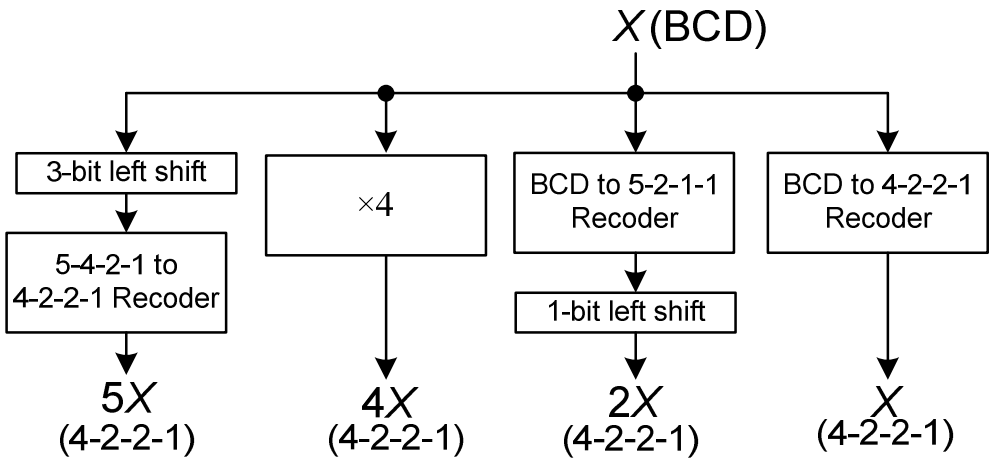


Figure 6.1: Generation of the easy-multiples in the proposed multiplier

Given this figure, the PPG mostly consists of wired shifts and various recoders the same as done in [39]. However, the 4X is generated directly from *X*, not by cascading two 2X blocks,

as to manage to perform PPG in one cycle. Consequently, the area of the proposed PPG is slightly higher than the previous counterparts [38, 39].

According to Algorithm 6.2, the next step after computing the easy-multiples is to select values for $U_i, V_i \in \{0, X, 2X, 4X, 5X\}$, regarding $y_i: y_i^3 y_i^2 y_i^1 y_i^0$, so as to generate the i^{th} partial product $P_i = U_i + V_i$. This is done through the logical expressions of Eqn. 6.3 (bit slice) derived based on Table 6.1.

Table 6.1: Selection of the easy-multiples

$y_i =$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
$U_i =$	0	X	0	X	$4X$	$5X$	$4X$	$5X$	$4X$	$5X$
$V_i =$	0	0	$2X$	$2X$	0	0	$2X$	$2X$	$4X$	$4X$

$$V_i = 2X y_i^1 \vee 4X y_i^3$$
$$U_i = X\left[\overline{y_i^3}\,\overline{y_i^2} y_i^0\right] \vee 4X\left[(y_i^2 \vee y_i^3)\overline{y_i^0}\right] \vee 5X[(y_i^2 \vee y_i^3) y_i^0] \quad (6.3)$$

It should be noted that the addition $U_i + V_i$ is not actually performed and hence redundant partial remainder P_i, represented by two 4-2-2-1 decimal numbers.

6.2.2 Partial Product Accumulation

Partial product accumulation is meant to realize Eqn. 6.2 for $P_i = U_i + V_i$ which leads to Eqn. 6.4 as the multiplication recurrence for $0 \leq i < n$.

$$P[i+1] = 0.1P[i] + U_i + V_i \quad (6.4)$$

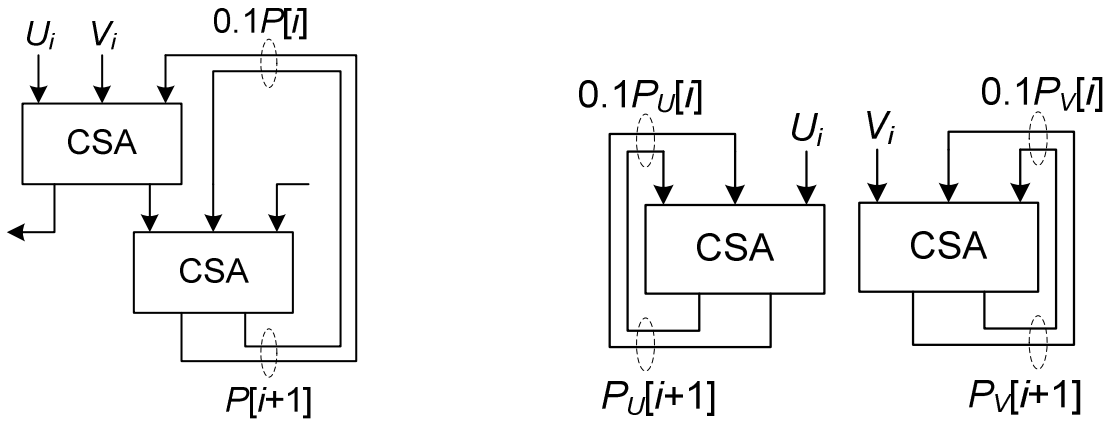
Given the 4-2-2-1 representation of U_i and V_i, one may implement Eqn. 6.4 in an straightforward approach in which P[i] is assumed to be represented in 4-2-2-1 encoding and hence a three-operand 4-2-2-1 addition. However, this causes carry-propagation per iteration which is not of interest for a high-frequency multiplier.

Using decimal carry-free adders (e.g., [30]) while keeping P[i] in a redundant representation is traditionally a solution for removing the carry-propagation per iteration. This modification typically calls for decimal (4:2)-compressors (i.e., two cascaded decimal CSAs) with 4-2-2-1 encodings [38]. Moreover, after n iterations a conversion from the redundant to non-redundant BCD format is required to generate the final product.

Intending for a high-speed PPA, it is noted that Eqn. 6.4 can be divided into two recurrences (Eqn. 6.5) each of which executed, independently, per iteration. Finally, after n iterations the outcomes of these two recurrences should be merged together and converted into the non-redundant BCD format.

$$P_V[i+1] = 0.1P_V[i] + V_i$$
$$P_U[i+1] = 0.1P_U[i] + U_i \tag{6.5}$$

Fig. 6.2 highlights the differences of the proposed versus the conventional PPA approach [40].



(a) (b)

Figure 6.2: PPA (digit-slice) (a) Conventional (b) Proposed

The proposed technique for the partial product accumulation is shown in Algorithm 6.3.

Algorithm 6.3 (The proposed PPA):

Inputs: Partial products $P_i = U_i + V_i$ for $0 \leq i < n$.

Output: The $2n$-digit final product $\mathrm{P} = \sum_{\mathrm{i}=0}^{2\mathrm{n}-1} \mathrm{P_i} \times 10^{\mathrm{i}}$.

Method:

I. Compute Eqn. 6.5, iteratively, for $0 \leq \mathrm{i} < n$ with $\mathrm{P_U}[0] = \mathrm{P_V}[0] = 0$.

II. $P = P_U[n] + P_V[n]$. ■

Step *a* of Algorithm 6.3 can be implemented via decimal CSAs with 4-2-2-1 encodings, assuming the redundant representation for $\mathrm{P_V}[\mathrm{i}+1]$ and $\mathrm{P_U}[\mathrm{i}+1]$. The details of these CSAs are illustrated in Fig. 6.3, adapted from [38].

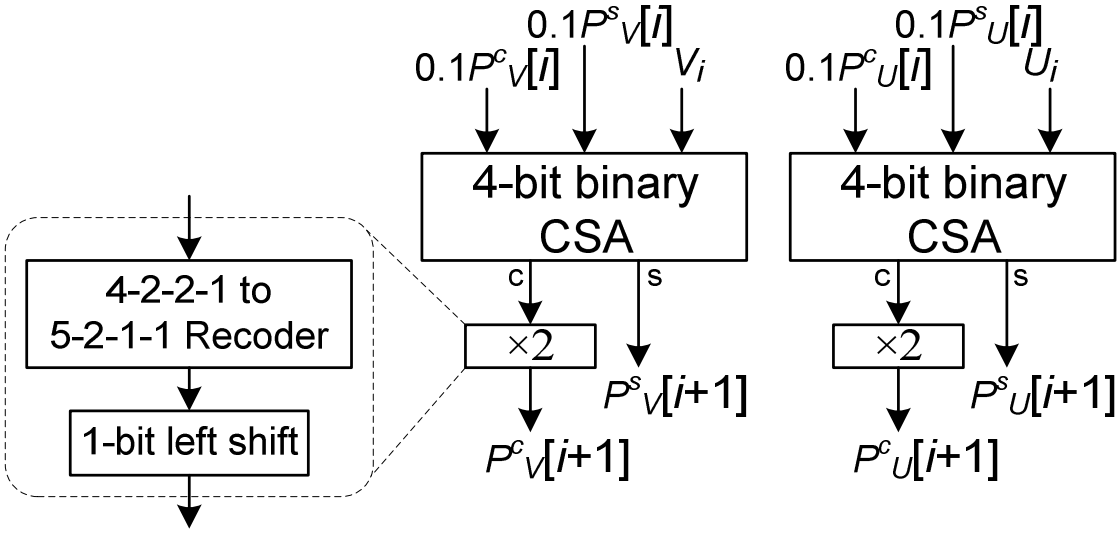


Figure 6.3: Implementation of Eqn. 6.5 via CSAs

The latency of the circuitry of Fig. 6.3 (i.e., the cycle time of the multiplier) can be reduced by postponing the operation of $\times 2$ until the next iteration. Consequently, the binary CSA and $\times 2$ can be performed, partially, in parallel as shown in Fig. 6.4 where the dashed lines highlight the critical paths i.e., $\times 2$ plus an HA.

According to the implementation of Fig. 6.4, after *n* iterations there are four decimal numbers represented in 4-2-2-1 encoding which should be merged and converted to the standard BCD format in order to cope with Step *b* of Algorithm 6.3. This is done via two decimal CSAs and a BCD CPA [41].

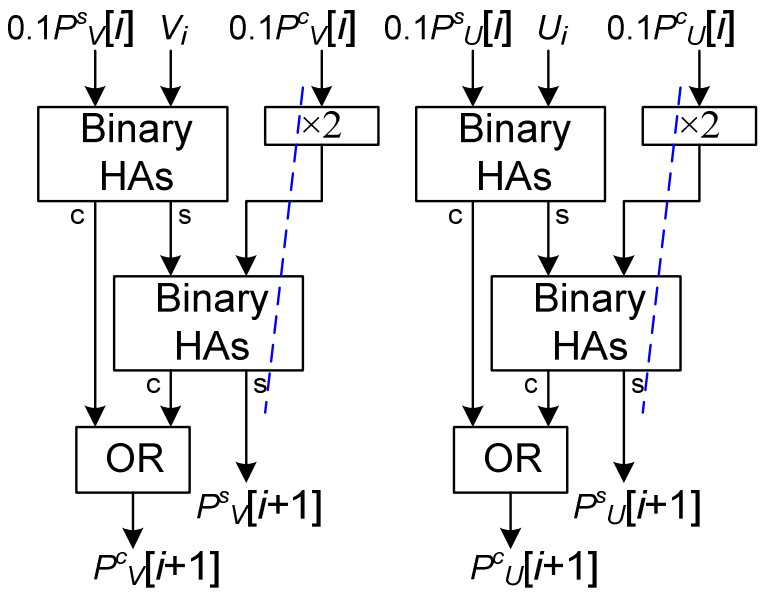


Figure 6.4: Modified Implementation of Eqn. 6.5

The whole architecture of the proposed sequential multiplier (including the PPG and PPA) is shown in Fig. 6.5, where the ($\times 2^*$) block performs both $\times 2$ and conversion to BCD.

6.3 Evaluation and Comparison of Decimal Sequential Multipliers

The evaluation results of the proposed multiplier, in terms of latency and area, are presented in this section where previous works are also examined and compared with. The architectures are compared based on the required number of cycles for a single multiplication, the cycle time and area. The entire proposed design is synthesized by Synopsys Design Compiler using the STM 90nm CMOS standard library [26] for 1.00 VDD and 25°C temperature in which the FO4 latency is 45ps and the area of a NAND2 gate is 4.4μm^2.

The proposed architecture is simulated for both pipelined and word-serial implementations, where in the former the throughput is higher and the latter saves more area cost by reusing a hardware for most of the iterations [17]. However, the latter costs an extra multiplexer added to the critical path delay.

