UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

A HIGH-PERFORMANCE INNER-PRODUCT PROCESSOR FOR REAL AND COMPLEX NUMBERS

A Dissertation

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

degree of

Doctor of Philosophy

By

GUOPING WANG

Norman, Oklahoma

2003

A HIGH-PERFORMANCE INNER-PRODUCT PROCESSOR
FOR REAL AND COMPLEX NUMBERS

A Dissertation APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY

Monte P. Tull
Dr. Monte P. Tull

[illegible]
Dr. Gerald Crain

[illegible]
Dr. Linda DeBrunner

[illegible]
Dr. Joe Havlicek

[illegible]
Dr. Murad Ozaydin

To my family
who have lovingly supported my years of study.

ACKNOWLEDGMENTS

In the years leading to this dissertation, I have had the great pleasure of working for and with an advisor who provided an environment conducive to learning. I would like to take this opportunity to thank my advisor, Dr. Monte P. Tull for his consistent guidance.

I would also like to thank Dr. Gerald Crain, Dr. Linda DeBrunner, Dr. Joe Havlicek and Dr. Murad Ozaydin for serving on my supervisory committee.

I appreciate many others who have been helpful at the University of Oklahoma. They make my years at the university a unique and memorable experience.

I am especially indebted to my family for their support. Guanglan Zhang, my wife, has been invaluable for her patience and love during my research. My daughter, Ying, and my son, Christopher always give me great joy and peace.

"This is what the Lord says ...
'Call to me and I will answer you and
tell you great and unsearchable things
you do not know.' "

Jeremiah 33:2-3

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT OF THE DISSERTATION

A High-Performance Inner-Product Processor for Real and Complex Numbers

by

Guoping Wang
Doctor of Philosophy in Electrical and Computer Engineering
University of Oklahoma, Norman, OK, 2003
Dr. Monte P. Tull, Chair

A novel, high-performance fixed-point inner-product processor based on a redundant binary number system is investigated in this dissertation. This scheme decreases the number of partial products to 50%, while achieving better speed and area performance, as well as providing pipeline extension opportunities. When modified Booth coding is used, partial products are reduced by almost 75%, thereby significantly reducing the multiplier addition depth. The design is applicable for digital signal and image processing applications that require real and/or complex numbers inner-product arithmetic, such as digital filters, correlation and convolution. This design is well suited for VLSI implementation and can also be embedded as an inner-product core inside a general purpose or DSP FPGA-based processor. Dynamic control of the computing structure permits different computations, such as a variety of inner-product real and complex number computations, parallel multiplication for real and complex numbers, and real and complex number division. The same structure can also be controlled to accept redundant binary number inputs for multiplication and inner-product computations. An improved 2's-complement to redundant binary converter is also presented.

Chapter 1 Introduction

Consider the definition of the inner-product. For two vectors $\boldsymbol{A} = (A_0, A_1, A_2, ...A_{M-1})$, and $\boldsymbol{B} = (B_0, B_1, B_2, ...B_{M-1})$, the inner-product of $\boldsymbol{A}$ and $\boldsymbol{B}$ is defined as:

$$<\boldsymbol{A,B}> = \boldsymbol{A}\left(\boldsymbol{B}^*\right)^T = \sum_{i=0}^{M-1} A_i B_i^* \qquad (1.1)$$

In general, A_i and B_i may be real or complex numbers. $\boldsymbol{AB}^*$ denotes matrix multiplication with the row vectors A and B considered as $1 \times M$ matrices, and $(B^*)^T$ denotes the conjugate transpose of B. In the traditional method, all of the multiplications are processed independently of one another, thereby requiring M multiplications and M-1 additions. To obtain high-performance circuit implementations of the inner-product, several salient features of Equation (1.1) can be utilized; namely, carry-free addition, high-speed multiplication, and parallel or pipelined multiplication and addition.

The application of redundant binary (RB) numbers was previously investigated for carry-free addition and fast multiplication. These techniques have proven to be easily laid out in VLSI and result in high-speed circuit implementations [1]-[4]. In this dissertation, high-performance and easily pipelined implementations of an inner-product processor are presented. The designs utilize RB numbers for achieving the carry-free addition of partial products. Redundant binary schemes are less viable in applications that require persistent conversion back to 2's-complement [5]-[7], since this process is relatively slow due to an unavoidable carry propagation requirement. The overall motivation for this work is the design of a high-performance Complex Arithmetic Signal Processor (CASP) capable of offering novel extended inner-product operations. The

CASP design relies on the high-speed multiplication afforded by redundant binary techniques, while avoiding the relatively slow conversion back to 2's-complement numbers until a final 2's-complement result is necessary. Inherently, the CASP device provides intermediate register storage for redundant binary, as well as 2's-complement numbers. The methods for implementing the core inner-product structure and general extensions are presented in this dissertation.