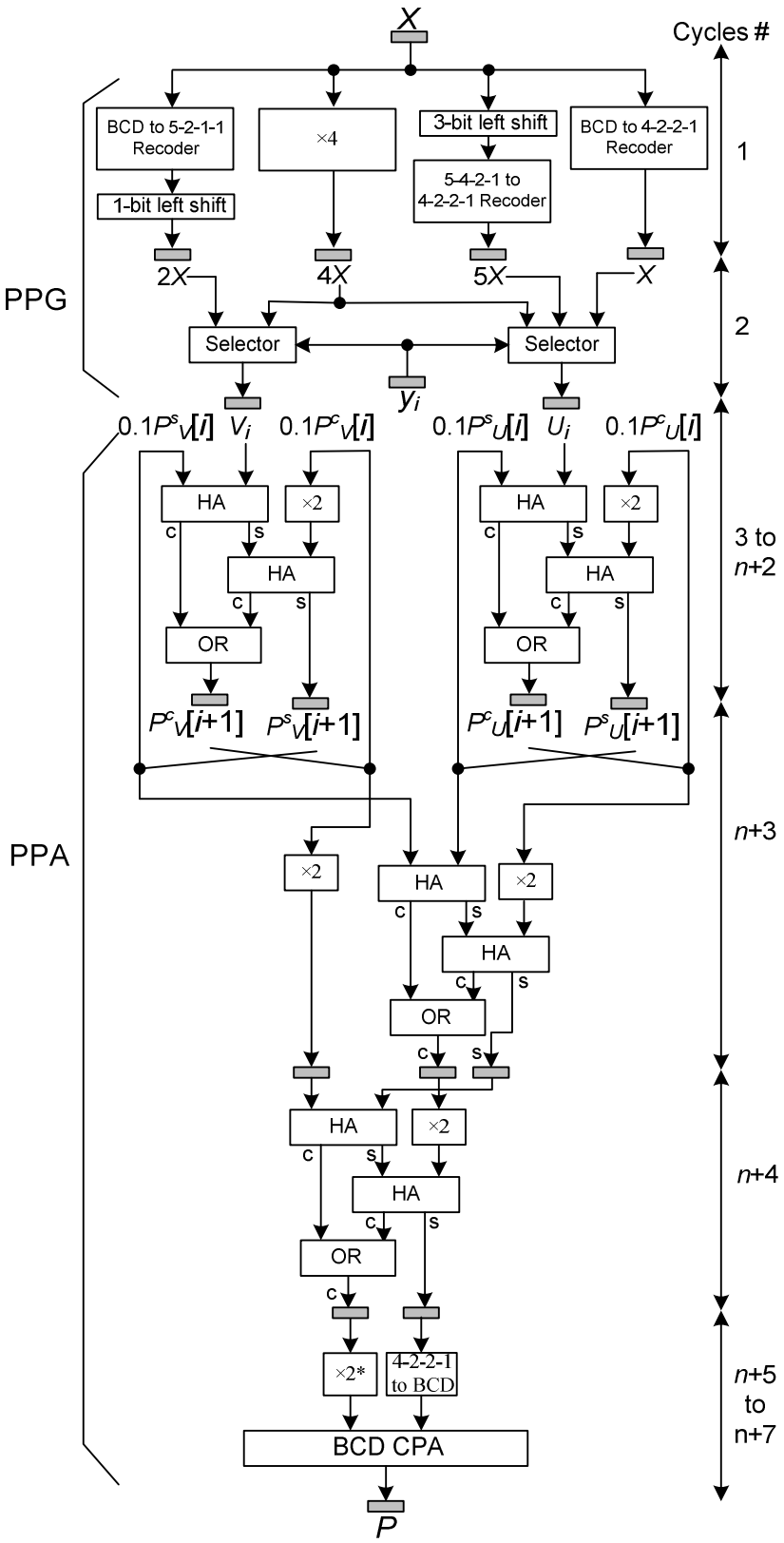


Figure 6.5: The Proposed Sequential Decimal Multiplier

6.3.1 The Proposed Architecture

According to Fig. 6.5 the proposed multiplier consists of two main parts namely PPG and PPA. The PPG phase consumes one cycle for generating the easy-multiples (once in the whole multiplication) and one cycle per iteration for generating the partial products, iteratively. The PPA phase accumulates partial products iteratively in n cycles and then five cycles are required for merging and conversion to produce the final product. Therefore, the entire single multiplication can be performed in n+7 cycles while the next multiplication can be commenced every n+1 cycles.

The cycle time, thus the clock frequency, is determined by the critical path of the iterative part in the PPA consists of the $\times 2$ block and a binary half-adder. The details of the critical path are shown in Table 6.2 which imposes the clock frequency of 2.22 GHz for 90nm CMOS technology.

Table 6.2: The Critical Path Delay of the Proposed Multiplier (*ns*)

	Mux	$\times 2$	HA	Register	Total
Pipelined	---	0.14	0.11	0.2	0.45
Word Serial	0.1	0.14	0.11	0.2	0.55

The area consumption of the proposed 16-digit multiplier is evaluated as the sum of the area cost of various constituent parts tabulated in Table 6.3.4

Table 6.3: Area Consumption of the Proposed 16-digit multiplier (μm^2)

	Area	
	Pipelined	**Word-serial**
PPG	2414	2414
PPA	13263	9041
Registers	7048	5443
Total	22725	16898

6.3.2 Previous Works on Decimal Sequential Multiplier

The multiplier in [40] is based on BCD compressors. It requires n+4 cycles for executing a single multiplication while the next operation can begin every n+1 cycles. The cycle time of this design is equal to the latency of a BCD (4:2)-compressor plus that of registers. The employed compressor is meant to add two BCD digits and two carry bits, per digit. According to [38] the cycle time of this design is evaluated to be equal to 16 FO4 with the area of about 16,000 NAND2 for a 16-digit multiplication.

It is the overloaded decimal representation used in [42] for representing the intermediate redundant operands which calls for a special decimal carry-free adder. This design strategy leads to a critical path consists of a (4:1) multiplexer, a +6 increment block, a binary full-adder plus registers. The latency of this path according to [38] is evaluated to be 12.7 FO4 where the number of cycles required for a single multiplication is n+8 with the initiation interval of n+1 cycles. The area of this multiplier for 16-digit operands is reported as 31,500 NAND2.

The multiplier in [43] takes advantage of the decimal signed-digit adder, introduced in [28], for the iterative portion of the PPA whose latency plus registers (i.e., 14.7 FO4) determines the cycle time. The number of required cycles is the same as [40] i.e., n+4 and the area cost is reported as 18,550 NAND2 for a 16-digit multiplication.

In accordance with the above discussions, Table 6.4 illustrates the details of the evaluation results and compares the proposed design with others in terms of latency and area. Moreover, the simulation results of the proposed multiplier based on delay constraints are depicted in Fig. 6.6. It is shown that the proposed pipelined design, with the cycle time of 10 FO4, is much faster than the previous works while the proposed word-serial implementation keeps the area as low as possible. It is concluded that the fastest previous design [34] works with

47% slower clock frequency, which leads to 27% slower multiplier, than the proposed pipelined architecture.

Table 6.4: Comparison of Decimal Sequential Multipliers (16-digit multipliers)

			Cycle time (FO4)	# of cycles	Total Latency (FO4)	Ratio	Area (NAND2)	Ratio
Ours	Proposed	Pipelined	10.0	23	230	1	22 725	1
		Word Serial	12.2	23	281	1.22	16 898	0.74
[40]			16.0	20	320	1.39	16 000	0.70
[42]			12.7	24	305	1.32	31 500	1.38
[43]			14.7	20	294	1.27	18 550	0.81

The evaluation and comparison results of Table 6.4 reveal the undisputed advantage of the proposed sequential decimal multiplier over the previous designs. The advantage of the proposed architecture could be more highlighted by considering the 34-digit decimal multiplication.

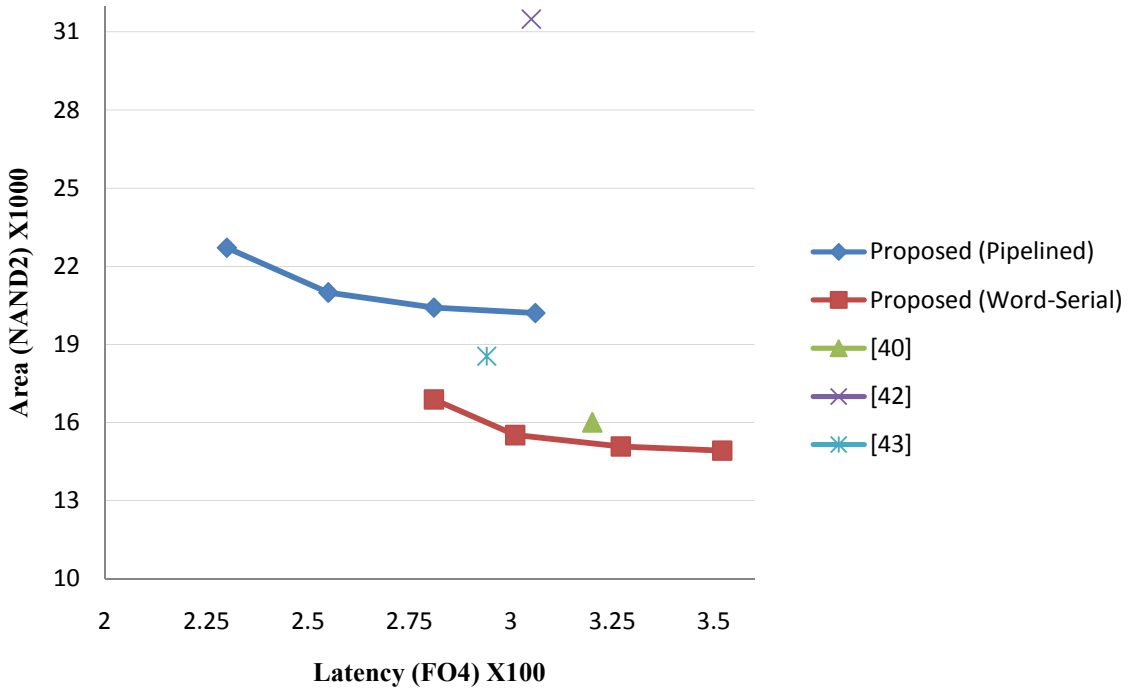


Figure 6.6: Delay Constrained Comparison

CHAPTER 7

DECIMAL DIVISION[4]

Implementing division algorithms in software (computer program) or hardware (digital circuit) is the most costly one of the four basic arithmetic operations. Decimal division algorithm, whose popularity gathered pace by the inclusion of decimal floating-point in the IEEE 754-2008 standard [3], is even more costly. This algorithm is exemplified in Eqn. 7.1, where $0.1 \leq X, D < 1,\ 0 < \mathrm{Q} < 10$ and R are the normalized dividend, normalized divisor, quotient and remainder, respectively.

$$X = Q \times D + R \tag{7.1}$$

Division hardware is usually designed based on digit recurrence or functional algorithms. The former, having latency linearly dependent on the quotient length, leads to lower hardware complexity. The latter, however, due to the use of a sequence of multiplications, progressively produce an approximation of the quotient, where the number of required iterations is logarithmically proportional to the number of quotient digits. It is the digit-recurrence algorithms elaborated here, due to their popularity and VLSI suitability.

In digit-recurrence algorithms the quotient is computed progressively (digit-by-digit), via a selection function, based on some selection rules and conditions. This quotient digit selection (QDS) is one of the key issues on which updating the partial remainder, regarding the divisor, is based. Therefore, the representations of the quotient, the divisor and the partial remainder are of paramount importance in determining the complexity of the division algorithm.

[4] Published @ Journal of Signal Processing Systems

There are miscellaneous decimal digit-recurrence division algorithms all use symmetric signed-digit representation of the quotient so as to simplify QDS implementation (e.g., [37, 44, 45, 46]). However, symmetric signed-digit representation of the quotient shows some drawbacks (is not optimal) in case of having not a symmetric error in QDS (e.g., in selection by truncation technique [47]). This justifies the need for a general investigation on the advantages and disadvantages of using diverse redundant quotient digit-sets. Moreover, not enough attention has been paid, in the literature, to the representations of the divisor and partial remainders and their significance in the division complexity, particularly QDS. Table 7.1 provides the notation used throughout this chapter.

Table 7.1: Notations and abbreviations

$0.1 \le X < 1$	Dividend	$[-\alpha_w, \beta_w]$	Partial remainder digit-set
$0.1 \le D < 1$	Divisor	ε_w	Error of truncated partial remainder
Q	Quotient	$[-\alpha_D, \beta_D]$	Divisor digit-set
R	Remainder	ε_D	Error of truncated divisor
QDS	Quotient digit selection	$H_n D$	Lower bound of partial remainder
GSD	Generalized signed-digit	$H_p D$	Upper bound of partial remainder
WBP	Weighted binary position	$\hat{*}$	$(*)$ truncated into *t* fractional WBP
ulp	Unit in the least significant position	t	Minimum # of fractional WBP of the partial remainder required in QDS
n	# of fractional quotient digits	τ	Minimum # of fractional digits of the partial remainder required in QDS
q_i	Quotient digit	$\overline{P_\tau^w}, \underline{P_\tau^w}$	Maximum, minimum truncation error of partial remainder in the digit of weight $r^{-\tau}$
$Q[i]$	Quotient in i^{th} iteration	$\overline{p_j^w}, \underline{p_j^w}$	Maximum, minimum value of binary position *j* of a partial remainder digit
$[-\alpha_q, \beta_q]$	Quotient digit-set	d	Minimum # of fractional WBP of the divisor required in QDS
$\varepsilon_q[i]$	Quotient error in i^{th} iteration	δ	Minimum # of fractional digits of the divisor required in QDS
h_n	Negative redundancy factor	$\overline{P_\delta^D}, \underline{P_\delta^D}$	Maximum, minimum truncation error of the divisor in the digit of weight $r^{-\delta}$
h_p	Positive redundancy factor	$\overline{p_j^D}, \underline{p_j^D}$	Maximum, minimum value of binary position *j* of a divisor digit
$w[i]$	Partial remainder in i^{th} iteration	(L_k, U_k)	The range of k^{th} selection interval
M_k	Comparison multiple	Δ_k	Overlap region between k^{th} and $(k-1)^{\text{th}}$ intervals
●	Dot notation for a bit with positive weight	○	Dot notation for a bit with negative weight

7.1 Decimal Digit-Recurrence Division Algorithm

Decimal digit-recurrence division algorithms estimate the final quotient, with the error less than $ulp = 10^{-n}$ (where n is the number of fractional quotient digits). In these algorithms, one quotient digit q_i $(0 \leq i \leq n)$ is generated per iteration, such that the quotient in the i^{th} iteration $Q[i]$ is assumed to be as in Eqn. 7.2.

$$Q[i] = \sum_{j=0}^{i} q_j 10^{-j} \tag{7.2}$$

Therefore, the error of the quotient estimation in the i^{th} iteration $\varepsilon_q[i]$ is defined as in Eqn. 7.3.

$$\varepsilon_q[i] = \frac{X}{D} - Q[i] \tag{7.3}$$

The final error $\varepsilon_q[n] = \frac{X}{D} - Q[n] < 10^{-n} = ulp$ imposes Eqn. 7.5 (based on Eqn. 7.4) as the required condition over the quotient estimation error, assuming a quotient digit with a redundant GSD representation i.e., $q_i \in [-\alpha_q, \beta_q]$ ($\alpha_q \geq 0, \beta_q \geq 0$ and $\alpha_q + \beta_q + 1 > 10$).

$$\begin{aligned} \varepsilon_q[i] &< \sum_{j=i+1}^{n} \left(\beta_q \times 10^{-j}\right) + ulp = \beta_q \times \left(\frac{10^{-i} - 10^{-n}}{9}\right) + ulp \\ \varepsilon_q[i] &> \sum_{j=i+1}^{n} \left(-\alpha_q \times 10^{-j}\right) - ulp = -\alpha_q \times \left(\frac{10^{-i} - 10^{-n}}{9}\right) - ulp \end{aligned} \tag{7.4}$$

$$10^{-i}\left(\frac{-\alpha_q}{9}\right) - 10^{-n}\left(1 + \frac{-\alpha_q}{9}\right) < \varepsilon_q[i] < 10^{-i}\left(\frac{\beta_q}{9}\right) + 10^{-n}\left(1 - \frac{\beta_q}{9}\right) \tag{7.5}$$

The i^{th} partial remainder $w[i]$ is defined as in Eqn. 7.6 by replacing the modified Eqn. 7.3 (i.e., multiplied by $10^i \times D$) into Eqn. 7.5, where $h_p = \frac{\beta_q}{9}$ and $h_n = \frac{-\alpha_q}{9}$ are positive and negative redundancy factors, respectively.

$$\begin{aligned} H_n D = D \times \left[h_n - 10^{i-n}(1 + h_n)\right] < w[i] &= 10^i (X - Q[i] \times D) \\ &< \left[h_p + 10^{i-n}\left(1 - h_p\right)\right] \times D = H_p D \end{aligned} \tag{7.6}$$

The admissible range for the partial remainder, defined in Eqn. 7.6, is also known as the convergence condition of the decimal digit-recurrence division algorithm. It should be noted that for digit-sets with $|h_p|, |h_n| < 1$ the range of the partial remainder can be deemed as simple as $h_n D \leq w[i] \leq h_p D$. However, in case of using maximally-redundant or over-redundant quotient digit-sets [48] (i.e., $|h_p|, |h_n| \geq 1$) the range should be exactly as is in Eqn. 7.6.

With the intention of having Eqn. 7.6 independent of *i*, one may use a range formulated with the tightest bound of all iterations (i.e., for $i = 0$). However, this simplification leads to an avoidable conservatism which imposes ranges tighter than what is strictly necessary for convergence (for $i > 0$). This point has been highlighted and elaborated on in [49] with an application example in [50], although not presenting a closed form formula.

To cap it all, it is suggested to apply the exact general range in Eqn. 7.6 so as to allow for more efficient algorithms to find grounds. Moreover, in case of using over-redundant quotient digit-sets (i.e., $|h_p|, |h_n| > 1$) a careful examination of the universal convergence condition (Eqn. 7.6) is in line, especially for large values of *i*. The recurrence equation of the decimal division algorithm is determined by substituting Eqn. 7.2 in Eqn. 7.6 for $w[i+1]$. This is shown in Eqn. 7.7.

$$\begin{aligned} w[i+1] &= 10^{i+1}(X - Q[i+1] \times D) \\ &= 10^{i+1}\left(X - Q[i] \times D - q_{i+1} \times 10^{-(i+1)} \times D\right) \\ &= 10^{i+1}(X - Q[i] \times D) - q_{i+1} \times D = 10w[i] - q_{i+1} \times D \\ \Longrightarrow w[i+1] &= 10w[i] - q_{i+1}D \end{aligned} \tag{7.7}$$

This recurrence equation should be performed such that Eqn. 7.6 satisfies for $w[i+1]$. This imposes an appropriate selection of the quotient digit q_{i+1}, regarding the values of $10w[i]$ and D. The redundant quotient digit-set allows for an imprecision in QDS such that the selection

function can be performed by manipulating the truncated version of $10w[i]$ and D; hence simpler QDS. This is going to be discussed in next section.

7.2 Quotient Digit Selection (QDS)

As mentioned in previous section QDS is one of the most important issues of the division hardware. This unit determines the next quotient digit q_{i+1} based on the value of the shifted partial remainder $10w[i]$ and the divisor D. With the intention of simplifying QDS, it is desirable to perform all the computations with truncated operands (i.e., $\widehat{10w[i]}$ and $\widehat{D}$) such that q_{i+1} is selected based on the divisor and the shifted partial remainder truncated into *d* and *t* fractional weighted-binary-positions (WBP) [51], respectively. The correct selection of quotient digit is guaranteed if computing Eqn. 7.7 (considering the truncation error) keeps the next partial remainder in the range shown by Eqn. 7.6.

Containment i.e., bounded partial remainders as in Eqn. 7.6; and continuity i.e., for any value of the shifted partial remainder there exist at least one quotient digit; are the main fundamental conditions to be satisfied by the selection function [17]. Regarding the containment condition, selection intervals (L_k, U_k) are defined in which it is possible to choose $q_{i+1} = k \in [-\alpha_q, \beta_q]$ if $10w[i] \in (L_k, U_k)$, while keeping the next partial remainder in range. Therefore, according to Eqn. 7.6, boundaries of selection intervals are as shown in Eqn. 7.8 and selecting $q_{i+1} = k$ is correct if $(H_n + k)D < 10w[i] < (H_p + k)D$.

$$q_{i+1} = k \Longrightarrow \begin{cases} L_k < 10w[i] < U_k \\ H_n D < 10w[i] - kD < H_p D \end{cases} \Longrightarrow \begin{cases} U_k = (H_p + k)D \\ L_k = (H_n + k)D \end{cases} \tag{7.8}$$

Regarding the continuity condition, selection intervals should overlap i.e., $L_k < U_{k-1}$; hence Eqn. 7.9.

$$L_k < U_{k-1} \Longrightarrow H_n + k < H_p + k - 1 \Longrightarrow H_n < H_p - 1 \qquad (7.9)$$

The Robertson's diagram (Fig. 7.1) illustrates the selection rules where the amount of the overlap Δ_k between two consecutive selection intervals is computed as in Eqn. 7.10.

$$\Delta_k = U_{k-1} - L_k = \big(H_p + k - 1\big)D - (H_n + k)D = \big(H_p - H_n - 1\big)D \qquad (7.10)$$

According to Eqn. 7.10 (and recalling Eqn. 7.6) the amount of the overlap region is determined by the quotient digit-set and the value of the divisor.

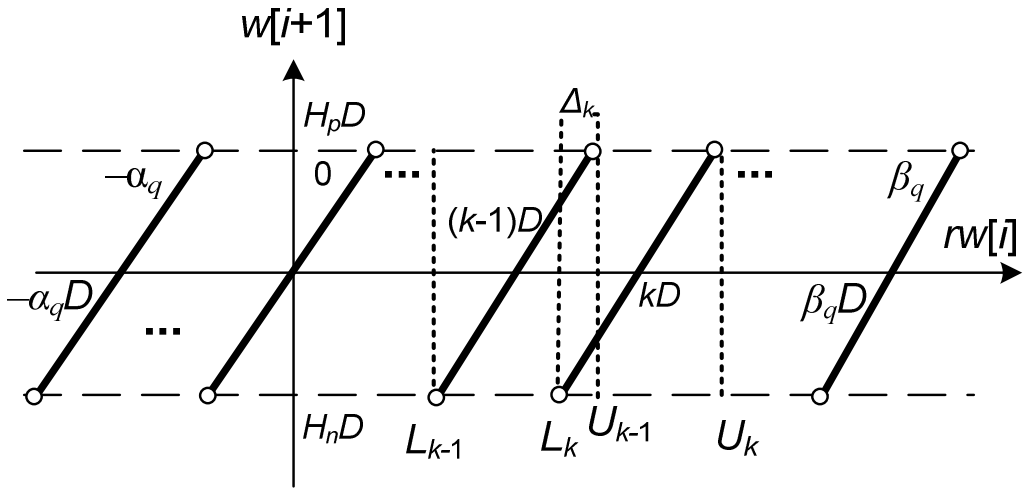


Figure 7.1: The value of Δ_k shown in Robertson's Diagram

In case of using truncated version of the divisor the corresponding error ε_D, according to Eqn. 7.10, involves the overlap region as $\widehat{\Delta_k} = \Delta_k - \varepsilon_D$. Moreover, the truncation error of the partial remainder defined as $\varepsilon_w = 10w[i] - 1\widehat{0w[i]}$ should be taken into account. Theorem 7.1 determines the maximum admissible value of these errors.

Theorem 7.1: The universal convergence condition (Eqn. 7.6) holds for truncated partial remainder and divisor if and only if $\{\forall k \in [-\alpha_q, \beta_q];\ |\varepsilon_w + \varepsilon_D| < \Delta_k\}$.

Proof: The most critical case of QDS occurs for which $10w[i] = U_{k-1}$, where the correct quotient digit is $q_{i+1} = k$. In case of using truncated partial remainders, QDS works fine if Eqn. 7.11 holds (see Fig. 7.1).

$$\widehat{10w[i]} = 10w[i] - \varepsilon_w > L_k \Longrightarrow U_{k-1} - \varepsilon_w > L_k \Longrightarrow \varepsilon_w < U_{k-1} - L_k = \Delta_k \qquad (7.11)$$

Considering the truncation error of the divisor, it must be $\varepsilon_w < \widehat{\Delta_k} = \Delta_k - \varepsilon_D$. The same approach can be done for negative partial remainders and hence $|\varepsilon_w + \varepsilon_D| < \Delta_k$. ■

The next quotient digit is selected by determining the interval (considering the truncated divisor) in which the truncated shifted partial remainder is. The impact of partial remainder's digit-set and that of the divisor, on QDS is discussed in next section.

7.3 Representation of the Divisor and Partial Remainders

It is shown in previous section that in case of using truncated divisor and partial remainders the correctness of QDS is not guaranteed unless $|\varepsilon_w + \varepsilon_D| < \Delta_k$, where ε_w and ε_D are the truncation errors and Δ_k indicates the amount of the overlap region between intervals. The amount of the truncation errors directly depends on the representation of the divisor and the partial remainder. Divisors usually take non-redundant representations so as to reduce the complexity of QDS while partial remainders are represented in redundant form so as to have carry-free computations and hence faster QDS [17].

Given that no advantage is recognized for using a redundant divisor, from now on, in this chapter, it is assumed that the divisor with a non-redundant representation with digit-set $[-\alpha_D, \beta_D]$ ($\alpha_D \geq 0, \beta_D \geq 0$ and $\alpha_D + \beta_D + 1 = 10$) and a redundant GSD representation for the partial remainder with digit-set $[-\alpha_w, \beta_w]$ ($\alpha_w \geq 0, \beta_w \geq 0$ and $\alpha_w + \beta_w + 1 > 10$). Consequently, the error of truncating the shifted partial remainder into *t* fractional WBP is as in Eqn. 7.12a, where $\overline{p_j^w}$ and $\underline{p_j^w}$ symbolize the maximum and minimum value of the binary position *j* of each digit of the partial remainder, respectively, and $\tau = \left\lceil \frac{t}{\log_2 10} \right\rceil$.

$$\varepsilon_w < \overline{P_\tau^w}10^{-\tau} + \sum_{i=\tau+1}^{\infty} \beta_w 10^{-i} = \overline{P_\tau^w}10^{-\tau} + \beta_w \times \left(\frac{10^{-\tau}}{9}\right); \qquad \overline{P_\tau^w} = \sum_{j=0}^{\tau(\log_2 10)-t-1} \overline{p_j^w}2^{-j}$$

$$\varepsilon_w > \underline{P_\tau^w}10^{-\tau} + \sum_{i=\tau+1}^{\infty} -\alpha_w 10^{-i} = \underline{P_\tau^w}10^{-\tau} - \alpha_w \times \left(\frac{10^{-\tau}}{9}\right); \qquad \underline{P_\tau^w} = \sum_{j=0}^{\tau(\log_2 10)-t-1} \underline{p_j^w}2^{-j} \tag{7.12a}$$

The same can be considered for the divisor (Eqn. 7.12b), considering ε_D as the error of truncating the divisor into d fractional WBP, where $\overline{p_j^D}$ and $\underline{p_j^D}$ symbolize the maximum and minimum value of the binary position j of each divisor digit, respectively, and $\delta = \left\lceil \frac{d}{\log_2 10} \right\rceil$.

$$\varepsilon_D < \overline{P_\delta^D}10^{-\delta} + \sum_{i=\delta+1}^{\infty} \beta_D 10^{-i} = \overline{P_\delta^D}10^{-\delta} + \beta_D \times \left(\frac{10^{-\delta}}{9}\right); \qquad \overline{P_\delta^D} = \sum_{j=0}^{\delta(\log_2 10)-d} \overline{p_j^D}2^{-j}$$

$$\varepsilon_D > \underline{P_\delta^D}10^{-\delta} + \sum_{i=\delta+1}^{\infty} -\alpha_D 10^{-i} = \underline{P_\delta^D}10^{-\delta} - \alpha_D \times \left(\frac{10^{-\delta}}{9}\right); \qquad \underline{P_\delta^D} = \sum_{j=0}^{\delta(\log_2 10)-d} \underline{p_j^D}2^{-j} \tag{7.12b}$$

Applying Eqn. 7.12 (for ε_w and ε_D) into $|\varepsilon_w + \varepsilon_D| < \Delta_k$ leads to Eqn. 7.13, as the required condition for the correctness of the QDS operation with truncated divisor and partial remainder.

$$\begin{gathered} \left(\beta_w \times \left(\frac{10^{-\tau}}{9}\right) + \overline{P_\tau}10^{-\tau} + \beta_D \times \left(\frac{10^{-\delta}}{9}\right) + \overline{P_\delta^D}10^{-\delta} < \Delta_k\right) \\ and \\ \left(\alpha_w \times \left(\frac{10^{-\tau}}{9}\right) - \underline{P_\tau}10^{-\tau} + \alpha_D \times \left(\frac{10^{-\delta}}{9}\right) - \underline{P_\delta^D}10^{-\delta} < \Delta_k\right) \end{gathered} \tag{7.13}$$

Various methods have been introduced in the literature to find the minimum required values of t and d (consequently τ and δ) [52, 53, 54]; which affect the QDS complexity. In addition to the latter parameters, it is inferred from Eqn. 7.13 that the digit-sets of the partial remainder, the divisor and the quotient (substituting Δ_k from Eqns. 7.10 and 7.6) have great impacts on the complexity of QDS, as are discussed below.

a) **Quotient digit-set $[-\boldsymbol{\alpha_q}, \boldsymbol{\beta_q}]$:** According to Eqns. 7.6 and 7.10, quotient digit-set determines $\Delta_k = \left[\left(\frac{\beta_q+\alpha_q}{9}\right) + 10^{i-n}\left(2 - \frac{\beta_q+\alpha_q}{9}\right) - 1\right] D$. Therefore, the higher the cardinality of this digit-set (i.e., $\alpha_q + \beta_q + 1$), which influences its redundancy, the looser the condition of Eqn. 7.13. This allows for more imprecision in QDS computation; hence simpler QDS. However, this simplification linearly increases the complexity of the generation of the divisor multiples $q_{i+1}D$, required for partial remainder computation (Eqn. 7.7) [17].

b) **The digit-set of the divisor $[-\boldsymbol{\alpha_D}, \boldsymbol{\beta_D}]$ and partial remainder $[-\boldsymbol{\alpha_w}, \boldsymbol{\beta_w}]$:** According to Eqn. 7.13 $\max[(\alpha_w + \alpha_D), (\beta_w + \beta_D)]$ has an impact on the complexity of QDS. The lower the value of the latter, the smaller τ and δ (consequently *t* and *d*) are required and hence simpler QDS. Therefore, it is important to keep $\max[(\alpha_w + \alpha_D), (\beta_w + \beta_D)]$ as low as possible.