Inner-product computations play a central role in digital signal processing, most often in digital filters, signal correlation, convolution, FFT, etc. Current implementations of inner-product computations include the following methods: 1) general purpose processors, 2) digital signal processor devices, such as Texas Instruments TMS320C60, 3) VLSI devices, such as FPGAs or ASICs. Various researchers have investigated the implementation of inner-product processors. Implementations include array multipliers [8],[9], VLSI Residue Number System architecture [10], serial implementations [11],[12], distributed arithmetic [13],[14], carry-save addition [15]-[19], specific DSP processor and FPGA [20]-[27], redundant binary implementations [28]-[30].

Complex number arithmetic computation is a key arithmetic feature required in modern digital communication and optical systems [31]-[38]. Many algorithms based on convolution, correlation, and complex number filters require complex number multiplication and high-speed inner-product computation. These applications require efficient representation and manipulation of complex numbers together with real numbers. Considerable research exists for hardware implementations of complex number systems [39]-[54] and representations of complex numbers in different radices [55]-[68].

The redundant binary (RB) representation is one of the signed-digit number representations originally introduced by Avizienis [69] for achieving the carry-propagation-free addition. RB numbers differ from the conventional 2's-complement representation in that the individual digits comprising a number may have negative values as well as positive values. High-speed VLSI multiplication algorithms, which are based on redundant binary numbers, are proposed in [1],[3],[4]. Since integer numbers in most digital systems are represented in 2's-complement form, a converter is needed to convert a redundant binary number to a 2's-complement number in the last step. Different implementations of this converter have been proposed in [5], [7], [12], [70]-[74].

Although division is an infrequent operation, it has been shown [75] that ignoring its hardware implementation can result in significant system performance degradation for many applications. Extensive literature describes the theory of division [75]-[90]. Division algorithms can be generally divided into the following classes: digit recurrence, functional iteration, table look-up and variable latency [84]. Choosing an optimal design of a divider depends heavily upon its requirements for area and speed.

In the following sections, these hardware implementations and research issues will be reviewed and investigated.

1.1 Inner-Product Implementation by the General Purpose Processors

General purpose processors, such as Intel Pentium and 80x86, Motorola 68000, AMD K6 and K7, *etc.*, can perform different algorithms using combinations of various machine instructions. The systems built with these programmable processors are adaptable to different applications and easily upgradeable to changing requirements.

Even with such potential advantages, traditional programmable processors have not been widely used for high-speed inner-product computation because of their limited performance. For example, in order to find the inner-product of two vectors ***A*** and ***B***, the flowchart in Figure 1-1 is usually employed.

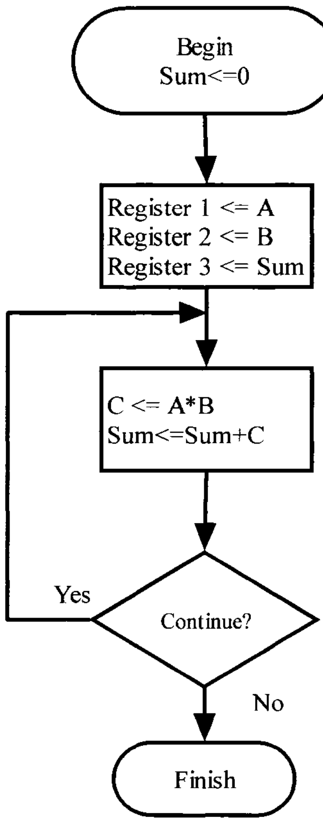


Figure 1-1. Flowchart of Inner-Product Computation by the General Purpose Processors

In a general purpose (GP) processor, all these computations are sequential and each load, multiplication or summation requires one or more clock cycles. Traditional multiplication and accumulation methods are generally used. Some GP processors provide additional hardware features for inner-product calculations. Among these processors, the Pentium MMX processor contains a super scalar architecture, which includes: 1) enhanced pipelines 2) two pipelined integer units capable of two instructions per clock, as well as other features. With the new architecture, the Pentium MMX can compute inner-products more efficiently than other general purpose processors. The diagram of inner-product implementation by the Pentium MMX processor is shown as Figure 1-2 [91]:

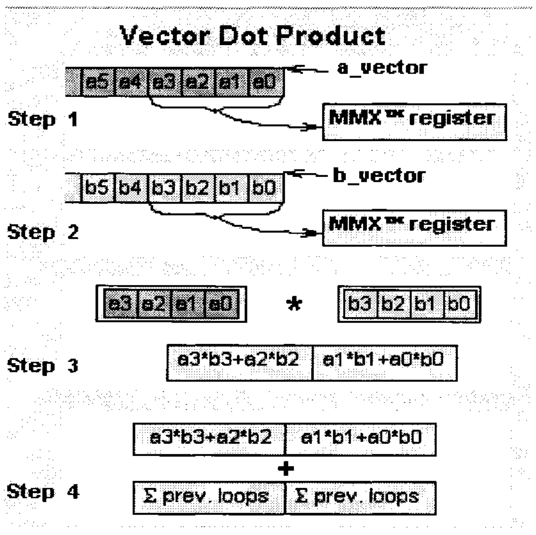
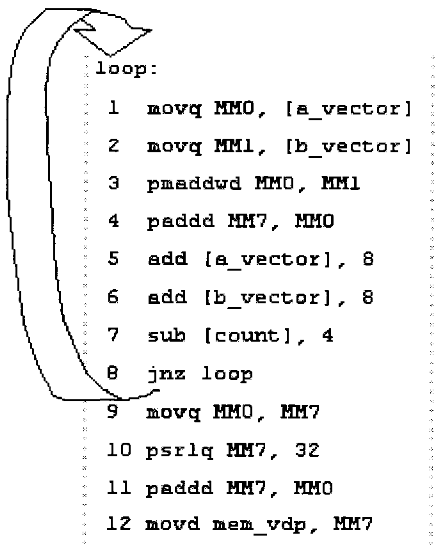


Figure 1-2. Inner-Product Implementation in Pentium MMX Processor [91]

Sample code [91] for an inner-product implementation using Pentium assembly language is shown Figure 1-3:



```
loop:
movq MM0, [a_vector]
movq MM1, [b_vector]
pmaddwd MM0, MM1
paddd MM7, MM0
add [a_vector], 8
add [b_vector], 8
sub [count], 4
jnz loop
movq MM0, MM7
psrlq MM7, 32
paddd MM7, MM0
movd mem_vdp, MM7
```

Figure 1-3. Sample Code for an Inner-Product by Pentium MMX [91]

1.2 Inner-Product Implementation by Digital Signal Processing Processors

DSP processors are specifically designed for DSP applications. One typical DSP processor is the Texas Instruments TMS320C60. It is a highly integrated, multiprocessor, single chip device specifically designed for DSP applications. The TMS320C60 integrates the following components onto a single device [92]:

1. a single 32-bit RISC master processor (MP) with an integral IEEE-754 floating point unit
2. four 32-bit integer DSP parallel processors (PP)
3. a sophisticated direct memory access (DMA) transfer controller (TC)
4. a video controller (VC)
5. 50K bytes of on-chip SRAM memory

The five processors on the TMS320C60, i.e. the MP and four PPs, can be configured for a variety of multiple-instruction, multiple-data, multiple-instruction, single-data, or single-instruction, multiple-data modes. The PPs, similar to most DSPs, perform all operations, except division, in a single cycle. For example, it can perform the parallel operations, A*B =>C and A+1 => A in one clock cycle, while in a general purpose processor, at least two cycles are required. Sample code of the fixed-point inner-product computation is shown Figure 1-4 [92].

```
          ZERO .L1    A7
          LDH  .D1    *A4++,A2      ;load ai from memory
          LDH  .D1    *A3++,A5      ;load bi from memory
          MPY  .M1    A2,A5,A6      ; ai * bi
          ADD  .L1    A6,A7,A7      ; sum += (ai * bi)
          SUB  .S1    A1,1,A1; decrement loop counter
[A1]      B    .S2    LOOP          ; branch to loop
```

Figure 1-4. Sample Code of the Fixed-Point Inner-Product by TMS 320C60 [92]

While DSP processors allow flexibility, for some applications that require high speed inner-product computation, FPGAs or ASICs can provide higher performance options.

1.3 Other Inner-Product Processor Implementation Methods

Besides the inner-product implementations on general purpose processors and DSP processors, other arithmetic and implementations of inner-product processors have been investigated. Ahmad and Poornalah [8] proposed an inner-product implementation using array multipliers. Although the array multipliers provide convenient layout for VLSI, this method may not be a good option in high-performance requirements for inner-product computation because of its high latency. Fahmi, *et al.,* [11] and Haynal and Parhami [12] investigated serial implementations of an inner-product processor. The designs result in a small area but has a high latency. Inner-product implementations based on distributed arithmetic are proposed by Burleson *et al.,* [13] and Vega, *et al.,* [14]. Various inner-product implementations using carry-save adders are investigated by many researchers [15]-[19]. Application specific inner-product processors are studied in [21]-[25] and redundant binary implementations are proposed in [28],[29]. In this research, which is focused upon the high-performance implementation of an inner-product processor, only implementations of high-performance inner-product processors will be reviewed and compared.