According to issue b) above, there is a need to minimize $\max[(\alpha_w + \alpha_D), (\beta_w + \beta_D)]$. Recalling that the divisor usually takes a non-redundant representation while partial remainders are represented in redundant form, the followings are suggested.

- Minimally redundant symmetric signed-digit [55] representation of the partial remainder i.e., $\alpha_w = \beta_w = 5$ for radix-10. Yet in case of using non-redundant partial remainders it is suggested to employ $\alpha_w = 5\ (4)$ and $\beta_w = 4\ (5)$.
- Minimally asymmetric non-redundant signed-digit representation of the divisor i.e., $\alpha_D = 5\ (4)$ and $\beta_D = 4(5)$ for radix-10.

7.4 The Proposed Decimal Divider

This section is meant to represent an architecture for the decimal division algorithm using the proposed divisor's and partial remainders' redundant representations (particularly their digit-sets). For this purpose, one would assume the minimally redundant symmetric signed-digit representation of the partial remainders (digit-set is $[-5,5]$). However, partial remainders are generated in digit-set of $[-6,5]$ to be able to utilize a carry-free addition in partial remainder computation (PRC) and use minimally asymmetric non-redundant signed-digit representation of the divisor (digit-set is $[-4,5]$). it is also assumed that the minimally redundant signed-digit quotient digit-set i.e., $q_{i+1} \in [-5,5]$. This digit-set is used due to the fact that decimal dividers consume high area costs and there is no intention to add more cost by using a more redundant quotient. Therefore, the convergence condition, according to Eqn. 7.6, can be deemed as Eqn. 7.14.

$$|w[i]| \leq \frac{5}{9} \times D \tag{7.14}$$

Moreover, according to Eqn. 7.10, the amount of the overlap is $\Delta_k > \frac{D}{9}$. Therefore, as a result of Theorem 7.1, $|\varepsilon_w + \varepsilon_D| \leq \frac{D}{9}$ guarantees the correctness of the algorithm in case of using the truncated shifted partial remainder and divisor in QDS, with the error of ε_w and ε_d, respectively. Given the range of the normalized divisor as $0.1 \leq D < 1$, Eqn. 7.15 must hold in all cases.

$$|\varepsilon_w + \varepsilon_D| \leq \frac{1}{90} \tag{7.15}$$

The proposed architecture is heavily based on the one introduced in [46] where the partial remainder is decomposed into the binary and decimal parts. The binary part is represented in 2's

complement carry-save while the decimal part employs $[-6,5]$ as the digit-set. The digit encoding of the decimal parts of the partial remainder and the divisor are illustrated in Fig. 7.2.

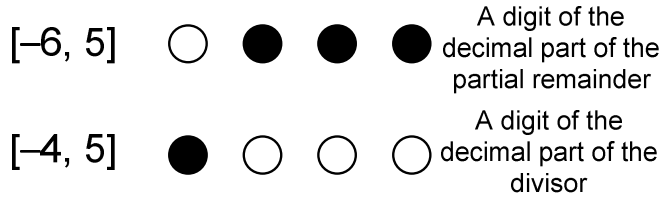


Figure 7.2: Digit Encodings for the Decimal Parts of the Partial Remainder and Divisor

The block diagram of the proposed architecture, adapted from [46], is shown in Fig. 7.3; where $M_k = (k - 0.5)D;\ \ k \in [-4, 5]$ are the comparison multiples. Recalling the admissible error range, described by Eqn. 7.15, *t* and *d* (the minimum number of truncated fractional WBPs required in QDS) can be determined based on the methods discussed in [52, 54] where the actual ranges of ε_w and ε_D are computed based on Eqn. 7.12 and applied into Eqn. 7.15. The computations impose $t = 8$ and $d = 8$ (i.e., two fractional digits) as to have Eqn. 7.15 satisfied. It should be noted that according to Fig. 7.3 it is required to have $100w[i]$ and $10q_{i+1}D$ in QDS and hence two integer and two fractional digits are involved in QDS. Therefore, the range of the binary part is determined by Eqn. 7.16. This range of values, given that $2^{12} < 5555 < 2^{13}$, requires at most 14 bits in binary two's complement.

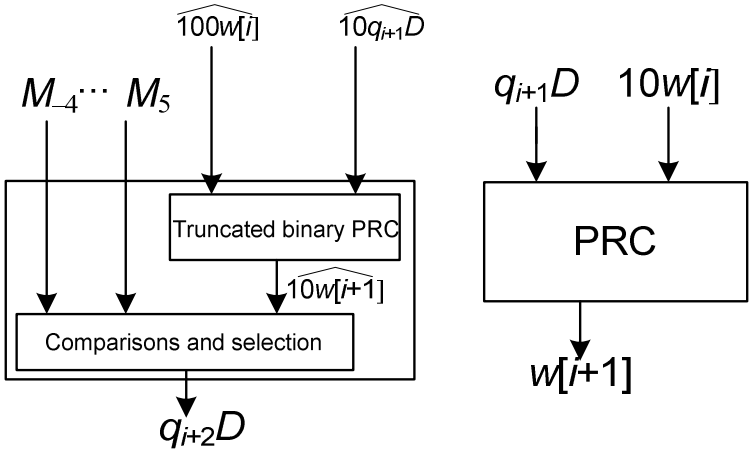


Figure 7.3: The Architecture including QDS and PRC

Besides, due to the comparison with $M_k = (k - 0.5)D$, one extra rounding bit (an input carry to QDS) is required to determine whether the third fractional digit (weigh 10^{-3}) is greater than or equal to 5.

$$\left|\widehat{10w}[i]\right| \leq 100\rho D = \frac{500}{9}D \leq \frac{500}{9} = 55.55 \tag{7.16}$$

Fig. 7.4, adapted from [46], depicts the abstract architecture of binary and decimal PRC and the QDS, where the symbols $D*$ and $w*$ (D and w), denoting binary (decimal) values.

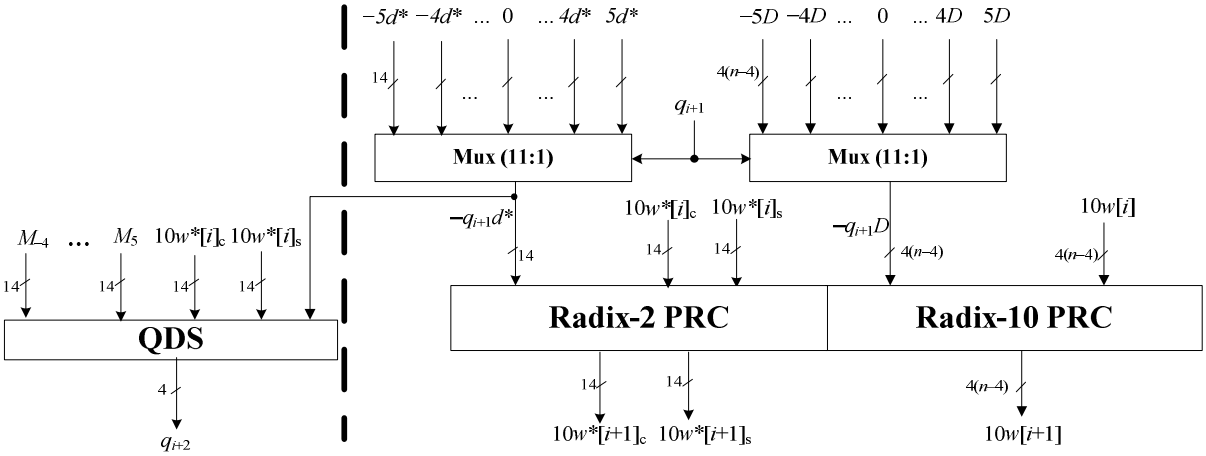


Figure 7.4: The Architecture including the Binary and Decimal QDS and PRC

The QDS and Radix-2 PRC are exactly the same as those in [46] but for 14-bit operands (instead of 17 bits). This reduction in the number of bits has a great impact on the latency of the QDS due to the carry-propagation involved in it.

The Radix-10 PRC is a simple redundant decimal adder whose details are illustrated in Fig. 7.5, where the Recoder block is a mere combinational logic with five input bits and six output bits. It should be noted that the Radix-10 PRC is not in the critical path and hence we strive for the minimum area of this block. The initialization phase of the proposed divider is the same as that of [46] except for the extra constant-time digit-set convertor which is responsible for the conversion from [0,9] to [−4,5] over the divisor multiples.

Given that the quotient representation in the proposed design is exactly the same as that of [46], the termination phase which is responsible for correction, conversion, normalization and rounding is equivalent in both designs.

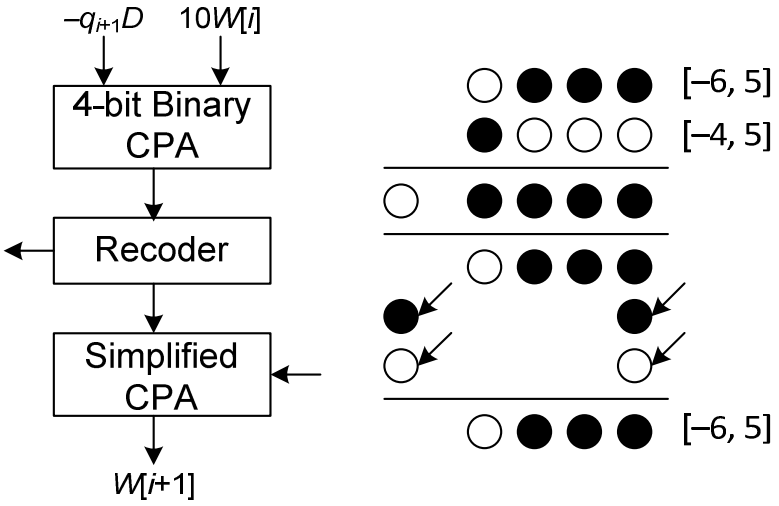


Figure 7.5: Radix-10 PRC (digit slice)

7.5 Evaluations and Comparison of Decimal Dividers

The overall latency and the critical path of the proposed decimal divider which determines the cycle time (i.e., the overall QDS delay) is compared to that of the fastest previous work [46] in Table 7.2 where the latency improvement of the proposed decimal divider is also illustrated. The evaluation data is reproduced from [46] where the delay of simple AND/OR gates with at most three inputs is denoted by ΔG and 1.5 ΔG is considered for an XOR gate.

According to Table 7.2 the proposed decimal divider with the suggested representations of the divisor and partial remainders shows 10% improvement in the latency. This enhancement is achieved at the cost of minimum area overhead due to the fact that most of the constituent parts of the proposed divider are the same as those in [46]. The price of the extra digit-set

convertor in the initialization phase of the proposed design can be considered remunerated by the area reduction in the QDS i.e., manipulating 14 bits instead of 17 bits.

For more accurate analysis the proposed divider is simulated by Synopsys Design Compiler using the STM $90nm$ CMOS standard library [26] for 1.00 VDD and 25°C temperature. This evaluation leads to $0.62ns$ as the cycle time and $56{,}468\mu m^2$ of area.

Table 7.2: Critical Path of Decimal Dividers (16 digits)

		Proposed	[46]
Critical Path	The (4:2) compressors	4.5 ΔG	4.5 ΔG
	Sign-Detection (Parallel Prefix Network)	9 ΔG (15-bit width)	11 ΔG (18-bit width)
	Enable Signal Generator	1.5 ΔG	1.5 ΔG
	Selector	4 ΔG	4 ΔG
Cycle Time (QDS Delay)		19 ΔG	21 ΔG
Ratio		0.90	1.00

The same simulation is done for the work in [46]. The outcomes show $0.68ns$ as the cycle time and $49{,}000\mu m^2$ of area. It should be noted that the latency and area of the registers are not included for simplicity. This does not make any changes in the comparison results given that both designs use the same size of registers. Table 7.3 compares the evaluation results of the proposed design with those of [46].

Table 7.3: Comparison based on the Synthesis Results

	Cycle Time (ns)	Ratio	Area (μm^2)	Ratio
Proposed	0.62	1.000	56,468	1.000
[46]	0.68	1.096	49,000	0.867

CHAPTER 8

DECIMAL SQUARE-ROOT[5]

Beside the popular four dyadic decimal operations (i.e., addition, subtraction, multiplication, and division), the unary square-root operation can be implemented as an instruction, directly in hardware. This boosts up the performance of the decimal floating-point unit in the processors, particularly when the square-root is implemented sharing hardware with decimal divider.