With a carry-save adder structure, Kazakova [15] investigated a fast and low-power three-dimensional inner-product processor. This processor consists of Booth encoders, a Wallace reduction tree, and a final two-operand adder. Its structure is shown in Figure 1-5.

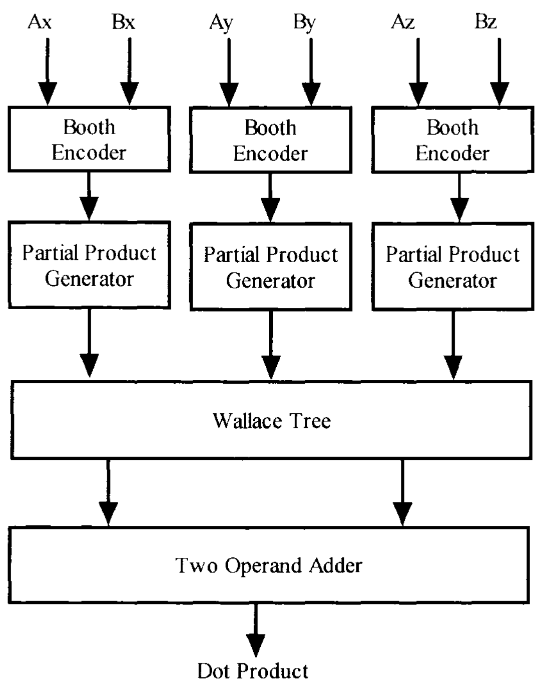


Figure 1-5. Kazakova's Inner-Product Processor Architecture [15]

A novel approach for high-performance inner-product processor, which is dynamically reconfigurable, was proposed by Lin [24],[25]. This processor mainly consists of an 8 × 8 or 4 × 4 array of small multipliers plus two or three arrays of adders. It requires very simple reconfigurable components. The entire summation network can be reconfigured by using a few control bits for the desired computations, where the reconfiguration can be done dynamically. The design is regular, modular, and, it can easily be pipelined. The diagram is shown in Figure 1-6.

Since the implementation of an array multiplier has a high latency, compared to the design of carry-save addition and redundant binary representation, this proposed inner-product processor has a high latency.

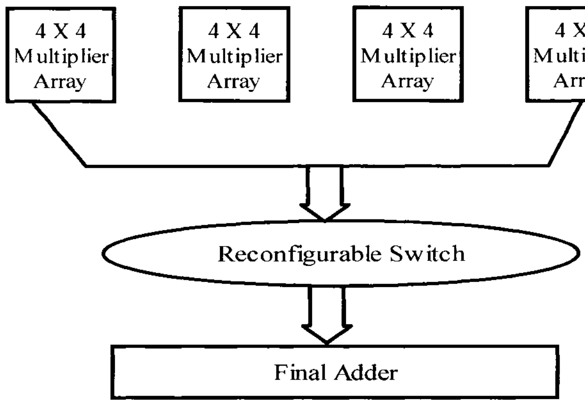


Figure 1-6. Lin's Reconfigurable Inner-Product Processor [24],[25]

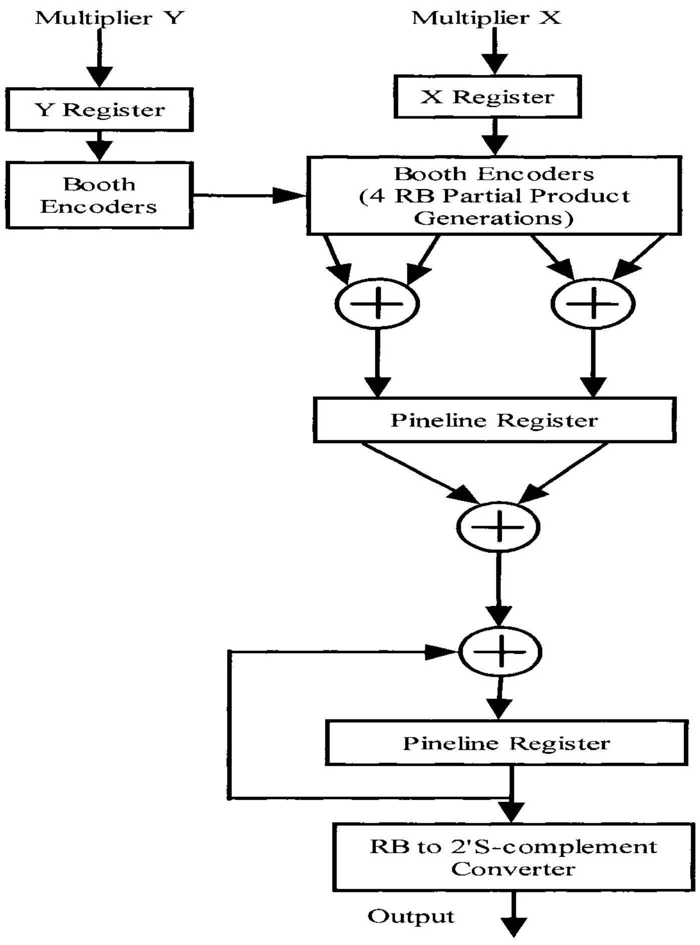


Figure 1-7. A RB MAC (Multiply and Accumulate) by Huang [2]

Based upon redundant binary numbers and the Booth encoding, Huang [2] proposed a high-performance, two-stage pipelined MAC (Multiply and Accumulate) unit, which is shown in Figure 1-7. Later, Sacristan [29] further developed this structure as a reusable inner-product unit for multipliers with different sizes of word length.

Baik *et al.* [28] proposed a redundant binary implementation of an FIR filter. The diagram is shown in Figure 1-8 (See Section 2.1.4. for further discussion):

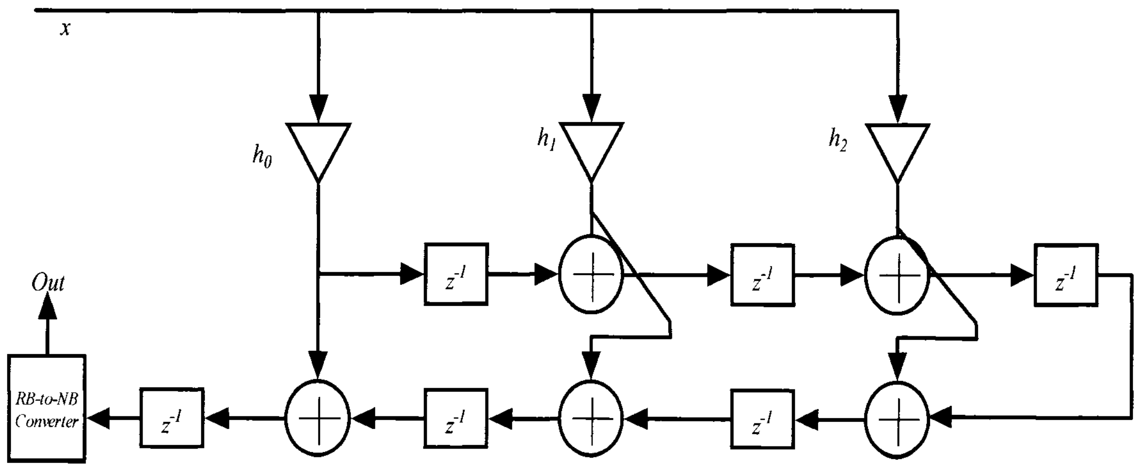


Figure 1-8. Baik's Redundant Binary Filter Implementation [28]

1.4 Multiplier Implementation Review

Multiplication is the key operation in the implementation of inner-product computation. Three popular implementations for multipliers are an array multiplier [85],[93], a multiplier using a Wallace tree [94] and a multiplier using redundant binary number representation [1],[4]. An array multiplier has good repeatability of unit cells and is very regular in its structure. It uses only short wires that connect one full adder to horizontally, vertically, or diagonally adjacent full adders. Thus, it results in a very simple and efficient layout in VLSI implementation. However, the *N*-bit multiplication time is linearly proportional to *N*. This method requires a long computation time for larger *N*'s. Figure 1-9 [85] is a design of 5×5 unsigned multiplier:

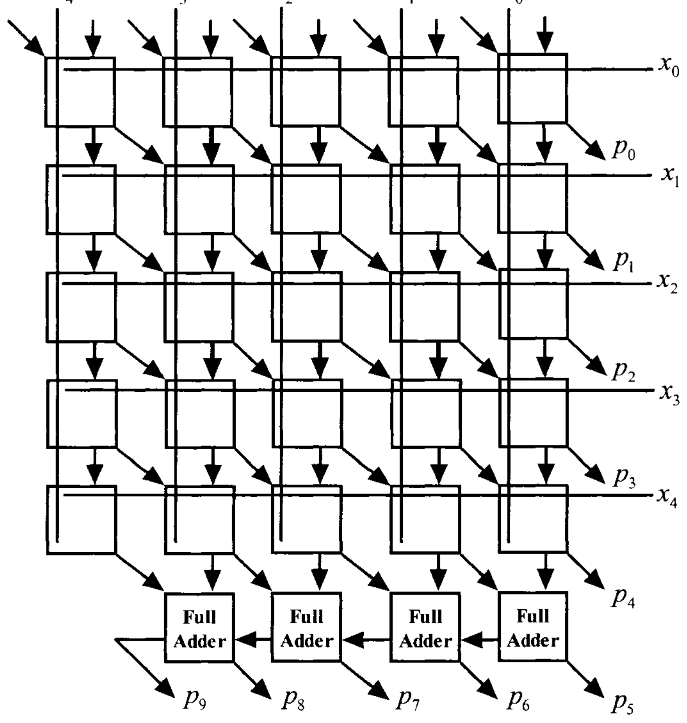


Figure 1-9. Design of a 5×5 Array Multiplier [85],[93]

The Wallace-tree method is commonly used to realize high-speed multiplication. The basic cell in Wallace-tree multiplication is 3-to-2 or 4-to-2 CSA (Carry Save Adder), also called 3:2 or 4:2 counter. A 3:2 counter can be realized by a full adder, which reduces three numbers to two numbers while a 4:2 counter can be realized by two 3:2 counters, as shown in Figure 1-10 [85]. Figure 1-11 and Figure 1-12 are 4×4 multipliers, using 3:2 counters and 4:2 counters.

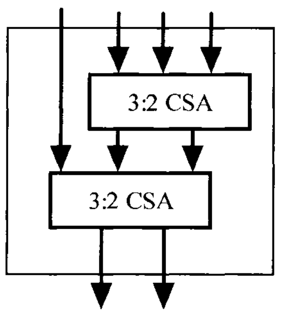


Figure 1-10. 4:2 Counter Using 3:2 Counter [94]

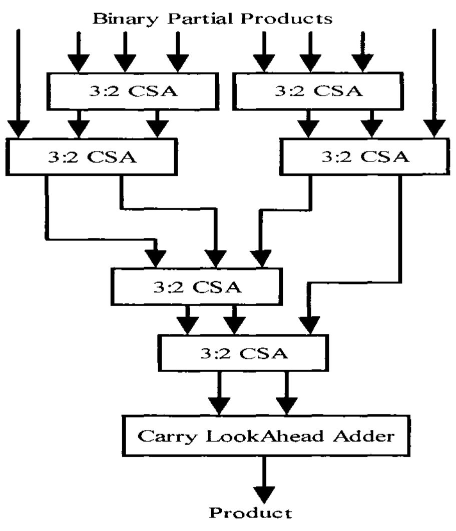


Figure 1-11. 3:2 Counter Based 4×4 Multiplier [85]

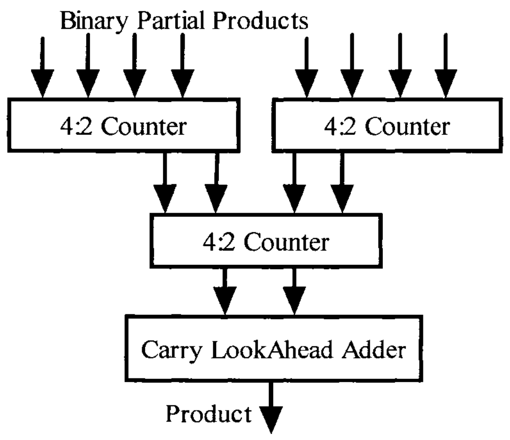


Figure 1-12. 4:2 Counter Based 4×4 Multiplier [85]

The traditional Wallace-tree method uses a 3:2 counter. This scheme results in a complicated interconnection between three-input/two-output counters. This makes the VLSI layout difficult and inefficient. The extended layout process increases the design complexity. As the multipliers increase in bit length, the interconnection becomes exponentially complicated. To solve this problem with conventional Wallace-trees, the following two methods have been proposed. One method is to use 4:2 counters [94] instead of 3:2 counters [94]. The use of 4:2 counters simplifies the interconnection drastically because the partial products are added using a binary tree. Another method is to use redundant binary representation for the partial products [1], [3], [4]. The use of the RB simplifies the interconnection because the RB partial products can be summed using an RB adder tree. The N-bit multiplication time of RB multipliers and Wallace-tree multipliers is proportional to $\log_2 N$. The physical layout of a RB multiplier has good repeatability. The RB multiplier does not require any optional sign bits for adding partial products. Makino's research [3] indicates that a 54×54-bit multiplier using redundant binary number representation is faster than the conventional 4:2 counter-based multiplier and has lower power dissipation. The power dissipation of 540 mW is estimated for the 54×54 RB multiplier operating at 100 MHz. These figures are more than 12% faster speed and 38% lower power than the conventional CSA multipliers.