Decimal square-root units are usually implemented in hardware using functional algorithms such as Newton-Raphson [56, 57, 58]. However, these methods require a multiplication per iteration. Consequently, given the high cost of parallel decimal multipliers, the functional algorithms seem inadequate to be employed for decimal square-root. The digit-recurrence algorithms, conversely, are conceptually simple and well suited for decimal square-root due to their low hardware complexity. Moreover, using these algorithms paves the way for the shared decimal division/square root unit.

The digit-recurrence square-root algorithm is exemplified in Eqn. 8.1, where $0.01 \leq \mathrm{X} < 1$, $0.1 \leq \mathrm{Q} < 1$ and $0 \leq \mathrm{R} < ulp = 10^{-\mathrm{n}}$ (where n is the number of fractional digits) are the normalized radicand, root and the remainder, respectively. It should be noted that for the floating-point representation the radicand should be scaled in a way to have an even exponent.

$$\sqrt{X} = Q + R \tag{8.1}$$

[5] Published @ Journal of Circuits, Systems and Signal Processing

The two recent pertinent works based on the Newton-Raphson algorithm are [56, 58], where the latter is the fastest available one in the literature. The former presents a hardware design for decimal floating-point square root in which the size of the required look-up table is reduced with respect to other similar designs. The most recent works [57, 58] have reduced the latency of the decimal square-root operation by taking advantage of a parallel fused-multiply-add (FMA), but at the expense of high area consumption.

The work by Ercegovac and McIlhenny [59], which is based on the digit-recurrence algorithm, uses a look-up table to compute a rough approximation of the root and then correct the result via a division operation. Moreover, one may use the CORDIC algorithm to compute the decimal square root [60, 61].

A new digit-recurrence algorithm and the corresponding hardware architecture to compute the decimal square-root are discussed in this chapter. The main advantage of the proposed algorithm, over the previous works, is to remove the slow and costly look-up tables. This, however, entails generating inconstant comparison multiples to be used in the proposed SRT algorithm.

8.1 Decimal Digit-Recurrence Square-Root

Decimal digit-recurrence square-root estimates the root Q, with the error less than $ulp = 10^{-n}$. In this approach, one root digit q_i $(0 \leq i \leq n)$ is generated per iteration, such that the root in the i^{th} iteration $Q[i]$ is assumed to be as in Eqn. 8.2. It should be noted that $0.1 \leq Q = \sqrt{X} - R < 1$ forces $q_0 = 0$, in case of non-redundant representation of Q.

$$Q[i] = \sum_{j=0}^{i} q_j 10^{-j} \tag{8.2}$$

Therefore, the error of the root estimation in the i^{th} iteration $\varepsilon_q[i]$ is

$$\varepsilon_q[i] = \sqrt{X} - Q[i] \tag{8.3}$$

The bounds of error after n iteration $0 \le \left|\varepsilon_q[n]\right| = \sqrt{X} - Q[n] < 10^{-n} = ulp$ imposes (as a conclusion of Eqns. 8.2 and 8.3 for i+1) Eqn. 8.4 as the required condition over the root estimation error, assuming a root digit with a redundant representation i.e., $q_i \in [-\alpha, \beta]$ ($\alpha, \beta \geq 0$ and $\alpha + \beta + 1 > 10$).

$$10^{-i}\left(\frac{-\alpha}{9}\right) < \varepsilon_q[i] < 10^{-i}\left(\frac{\beta}{9}\right) \tag{8.4}$$

Defining bounds $H_n[i]$ ($H_p[i]$) as the lower (upper) bounds of the i^{th} partial remainder $w[i]$, it must satisfy Eqn. 8.5, derived by replacing Eqn. 8.3 into Eqn. 8.4.

$$\begin{gathered}
10^{-i}\frac{-\alpha}{9} < \sqrt{X} - Q[i] < 10^{-i}\frac{\beta}{9} \\
\Downarrow \\
Q[i]^2 + \left(10^{-i}\frac{-\alpha}{9}\right)^2 + 2Q[i]10^{-i}\frac{-\alpha}{9} < X < Q[i]^2 + \left(10^{-i}\frac{\beta}{9}\right)^2 + 2Q[i]10^{-i}\frac{\beta}{9} \\
\Downarrow \\
\left(10^{-i}\frac{-\alpha}{9}\right)^2 + 2Q[i]10^{-i}\frac{-\alpha}{9} < X - Q[i]^2 < \left(10^{-i}\frac{\beta}{9}\right)^2 + 2Q[i]10^{-i}\frac{\beta}{9} \\
\Downarrow \\
H_n[i] = 10^{-i}\left(\frac{-\alpha}{9}\right)^2 + \left[2 \times \left(\frac{-\alpha}{9}\right) \times Q[i]\right] < w[i] = 10^i(X - Q[i]^2) \\
< 10^{-i}\left(\frac{\beta}{9}\right)^2 + \left[2 \times \left(\frac{\beta}{9}\right) \times Q[i]\right] = H_p[i]
\end{gathered} \tag{8.5}$$

The admissible range for the partial remainder, defined in Eqn. 8.5, is also known as the convergence condition of the decimal digit-recurrence square-root algorithm.

The recurrence equation of the decimal square-root algorithm is determined, in Eqn. 8.6, by substituting Eqn. 8.2 in Eqn. 8.5 for $w[i+1]$.

$$\begin{aligned}
w[i+1] &= 10^{i+1}(X - Q[i+1]^2) \\
&= 10^{i+1}\left(X - Q[i]^2 - (q_{i+1})^2 \times 10^{-2(i+1)} - 2Q[i] \times q_{i+1} \times 10^{-(i+1)}\right) \\
&= 10w[i] - 2Q[i] \times q_{i+1} - (q_{i+1})^2 \times 10^{-(i+1)} \\
\Longrightarrow w[i+1] &= 10w[i] - \mathbb{Q}[i]; \qquad \mathbb{Q}[i] = Q[i] \times 2q_{i+1} + (q_{i+1})^2 \times 10^{-(i+1)}
\end{aligned} \tag{8.6}$$

The recurrence equation should be performed such that $w[i+1]$ be bounded as in Eqn. 8.5. This imposes a careful computation for $\mathbb{Q}[i]$ and hence a suitable selection of the q_{i+1}, regarding the values of $10w[i]$ and $Q[i]$ i.e, Root Digit Selection (RDS). The correct selection is guaranteed if computing Eqn. 8.6 satisfies Eqn. 8.5 for the next partial remainder. For this purpose, selection intervals $(L_k[i], U_k[i])$ are defined such that if $10w[i] \in (L_k[i], U_k[i])$ with $k \in [-\alpha, \beta]$ then $q_{i+1} = k$ is admissible i.e., keeping the next partial remainder within the required bound. Therefore, according to Eqns. 8.5 and 8.6, boundaries of the selection intervals ($U_k[i]$ and $L_k[i]$) are

$$\begin{aligned} q_{i+1} = k \Longrightarrow & \begin{cases} L_k[i] < 10w[i] < U_k[i] \\ H_n[i+1] < 10w[i] - \mathbb{Q}[i] < H_p[i+1] \end{cases} \\ \Longrightarrow & \begin{cases} U_k[i] = H_p[i+1] + Q[i] \times 2k + k^2 \times 10^{-(i+1)} \\ L_k[i] = H_n[i+1] + Q[i] \times 2k + k^2 \times 10^{-(i+1)} \end{cases} \end{aligned} \tag{8.7}$$

It is necessary that for any value of the shifted partial remainder $10w[i]$ there exist at least one root digit. Therefore, selection intervals must overlap i.e., $L_k[i] < U_{k-1}[i]$; hence Eqn. 8.8 must hold in all iterations i.e., for $0 \leq i \leq n$.

$$\begin{gathered} L_k[i] < U_{k-1}[i] \\ \Downarrow \\ H_n[i+1] + Q[i] \times 2k + k^2 \times 10^{-(i+1)} < H_p[i+1] + Q[i] \times 2(k-1) + (k-1)^2 \times 10^{-(i+1)} \\ \Downarrow \\ 2Q[i] + (2k-1) \times 10^{-(i+1)} < H_p[i+1] - H_n[i+1] \end{gathered} \tag{8.8}$$

To make RDS less costly it is common to select q_{i+1} via comparing the truncated shifted partial remainder $(10w[i])'$ with comparison multiples $M_k[i]$ for $k \in (-\alpha, \beta]$. These comparison multiples should be bounded as in Eqn. 8.9 (also shown in Fig. 8.1) in order to be able to compensate for the error caused by using truncated operands. Therefore, the next quotient digit is

selected based on Eqn. 8.10 and hence the maximum absolute admissible selection error is equal to $min[(M_k[i] - L_k[i]), (U_k[i] - M_k[i])]$.

$$\begin{gathered} L_k[i] < M_k[i] < U_{k-1}[i] \\ \Downarrow \\ H_n[i+1] + Q[i] \times 2k + k^2 \times 10^{-(i+1)} < M_k[i] \end{gathered} \tag{8.9}$$

$$< H_p[i+1] + Q[i] \times 2(k-1) + (k-1)^2 \times 10^{-(i+1)}$$

$$q_{i+1} = k\,, \;\; if \;\; M_k[i] \;\leq (10w[i])' < M_{k+1}[i] \tag{8.10}$$

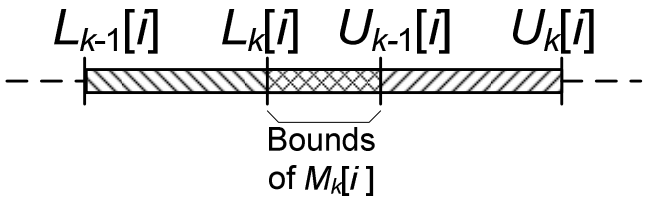


Figure 8.1: The Selection Intervals and the Comparison Multiples

8.2 The Proposed Decimal Square-Root Unit

It is $\alpha = \beta = 5$ for the proposed square-root algorithm for two reasons; First, to reduce the complexity of computing $(q_{i+1})^2$, required in Eqn. 8.6; second, for fewer comparison multiples. Consequently, the bounds of the partial remainder (according to Eqn. 8.5) are as Eqn. 8.11. These bounds, according to Eqn. 8.8, require $Q[i] > 0.05$ which always hold given the assumed radicand and root i.e., $0.1 \leq Q < 1$.

$$H_n[i] = \frac{25}{81} \times 10^{-i} - \frac{10}{9} Q[i] < w[i] < \frac{25}{81} \times 10^{-i} + \frac{10}{9} Q[i] = H_p[i] \tag{8.11}$$

The comparison multiples should be bounded as Eqn. 8.12, derived from Eqn. 8.9 by replacing $H_p[i+1]$ from Eqn. 11 and $Q[i+1] = Q[i] + q_{i+1} \times 10^{-(i+1)}$.

$$\left(2k - \frac{10}{9}\right) Q[i] + 10^{-i}\left[\frac{k^2}{10} - \frac{k}{9} + \frac{25}{810}\right] < M_k[i] < \left(2k - \frac{8}{9}\right) Q[i] + 10^{-i}\left[\frac{(k-1)^2}{10} + \frac{k}{9} + \frac{25}{810}\right] \tag{8.12}$$

Therefore, the comparison multiples $M_k[i]$ are

$$M_k[i] = \frac{L_k[i]+U_{k-1}[i]}{2} \Longrightarrow M_k[i] = (2k-1)Q[i] + 10^{-i}\left[\frac{k^2+(k-1)^2}{20} + \frac{25}{810}\right] \quad (8.13)$$

In the RDS the truncated comparison multiples $(M_k[i])'$ are subtracted from the truncated shifted partial remainder $(10w[i])'$, where the maximum admissible error, for all values of *k,* is defined as $|\varepsilon[i]| < min[(M_k[i] - L_k[i]), (U_k[i] - M_k[i])]$.

Given that $M_k[i] - L_k[i] = U_k[i] - M_k[i] = \frac{Q[i]}{9} + 10^{-i}\left(\frac{2k+9}{180}\right)$, $|\varepsilon[i]| < \frac{Q[i]}{9} + 10^{-i}\left(\frac{2k+9}{180}\right)$.

For $i = 0$

$$if\ Q[0] = 0\ then\ 1 \le q_1 = k \le 5 \Longrightarrow |\varepsilon[0]| < \frac{11}{180} = 0.0611\ldots$$
$$if\ Q[0] = 1\ then\ -5 \le q_1 = k \le 0 \Longrightarrow |\varepsilon[0]| < \frac{1}{9} - \frac{1}{180} = 0.1055\ldots$$

For $i \ge 1$, regarding $0.1 \le Q[i] < 1$, the admissible error of the RDS ($\varepsilon[i]$) is bounded as

$$|\varepsilon[i]| < \frac{1}{90} - \left(\frac{10^{-i}}{180}\right) \xrightarrow{i\ge 1} |\varepsilon[i]| < 0.01055\ldots \quad (8.14)$$

The initial values of the square root recurrence (Eqn. 6) are determined, given the minimally redundant root digit-set, as $Q[0] = q_0 = \begin{cases} 0 & if \quad 0.01 \le X < 0.3 \\ 1 & if \quad 0.3 \le X < 1 \end{cases}$ and $w[0] = X - Q[0]$. The upper bound of *X* with $q_0 = 0$ is $(0.55\ldots)^2 = \left(\frac{5}{9}\right)^2 = \frac{25}{81} \approx 0.3$.