Using redundant binary representation in our research results in an easily controlled/reconfigurable high-performance computing structure capable of handling various computations for both real and complex numbers.

1.5 The Redundant Binary Number System

Redundant binary (RB) representation is one of the signed-digit (SD) number systems originally introduced by Avizienis [69], which provides carry-propagation-free (CPF) addition. In a signed-digit system, the individual digits have negative as well as positive values. Given a radix-(r) signed-digit number, each digit of the signed-digit number can take one of the following $2\alpha + 1$ values:

$$\{-\alpha, ..., -1, 0, 1, ..., \alpha\} \qquad (1.2)$$

where the magnitude of a positive integer α must be within the following interval:

$$\left\lfloor \frac{r}{2} \right\rfloor \le \alpha \le r-1 \tag{1.3}$$

The radix-(2) signed-digit system (Redundant Binary (RB) representation) uses the digit set {-1, 0, 1} to represent numbers. The SD number system is also called redundant because a given integer number may have more than one representation. For example, the radix-(2) integer, $(7)_{10}$, can be represented in several ways, e.g., $[0\ 1\ 1\ 1]_{RB}$, $[1\ 0\ 0\ \text{-}1]_{RB}$, or $[1\ \text{-}1\ 1\ 1]_{RB}$. Based on the SD redundancy property, addition rules can be devised so that carry propagation is limited to only one digit position, thereby eliminating the possibility of a carry from the LSD (Least-Significant-Digit) to the MSD (Most-Significant-Digit). In a RB adder circuit implementation, the addition time is fixed and does not depend on the word length. Also, no explicit mechanism to handle the overall sign of a signed-digit number is required since it is determined by the most significant non-zero digit. Since the multiplication of two numbers is generally performed by the addition of partial products, the carry-propagation-free (CPF) feature of the RB arithmetic can be used to design high-speed multipliers [1], [3], [4] and multiply-and-accumulate (MAC) units [2].

The algorithmic rules for the RB addition are defined by Takagi, et al., [4]. Basically, two steps are required. In the first step, the intermediate carry-out, $\chi_i \in$ {-1, 0, 1}, and the intermediate sum digit, $\sigma_i \in$ {-1, 0, 1}, is generated at each position and satisfy the equation:

$$\alpha_i + \beta_i = 2\chi_i + \sigma_i \tag{1.4}$$

where α_i and β_i are the RB augend and addend digits, respectively. Note that for increased speed, the circuit implementation may utilize the next lower order digits,

α_{i-1} and β_{i-1}, to determine the carry-out from that digit position. Table 1-A describes these rules of step 1 in detail. In the second step, the final sum digit ζ_i is obtained at each position by adding the intermediate sum digit, σ_i, and the intermediate carry, χ_{i-1}, from the next-lower-order position, without generating a carry. That is,

$$\zeta_i = \sigma_i + \chi_{i-1} \tag{1.5}$$

Table 1-A. Computation Rules for the First Step in Carry-Propagation-Free Addition for RB Numbers [1]

Type	Augend digit α_i	Addend digit β_i	Digits at the next-lower-order position $(\alpha_{i-1}, \beta_{i-1})$	Intermediate carry (χ_i)	Intermediate sum digit (σ_i)
<1>	1	1	------------	1	0
<2>	1 0	0 1	Both are negative	0	1
			All other cases	1	-1
<3>	0	0	------------	0	0
<4>	1 -1	-1 1	------------	0	0
<5>	0 -1	-1 0	Both are negative	-1	1
			All other cases	0	-1
<6>	-1	-1	------------	-1	0

In general, throughout this dissertation, RB numbers are expressed using Greek symbols.

1.6 The Conversion of 2's-Complement to Redundant Binary

A limited precision RB number, Δ, can be derived from the addition of a pair of N-bit 2's-complement numbers A and B [2].

$$\begin{aligned}(A+B)_{2c} &= A-(-B)_{2c} \\ &= A-(\overline{B}+1) \\ &= A-\overline{B}-1 \\ &= \left(-a_{N-1}2^{N-1}+\sum_{i=0}^{N-2}a_i2^i\right)+\left(\overline{b_{N-1}}2^{N-1}-\sum_{i=0}^{N-2}\overline{b_i}2^i\right)-1 \\ &= (-a_{N-1}+\overline{b_{N-1}})2^{N-1}+\left(\sum_{i=0}^{N-2}(a_i-\overline{b_i})2^i\right)-1 \\ &= \left(\delta_{N-1}2^{N-1}+\sum_{i=0}^{N-2}\delta_i 2^i\right)-1 \\ &= \varDelta-1\end{aligned} \tag{1.6}$$

where $\delta_{N-1}=-a_{N-1}+\overline{b_{N-1}}$, $\delta_i=a_i-\overline{b}_i$ for $0\le i\le N-2$, 2c is the 2's-complement operations, $\overline{B}$ is the 1's-complement operations, $\overline{b_i}$ is the bit-complement, and –1 can be considered as a -1 carry-in to a subsequent RB addition. For inner-product calculations, the –1 correction is applied in the RB partial product adder tree.