Initialization:

if $X < 0.3$ ***then*** $Q[0] = q_0 = 0$ ***else*** $Q[0] = q_0 = 1$; $\quad w[0] = X - Q[0]$

Recurrence:

For $0 \le i \le n$ do

1) $M_k[i] = (2k-1)Q[i] + 10^{-i}\left[\frac{k^2+(k-1)^2}{20} + \frac{25}{810}\right]$ for $-4 \le k \le 5$.
2) $q_{i+1} = RDS[(10w[i])', (M_k[i])'] \Longrightarrow q_{i+1} = k \ \ if \ \ (M_k[i])' \le (10w[i])' < (M_{k+1}[i])'$.
3) $\mathbb{Q}[i] = Q[i] \times 2q_{i+1} + (q_{i+1})^2 \times 10^{-(i+1)}$; $\quad Q[i+1] = Q[i] + q_{i+1} \times 10^{-(i+1)}$.
4) $w[i+1] = 10w[i] - \mathbb{Q}[i]$.

Termination:

Perform the rounding and normalization and conversion of $Q[n+1]$ to BCD format.

Figure 8.2: The proposed decimal square root algorithm

The proposed decimal square root algorithm is summarized in Fig. 8.2. Example 8.1 presents the required steps, based on Fig. 8.2, to compute a decimal square-root.

Example 8.1: Decimal Square-Root

Initialization:

$$\boldsymbol{X = 0.3521986} \Rightarrow\ Q[0] = q_0 = 1 \quad and \quad w[0] = 0.3521986 - 1 = -0.6478014$$

Recurrence:

$\boldsymbol{i = 0}$: $-7.00 = M_{-4}[0] < (10w[0])' = -6.47 < M_{-3}[0] = -6 \Rightarrow w[1] = -0.078014;\ \boldsymbol{q_1 = -4}$.

$\boldsymbol{i = 1}$: $-1.80 = M_{-1}[1] < (10w[1])' = -0.78 < M_0[1] = -0.6 \Rightarrow w[2] = 0.40986;\ \boldsymbol{q_2 = -1}$.

$\boldsymbol{i = 2}$: $+2.95 = M_3[2] < (10w[2])' = 4.09 < M_4[2] = 4.14 \Rightarrow w[3] = 0.5496;\ \boldsymbol{q_3 = 3}$.

$\boldsymbol{i = 3}$: $-5.33 = M_5[3] < (10w[3])' = 5.49 \Rightarrow w[4] = -0.4365;\ \boldsymbol{q_4 = 5}$.

$\boldsymbol{i = 4}$: $-5.34 = M_{-4}[4] < (10w[4])' = -4.36 < M_{-3}[4] = -4.15 \Rightarrow w[5] = 0.38284;\ \boldsymbol{q_5 = -4}$.

$\boldsymbol{i = 5}$: $+2.96 = M_3[5] < (10w[5])' = 3.8284 < M_4[5] = 4.15 \Rightarrow w[6] = 0.267631;\ \boldsymbol{q_6 = 3}$.

$\boldsymbol{i = 6}$: $+1.78 = M_2[6] < (10w[6])' = 2.67 < M_3[6] = 2.96 \Rightarrow w[7] = 0.302458;\ \boldsymbol{q_7 = 2}$.

$\boldsymbol{i = 7}$: $+2.96 = M_3[7] < (10w[7])' = 3.02 < M_4[7] = 4.15 \Rightarrow w[8] = -0.536203;\ \boldsymbol{q_8 = 3}$.

Termination:

$$Q[8] = 1.\bar{4}\bar{1}35\bar{4}323 \xrightarrow{Conversion} Q[8] = 0.59346323 \xrightarrow{Rounding} \boldsymbol{Q = 0.5934632}$$

8.2.1 Proposed Architecture

The most straight-forward architecture to implement the recurrence stage of Fig. 8.2, is shown in Fig. 8.3, where $K = \frac{k^2+(k-1)^2}{20} + \frac{25}{810}$. For a faster design, Fig. 8.3 can be modified as shown in Fig. 8.4 such that the next root digit is generated partially in parallel with the partial remainder computation. The details of each constituent block in Fig. 8.4 are presented in the following.

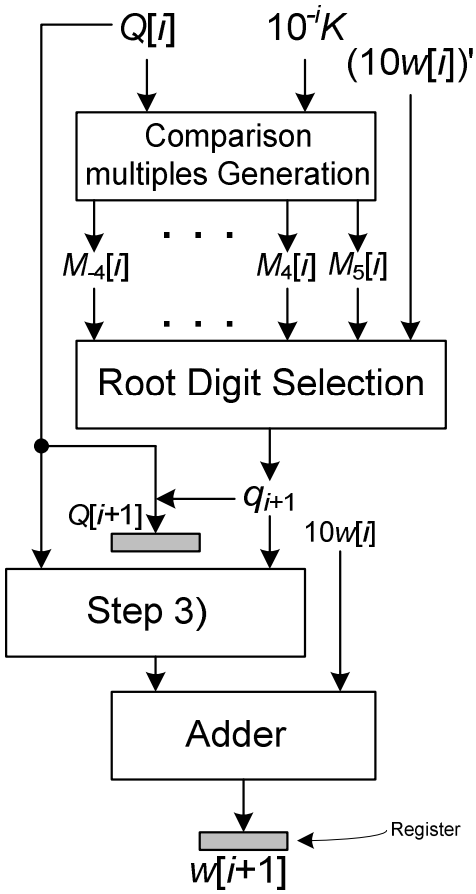


Figure 8.3: The Straight-Forward Architecture

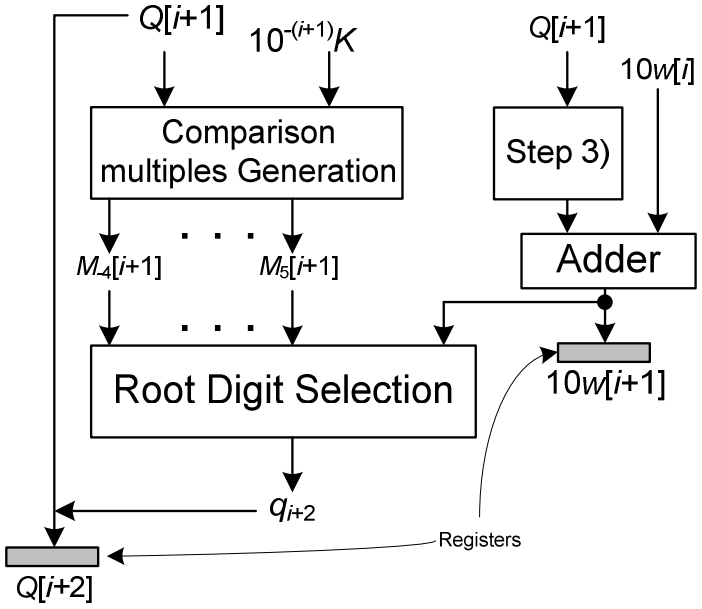


Figure 8.4: Block Diagram of the Proposed Architecture

Step 3) in Fig. 8.4, according to Fig. 8.2, is meant to compute $\mathbb{Q}[i] = Q[i] \times 2q_{i+1} + (q_{i+1})^2 \times 10^{-(i+1)}$, which is performed in three parts.

- **Part I:** Compute $Q[i] \times 2q_{i+1}$, primarily, as two minimally redundant decimal numbers (i.e., $U_Q + V_Q = Q[i] \times 2q_{i+1}$). For this purpose, the required easy-multiples of $Q[i]$ (i.e., $\pm 2Q[i], \pm 4Q[i]\ and\ \pm 10Q[i]$) are generated, next q_{i+1} selects the appropriate multiples to be assigned as U_Q and V_Q.
- **Part II:** Compute $(q_{i+1})^2$ via a simple combinational logic. Given the minimally redundant digit-set $[-5,5]$, we have $0 \leq 10C + S = (q_{i+1})^2 \leq 25$; hence $0 \leq C \leq 2$ and $-5 \leq S \leq 5$.
- **Part III:** Compute $U_Q + V_Q + C$ via a minimally redundant decimal adder [31, 62] where C fits into the adder as the low-significant bits of V_q and U_q, due to their even value.

Fig. 8.5 shows how the aforementioned parts are connected together to generate $\mathbb{Q}[i]$. The decimal redundant adder shown in Fig. 8.4, to generate the partial remainder, receives two inputs and generates an output all in $[-6,6]$ digit-sets. The details of this (and other) redundant adders are extensively discussed in [31, 30, 62]. The *comparison multiples generation* block, according to Fig. 8.2, is responsible to generate

$$M_k[i+1] = (2k-1)Q[i+1] + 10^{-(i+1)}\mathrm{K};\ -4 \leq k \leq 5;\ \text{where K} = \left\lceil \frac{k^2+(k-1)^2}{20} + \frac{25}{810} \right\rceil \quad (8.15)$$

For this purpose, first $(2k-1)Q[i+1]$ is generated by means of easy-multiples of $Q[i+1]$. The required multiples are $\pm Q[i+1], \pm 2Q[i+1], \pm 3Q[i+1]$ and $\pm 10Q[i+1]$ to generate 10 interim sums $W_k[i+1]$ as follows, where the addition is performed via a redundant adder whose inputs are in $[-5,5]$ and $[-6,6]$ and the output is $[-6,6]$. In essence, $\pm Q[i+$

$1], \pm 2Q[i+1]$ and $\pm 10Q[i+1]$ are generated in $[-5,5]$ digit set while $\pm 3Q[i+1]$ is in $[-6,6]$.

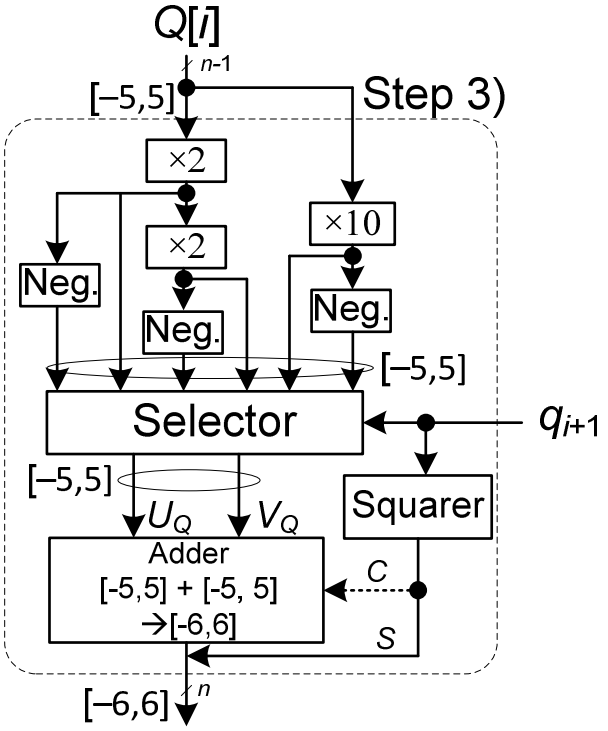


Figure 8.5: Details of Step 3) in Fig. 8.4

$$W_{-4}[i+1] = -10Q[i+1] + Q[i+1]; \qquad W_{-3}[i+1] = -10Q[i+1] + 3Q[i+1]$$
$$W_{-2}[i+1] = -3Q[i+1] - 2Q[i+1]; \qquad W_{-1}[i+1] = -3Q[i+1]$$
$$W_0[i+1] = -Q[i+1]; \qquad W_1[i+1] = Q[i+1] \qquad (8.16)$$
$$W_2[i+1] = 3Q[i+1]; \qquad W_3[i+1] = 2Q[i+1] + 3Q[i+1]$$
$$W_4[i+1] = 10Q[i+1] + (-3Q[i+1]); \qquad W_5[i+1] = 10Q[i+1] + (-Q[i+1])$$

Next, each $W_k[i+1]$ is added to the constant value $10^{-(i+1)}$K by a redundant decimal adder with $[-6,6]$ as the digit set.

Regarding the admissible error of the Root Digit Selection (RDS) (Eqn. 8.14) the four most significant digits of the comparison multiples and the shifted partial remainder are required to be involved in the RDS. This block is meant to generate the output carries (i.e., $\in \{-1,0,1\}$) of the addition of $10w[i+1] - M_k[i+1]$.

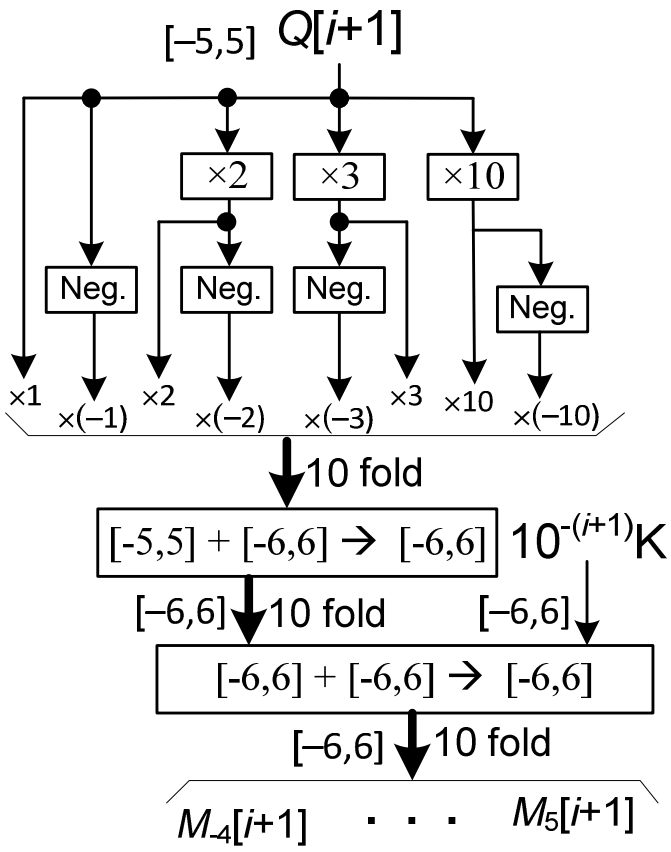


Figure 8.6: Comparison Multiples Generation

With the purpose of reducing the latency and complexity of this carry-generation block, $(10w[i+1] - M_k[i+1])$ is represented as shown in Fig. 8.7, where white (black) dots symbolize negative- (positive-) weighted bits. Consequently, only 13 bits of each operand are required to meet the error bounds in Eqn. 8.14 (i.e., $|\varepsilon[i]| < 0.01055$ ...).

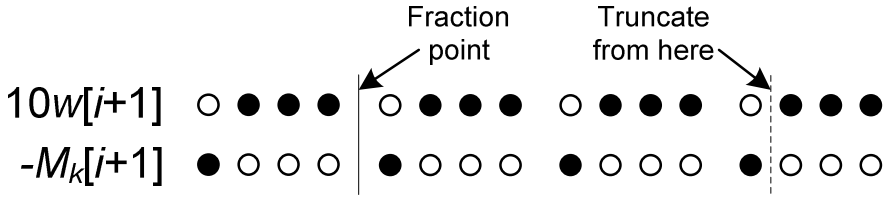


Figure 8.7: Bit representations used in RDS

Next, the comparing signal is produced, based on the values of the carry bits, to indicate whether $10w[i+1] \geq M_k[i+1]$. Finally, an encoder is used to generate q_{i+2}, given the 10 comparison signals. The architecture of the recurrence stage is shown in Fig. 8.8.

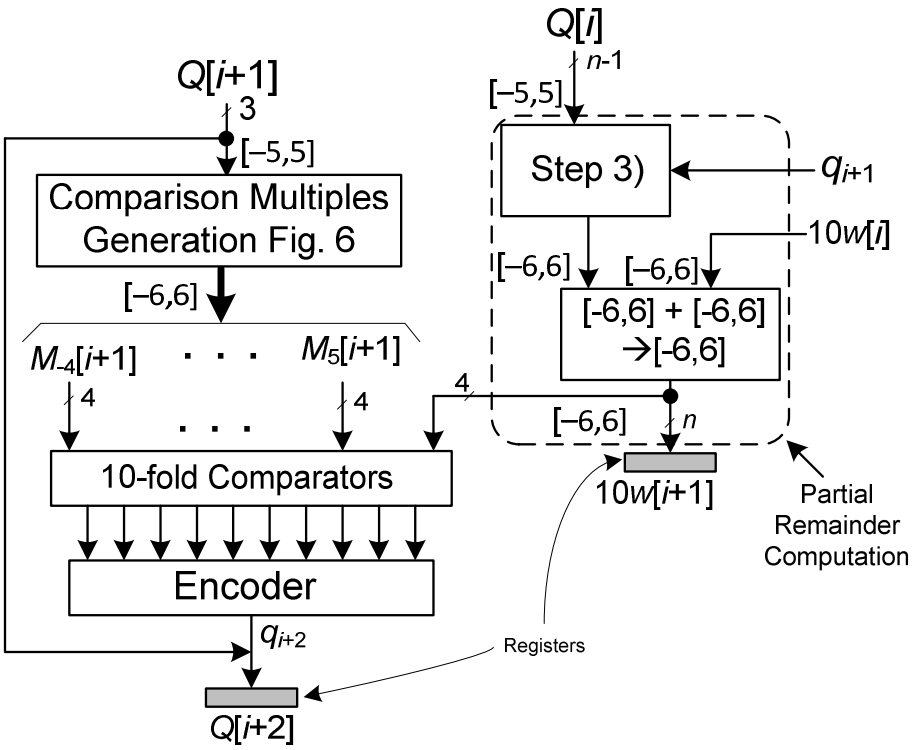


Figure 8.8: The Proposed Architecture of the Recurrence Stage

In the initialization stage, according to Fig. 8.2, there is a need to convert the BCD representation of the radicand to redundant decimal encoding ($[-6,6]$) and to compare the most significant digit of the radicand with 3. Next, based on this comparison result the most significant digit of the redundant radicand takes the value of $\{-1,0,1\}$. In the termination stage, some operations are needed to convert the minimally redundant decimal root Q to the final BCD result, namely conversion to BCD, rounding and normalization.

These operations are performed in the standard manner explained in division papers [37] where conversion and normalization are done on-the-fly and the rounding mode is RoundTiesToEven.

8.3 Evaluations and Comparisons of Decimal Square-Root Units

The evaluation results of the proposed architecture, in terms of latency and area, are presented and compared with the fastest previous work, in this section. The proposed design is synthesized by Synopsys Design Compiler using the STM $90nm$ CMOS standard library [26] for 1.00 VDD and 25°C temperature in which an FO4 (i.e., the latency of an inverter driving 4 similar inverters in the output) is $45ps$ and the area of a NAND2 is $4.4\mu m^2$.

According to Fig. 8.8 the recurrence stage of the proposed multiplier consists of three main parts namely comparison multiples generation, partial remainder computation and root digit selection. The simulation results show that the critical path delay consists of comparison multiples generation and root digit selection. Moreover, the number of cycles required for a 16-digit root is 1+17+1=19. Table 8.1 illustrates the critical path delay, the number of cycles and the total latency of the 16-digit proposed decimal square-root architecture.

Table 8.1: Critical Path Delay of the Proposed Design (16 digits)

	Register	x(-3)	Adder 1	Adder 2	Comparator	Encoder	Cycle Time	# of Cycles	Total Latency
Delay (ns)	0.17	0.22	0.51	0.54	0.26	0.10	**1.80**	19	**34.2**

The area consumption of the proposed 16-digit architecture is evaluated as the sum of the area cost of various constituent parts tabulated in Table 8.2. The cycle time of the proposed decimal square root architecture is 40 FO4 and for the 16-digit root the total latency and area are 760 FO4 and about 31,000 NAND2, respectively.

Table 8.2: Area consumption of the proposed 16-digit architecture (NAND2)

	Combinational	Registers	Total
Initialization	276	441	717
Recurrence	20,870	4,743	25,613
Termination	3,662	1,000	4,662
Whole Design	24,808	6,184	**$\approx 31K$**

The fastest previous pertinent works [57, 58] is based on the Newton-Raphson iterative method where a decimal FMA is the main building block. This is a floating-point decimal square root unit with cycle time of 62.22 FO4 [57] and requires 15 cycles to compute a 16-digit root; thus the total latency of 933.3 FO4. The area of this design is reported as 157,284 NAND2.

For fair comparison the latency and area of the proposed design is estimated for the floating-point computation. In this case, a pre-processing unit is required to convert the radicand from Densely-Packed-Decimal (DPD) to BCD encoding, determine the number of leading zeros and normalize the radicand. This pre-processing unit adds one extra cycle to the proposed fixed-point design and consumes about 2378 NAND2 of area [56]. Moreover, a post-processing unit is required to deal with the exponents, handle the exceptions and convert the BCD root back to the IEEE 754-2008 format. This can be performed in the termination stage with the extra area of 3,027 NAND2 [56].

Consequently, using the proposed decimal square root architecture for the floating-point computation leads to the total latency of $40 \times 20 = 800$ FO4 and consume the area of about 36,400 NAND2. Table 8.3 compares the proposed design with that of [57, 58] and the work based on the CORDIC algorithm [61], in terms of latency and area. There is also another work, based on the digit-recurrence algorithm [59], which uses look-up tables, small adders and multipliers. However, it is optimized and implemented on FPGA and hence no ASIC evaluation results are available to compare with.

Table 8.3: Comparison of the FP architectures

	Cycle time (FO4)	# of cycles	Total Latency (FO4)	Ratio	Area (NAND2)	Ratio
Proposed	40.00	20	800.0	1.00	36,400	1.00
[57]	62.22	15	933.3	1.16	157,284	4.32
[61]	34.63	35	1211	1.51	18,826 + 4.5KB	---

According to the comparison results the proposed design is about 14% faster than the fastest previous work with about a quarter of the area. This implies that, due to the high latency and area cost of decimal multipliers, using digit-recurrence algorithms for computing decimal square-root is more efficient.

CHAPTER 9
CONCLUSIONS & FUTURE WORKS

9.1 Conclusions

Co-processors, working in parallel with primary processors, are meant to supplement the functions of general-purpose processors. Computationally intensive tasks are usually delegated to co-processors so as to offload heavy functions from the primary processor, which leads to higher overall performance. Since a co-processor, unlike a general-purpose processor, is designed for a specific application (e.g., floating-point, FFT, decimal arithmetic), it can be a custom-designed product that allows for use of unconventional methods and algorithms to increase the performance of the co-processor.

One of the unconventional approaches to speeding up an arithmetic co-processor is using redundant number systems. This delays carry-propagation until the last phase of the whole operation and leads to a much faster architecture. However, using redundant number systems adds extra complexity to the arithmetic circuitries, which usually increases area. Therefore, the number system and the corresponding arithmetic units must be designed meticulously to reach the optimized balance between area and speed.

In this thesis, new architectures for two co-processors (FFT and decimal) are presented and discussed. It is shown how redundant number systems improve the speed and performance of these co-processors. The first part is the proposal of a high-speed FFT architecture that is much faster than the best previous works at the cost of higher area. The reasons for this speed improvement are twofold:

- Using Binary-Signed-Digit (BSD) representation for the significands of the floating-point operands to eliminate carry-propagation.
- The proposed Fused-Dot-Product-Add (FDPA) unit that combines the multiplications and additions required in a floating-point butterfly unit. Higher speed is achieved by eliminating extra leading-zero-detection (LZD), normalization and rounding units.

High-speed floating-point FMAs performing over complex numbers are required in order to have a high-performance butterfly unit. This, in turn, requires floating-point multipliers along with high-speed three-operand floating-point adders. Floating-point multipliers are typically designed based on the conventional architecture or Golub's approach. It is shown that Golub's method does not have any advantage over the conventional one when being used over floating-point operands. Therefore, a new redundant floating-point multiplier is designed based on the conventional architecture. A new three-operand redundant floating-point adder is developed. Modified Booth encoding is also used to speed up the proposed constant Binary-Signed-Digit multiplier. Also, operands are stored in registers and used in the next stage as a redundant representation; thus, carry-propagation is involved neither inside a butterfly unit nor between FFT stages. This results in a faster FFT processor, but a larger area due to the need for more registers to store redundant operands.

For the second part, four of the most useful decimal operations (addition, multiplication, division and square-root) are implemented in hardware and it is shown that using redundant number systems improves the clock frequency of these decimal arithmetic units. A decimal signed digit adder that uses the stored carry representation of the operands is proposed. Using this representation requires $[-9,7]$, as the digit-set. It is shown that this representation simplifies

the final addition by removing the input carry. The proposed adder consists of three main building blocks:

1) Combinational logic F1 to generate the transfer bits and the high-portion of the intermediate sum
2) Combinational logic F2 to generate the low-portion of the intermediate sum
3) Three-bit CLA to generate the final sum

The performance of the proposed redundant adder was compared with that of the fastest previous work. The results demonstrated that the proposed architecture is 15 % faster than the most latency efficient previous work while sacrificing some area and power.

A high-frequency sequential decimal multiplier is also proposed where the easy-multiples (i.e.,$X, 2X, 4X, 5X$) are used to generate the partial products represented in 4-2-2-1 encoding. The cycle-time of the proposed sequential multiplier is minimized by using efficient decimal encodings and by retiming the constituent parts of a decimal carry-save adder. It is shown that the proposed pipelined design, with its cycle time of 10 FO4, is much faster than the previous works, while the proposed word-serial implementation keeps the area as low as possible. The fastest previous design [34] works with 47% slower clock frequency and consequently a 27% slower multiplier than the proposed pipelined architecture.

For decimal division, firstly, the general rules and conditions for quotient digit selection (QDS) are presented in decimal digit-recurrence division algorithms with operands represented in generalized signed-digit (GSD) format. As a consequence of this generalization, the convergence condition usually used in division algorithms is unnecessarily strict and conservative, which might exclude some correct algorithms. Using the proposed general condition circumvents this problem and allows for more efficient dividers. Secondly, the

suggested representations were applied into the fastest decimal divider and gained about 10% speed improvement with almost the same area cost.

Finally, a decimal square-root architecture is proposed based on the SRT algorithm in which the root is computed iteratively. A root digit is determined by comparing the partial remainder to ten inconstant comparison multiples. The radix-10 minimally redundant representation of the root (i.e., digit-set of $[-5,5]$) leads to a simple root digit selection unit. The partial remainder, however, is represented in the digit-set equal to $[-6,6]$ so as to reduce the complexity of the intermediate redundant decimal adders. This endeavor leads to an architecture that is 14% faster than the fastest previous work (based on the Newton-Raphson algorithm) with about a quarter of the area. It is also suggested that implementing decimal square-root using digit recurrence algorithms is more efficient than the designs based on functional methods e.g., Newton-Raphson. The main reason lies within the fact that functional methods necessitate a decimal multiplication per iteration.

9.2 Future Works

There are two approaches which can be pursued in the future as an extension of the current work described in this thesis.

- Conduct research on redundant number systems for other useful co-processors (e.g., for graphic and encryption applications).
- Search for more efficient redundant number systems for each of the constituent blocks of the FFT and decimal co-processors discussed in this thesis.

In the first approach, one may need to identify the performance bottleneck of the desired co-processor where multiple consecutive arithmetic operations are being performed. The next

step is finding an efficient redundant representation for the operands involved in the bottleneck. Finally, a high-speed algorithm and architecture are required to perform the corresponding redundant arithmetic operations.

The second approach includes finding other redundant representations for FFT Butterfly units, decimal adders and subtractors, decimal sequential and parallel multipliers, fused-multiply-add units, decimal dividers and decimal square-root units (both digit-recurrence and multiplicative-based algorithms).

Furthermore, one could pursue the implementation of FFT co-processors based on range addressable look-up tables. Using this method, the FFT co-processor is expected to be faster. The cost of these look-up tables are lower than the conventional look-up tables and hence saves the area cost.

Implementing redundant floating-point FFT co-processors based on CORDIC (COrdinate Rotation DIgital Computer) algorithms sounds a promising research topic to be worked on in the future. Using the CORDIC approach saves area cost but sacrifices the speed of the FFT co-processor.

REFERENCES

[1] Swartzlander, E. E. Jr. and H. H. Saleh, "FFT Implementation with Fused Floating-Point Operations," *IEEE Transactions on Computers*, Vol. 61, No. 2, pp. 284-288, 2012.

[2] Sohn, J. and E. E. Swartzlander, Jr., "Improved Architectures for Floating-Point Fused Dot Product Unit," *Proceedings of IEEE 21st Symposium on Computer Arithmetic*, pp. 38-41, 2013.

[3] IEEE Standards Committee, "*754-2008 IEEE Standard for Floating-Point Arithmetic*," pp. 1-58, August 2008.

[4] Swartzlander, E. E. Jr. and H. H. Saleh, "Fused Floating-Point Arithmetic for DSP," *Proceedings of the 42nd Asilomar Conference on Signals, Systems and Computers*, pp. 767-771, 2008.

[5] Min, J. H, S. W. Kim and E. E. Swartzlander, Jr., "A Floating-Point Fused FFT Butterfly Arithmetic Unit with Merged Multiple-Constant Multipliers," *Proceedings of the 45th Asilomar Conference on Signals, Systems and Computers*, pp. 520-524, 2011.

[6] Jaberipur, G, B. Parhami and M. Ghodsi, "Weighted Bit-Set Encodings for Redundant Digit Sets: Theory and Applications," *Proceedings of the 36th Asilomar Conference on Signals Systems and Computers*, pp. 1629-1633, 2002.

[7] Parhami, B., "*Computer Arithmetic: Algorithms and Hardware Designs*," 2nd Edition, Oxford University Press, New York, 2010.

[8] Cowlishaw, M. F., "Decimal Floating-Point: Algorithm for Computers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pp. 104-111, 2003.

[9] Eisen et al. "IBM POWER6 Accelerators: VMX and DFU," *IBM Journal of Research and Development*, Vol. 51, No. 6, pp. 663-684, 2007.

[10] Cooley, J. W. and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computations*, Vol. 19, No. 90, pp. 297-301, 1965.

[11] Dubois, E. and A. Venetsanopoulos "A New Algorithm for the Radix-3 FFT," *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. 26, No. 3, pp. 222-225, 1978.

[12] Taylor, F. J., G. Papadourakis, A. Skavantzos and A. Stouraitis, "A Radix-4 FFT Using Complex RNS Arithmetic," *IEEE Transactions on Computers*, Vol. C-34, No. 6, pp. 573-576, 1985.

[13] Singleton, R. C. "An algorithm for computing the mixed radix fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, Vol. 17, No. 2, pp. 93-103, 1969.

[14] Duhamel, P. and H. Hollmann, "Split-Radix FFT Algorithm," *IET Electronics Letters*, Vol. 20, No. 1, pp. 14-16, 1984.

[15] Winograd, S., "On computing the discrete Fourier transform" *Mathematics of Computation*, Vol. 32, pp. 175–199, 1978.

[16] Bluestien, L. I, "A linear filtering approach to the computation of the discrete Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, Vol. 18, No. 4, pp. 451-455, 1970.

[17] Ercegovac, M. D. and Lang, T. "*Digital Arithmetic*," Morgan Kaufmann Publishers, 2004.

[18] Parhami, B., "Generalized Signed-Digit Number Systems: A Unifying Framework for Redundant Number Representations," *IEEE Transactions on Computers*, Vol. 39, No. 1, pp. 89-98, 1990.

[19] Swartzlander, E. E. Jr. and H. H. Saleh, "Floating-Point Implementation of Complex Multiplication," *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, pp. 926-929, 2009.

[20] Fahmy, A. H. and M. Flynn, "The Case for a Redundant Format in Floating-Point Arithmetic," *Proceedings of IEEE 16th Symposium on Computer Arithmetic*, pp.95-102, 2003.

[21] Tenca, A. F. "Multi-Operand Floating-Point Addition," *Proceedings of IEEE 19th Symposium on Computer Arithmetic*, pp. 161-168, 2009.

[22] Tao, Y., g. Deyuan, F. Xiaoya and R. Xianglong, "Three-Operand Floating-Point Adder," *Proceedings of the 12th IEEE International Conference on Computer and Information Technology*, pp. 1920196, 2012.

[23] Nielsen, A. M., D. W. Matula, C. N. Lyu and G. Even, "An IEEE Compliant Floating-Point Adder that Conforms with the Pipelined Packet-Forwarding Paradigm," *IEEE Transactions on Computers*, Vol. 49, No. 1, pp. 33-47, 2000.

[24] Kornerup, P., "Correcting the Normalization Shift of Redundant Binary Representations," *IEEE Transactions on Computers*, Vol. 58, No. 10, pp. 1435-1439, 2009.

[25] Oklobdzija, V. G., "An Algorithmic and Novel Design of a Leading Zero Detector Circuit: Comparison with Logic Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2, No. 1, pp. 124-128, 1994.

[26] STMicroelectronics, 90nm CMOS090 Design Platform, 2007.

[27] Karatsuba, A. and Y. Ofman, "Multiplication of Many-Digital Numbers by Automatic Computers," *academic journal Physics-Doklady*, Vol. 7, pp. 595–596, 1963.

[28] Svoboda, A. "Decimal Adder with Signed Digit Arithmetic," *IEEE Transactions on Computers*, Vol. C-18, No. 3, pp. 212-215, 1969.

[29] Shirazi, B., D.Y. Yun and C.N. Zhang, "RBCD: Redundant Binary Coded Decimal Adder," *IEE Proc. Computer & Digital Techniques (CDT)*, Vol.36, No.2, 1989.

[30] Kaivani, A. and G. Jaberipur, " Fully redundant decimal addition and subtraction using stored-unibit encoding," *Integration, the VLSI Journal*, Vol. 43, No. 1, pp. 34-41, 2010.

[31] Gorgin, S. and G. Jaberipur, "Fully Redundant Decimal Arithmetic," *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, pp. 145-152, 2009.

[32] Jaberipur, G., B. Parhami and M. Ghodsi, "A Class of Stored-Transfer Representations for Redundant Number Systems," *Proceedings of 35th Asilomar Conf. Signals Systems and Computers*, pp. 1304-1308, 2001.

[33] Schwarz, E., M., J. S. Kapernick and M. F. Cowlishaw, "Decimal Floating-Point Support on the IBM System z10 Processor," *IBM Journal of Research and Development*, Vol. 53, No. 1, pp. 4:1-4:10, 2009.

[34] Carlough, S., A. Collura, S. Mueller and M. Kroener, "The IBM Zenterprise-196 Decimal Floating-Point Accelerator," *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pp. 139-146, 2011.

[35] Jaberipur, G. and A. Kaivani, "Binary-Coded Decimal Digit Multipliers," *IET Computers & Digital Techniques*, Vol. 1, No. 4, pp. 377-381, July 2007.

[36] Richards, R. K., "*Arithmetic Operations in Digital Computers*," Van Nostrand, New York, 1955.

[37] Lang, T. and A. Nannarelli, "A Radix-10 Digit-Recurrence Division Unit: Algorithm and Architecture," *IEEE Transactions on Computers*, Vol. 56, No. 6, pp. 727-739, 2007.

[38] Vazquez, A., E. Antelo and P. Montuschi, "Improved Design of High-Performance Parallel Decimal Multipliers," *IEEE Transactions on Computers*, Vol. 59, No. 5, pp. 679-693, 2010.

[39] Vazquez, A., E. Antelo, and P. Montuschi, "A New Family of High-Performance Parallel Decimal Multipliers," *Proceedings of 18th IEEE Symposium on Computer Arithm*etic, pp. 195-204, 2007.

[40] Erle, M. A. and M. J. Schulte, "Decimal Multiplication via Carry-Save Addition," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 348–358, 2003.

[41] Vazquez, A. and E. Antelo, "Conditional Speculative Decimal Addition," *Proceedings of the 7th Conference on Real Numbers and Computers (RNC7)*, pp. 47–57, 2006.

[42] Kenney, R. D., M. J. Schulte and M. A. Erle, "A High-Frequency Decimal Multiplier," *Proceedings of the IEEE International Conference on Computer Design*, pp. 26-29, 2004.

[43] Erle, M. A., E. M. Schwarz and M. J. Schulte, "Decimal Multiplication with Efficient Partial Product Generation," *Proceedings of 17th IEEE Symposium on Computer Arithmetic*, pp. 21-28, 2005.

[44] Vazquez, A. Antelo and E. P.Montuschi, "A Radix-10 SRT Divider Based on Alternative BCD Codings", *25th IEEE International Conference on Computer Design (ICCD 2007)*, pp. 280-287, 2007.

[45] Schwarz, E. M., and S. R. Carlough, "Power6 Decimal Divide," *Proceedings of IEEE International Conference on Application-Specific Systems, Architecture, Processors (ASAP)*, pp. 128-133, 2007.

[46] Kaivani, A., A. Hosseiny, G. Jaberipur, "Improving the Speed of Decimal Division," *IET Computer & Digital Techniques*, Vol. 5, No. 5, pp. 393-404, September 2011.

[47] Ercegovac, M. D., "A higher-radix division with simple selection of quotient digits," *in Proceedings of 6th IEEE Symposium on Computer Arithmetic (ARITH6)*, pp. 94-98, 1983.

[48] Montuschi, P. and L. Ciminiera, "Over-Redundant Digit Sets and the Design of Digit-by-Digit Division Units," *IEEE Transactions on Computers*, Vol. 43, No. 3, pp. 269-277, 1994.

[49] Tang, P. T. P., J. A. Butts, R. O. Dror and D. E. Shaw, "Tight Certification Techniques for Digit-by-Rounding Algorithms with Application to a New $\frac{1}{\sqrt{x}}$ Design," *in Proceedings of 20th IEEE Symposium on Computer Arithmetic (ARITH20)*, pp. 159-168, 2011.

[50] Butts, J. A., P. T. P. Tang, R. O. Dror and D. E. Shaw, "Radix-8 Digit-by-Rounding: Achieving High-Performance Reciprocals, Square Roots, and Reciprocal Square

Roots," *in Proceedings of 20th IEEE Symposium on Computer Arithmetic (ARITH20)*, pp. 149-158, July 2011.

[51] Jaberipur, G., B. Parhami and M. Ghodsi, "Weighted Two-Valued Digit-Set Encodings: Unifying Efficient Hardware Representation Schemes for Redundant Number Systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 52, No. 7, pp. 1348-1357, 2005.

[52] Parhami, B., "Precision Requirements for Quotient Digit Selection in High-Radix Division," *Proceedings of 35th Asilomar Conference on Circuits, Systems, and Computers*, pp. 1670-1673, 2001.

[53] Parhami, B., "Tight Upper Bounds on the Minimum Precision Required of the Divisor and the Partial Remainder in High-Radix Division," *IEEE Transactions on Computers*, Vol. 52, No. 11, pp. 1509-1514, 2003.

[54] Kornerup, P., "Digit Selection for SRT Division and Square Root," *IEEE Transactions on Computers*, Vol. 54, No. 3, pp. 294-303, 2005.

[55] Parhami, B., "On the implementation of arithmetic support functions for generalized signed-digit number systems," *IEEE Transactions on Computers*, Vol. 42, No. 3, pp. 379-384, 1993.

[56] Wang, L. K. and M. J. Schulte, "Decimal Floating-Point Square Root Using Newton-Raphson Iteration," *Proceedings of the 16th International Conference on Application Specific Systems, Architecture and Processors*, pp. 309-315, 2005.

[57] SilMinds, "DFP Newton-Raphson Square Root Units," *IP Core Product Data Sheet, NRDecDiv64/128*.

[58] Raafat, Ramy et al., "Decimal Floating-Point Square-Root Unit Using Newton-Raphson Iterations," *US Patent Application Publication, US 2012/0011182*, 2012.

[59] Ercegovac, M. D. and R. McIlhenny, "Design and FPGA Implementation of Radix-10 Algorithm for Square Root with Limited Precision Premitives," *Proceedings of the 43rd Asilomar Conference on Signals, Systems and Computers*, pp. 935-939, 2009.

[60] Vazquez, A., J. Villalba and E. Antelo, "Computation of Decimal Transcendental Functions Using the CORDIC Algorithm," *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, pp. 179-186, 2009.

[61] Kaivani, A. and G. Jaberipur, "Decimal CORDIC Rotation based on Selection by Rounding," *The Computer Journal*, Vol. 54, No. 11, pp. 1798-1809, 2011.

[62] Gorgin, S. and G. Jaberipur, "A Family of Signed Digit Adders," *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pp. 112-120, 2011.

[63] Kaivani, A. and S. Ko, "Floating-Point Butterfly Architecture Based on Binary Signed-Digit Representation," *IEEE Transactions on Very Large Scale Integration*, to appear.

[64] Kaivani, A. and S. Ko, "High-Speed FFT Processors Based on Redundant Number Systems," *IEEE International Symposium on Circuits and Systems (ISCAS'14)*, pp. 2237-2240, 2014.

[65] Kaivani, A. and S. Ko, "Decimal signed digit addition using stored transfer encoding," *26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'13)*, pp.1-4, 2013.

[66] Kaivani, A., L. Han and S. Ko, "Improved design of high-frequency sequential decimal multipliers," *IET Electronics Letters*, Vol. 50, No. 7, pp. 558-560, 2014.

[67] Kaivani, A., L. Chen and S. Ko, "High-frequency sequential decimal multipliers," IEEE International Symposium on Circuits and Systems (ISCAS'12), pp. 3045-3048, 2012.

[68] Han, L., A. Kaivani and S. Ko, "Area Efficient Sequential Decimal Fixed-point Multiplier," *Journal of Signal Processing Systems*, Vol. 75, No. 1, pp. 39-46, 2014.

[69] Kaivani, A. and S. Ko, "Decimal Division Algorithms: The Issue of Partial Remainders," *Journal of Signal Processing Systems*, Vol. 73, No. 2, pp. 181-188, 2013.

[70] Kaivani, A. and S. Ko, "Decimal SRT Square Root: Algorithm and Architecture," Circuits, Systems and Signal Processing, Vol. 32, No. 5, pp.2137-2150, 2013.