The binary-signed digits can be encoded into binary in several ways. In this work, the binary signed digits{-1, 0, 0, 1} are coded as {00, 01, 10, 11}, respectively, as given in Table 1-B. Another encoding method is to encode redundant binary number in signed-magnitude [28], that is, to encode {-1, 0, 0, 1} as {11,00,10,01}. It is less efficient to map 2C to RB for signed-magnitude encoding. See Section 2.1.4 for further discussions.

Table 1-B. Coding Table for Binary Signed Digits

a value	Encoded a^-a^+
-1	0 0
0	0 1
0	1 0
1	1 1

Examining Equation (1.6), beginning with the δ_i term, the signed digits are encoded using the relationship, $\delta_i = a_i - \overline{b_i}$, where δ_i is a binary signed digit, $\delta_i \in$ {-1, 0, 1}. The mapping equations for δ_i^- and δ_i^+ are [2],[49],[50]

$$\begin{aligned} \delta_i^- &= a_i \\ \delta_i^+ &= b_i \quad \text{for } 0 \le i \le N-2 \end{aligned} \tag{1.7}$$

Similarly, in the Most Significant Digit (MSD) term of Equation (1.6), δ_{N-1} is encoded with the mapping equations

$$\begin{aligned} \delta_{N-1}^- &= \overline{a_{N-1}} \\ \delta_{N-1}^+ &= \overline{b_{N-1}} \end{aligned} \tag{1.8}$$

The structure of mapping the sum of two 2's-complement binary numbers to a RB number is shown in Figure 1-13:

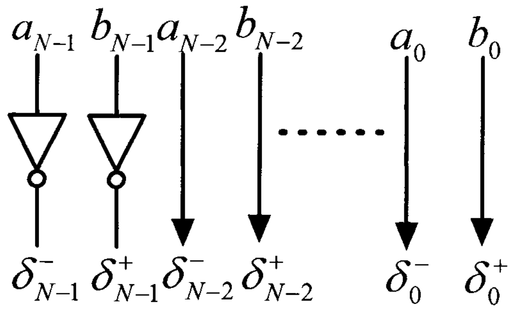


Figure 1-13. Mapping from the Sum of Two 2's-Complement Numbers to a RB Number

Note that a single 2's-complement number, *A*, is converted directly into a RB number *Δ* in Equation (1.9):

$$\begin{aligned} &\delta_{N-1}^+ = \overline{a_i},\ \delta_{N-1}^- = 0 \\ &\delta_{i-1}^+ = a_i,\ \ \delta_i^- = 1 \quad (0 \le i \le N-2) \end{aligned} \tag{1.9}$$

For example, a 2's-complement number $(00000101)_{2C}$ is converted directly into a RB number $(01\ 01\ 01\ 01\ 01\ 11\ 01\ 11)_{RB}$.

The subtraction of two N-bit 2's-complement numbers can also be represented by a redundant binary number:

$$\begin{aligned} E = (A-B)_{2c} &= \left(-a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i\right) + \left(b_{N-1}2^{N-1} - \sum_{i=0}^{N-2} b_i 2^i\right) \\ &= (-a_{N-1} + b_{N-1})2^{N-1} + \sum_{i=0}^{N-2}(a_i - b_i)2^i \qquad (1.10) \\ &= \varepsilon_{N-1}2^{N-1} + \sum_{i=0}^{N-2}\varepsilon_i 2^i \end{aligned}$$

where $\varepsilon_{N-1} = -a_{N-1} + b_{N-1}$ and $\varepsilon_i = a_i - b_i$ for $0 \le i \le N-2$.

The mapping equations for the encoded ε_i, $(0 \le i \le N-1)$ in Equation (1.10) are:

$$\begin{aligned} \varepsilon_i^- &= a_i \\ \varepsilon_i^+ &= \overline{b_i} \text{ for } 0 \le i \le N-2 \end{aligned} \qquad (1.11)$$

and

$$\begin{aligned} \varepsilon_{N-1}^- &= \overline{a_{N-1}} \\ \varepsilon_{N-1}^+ &= b_{N-1} \end{aligned} \qquad (1.12)$$

The structure of mapping the subtraction of two 2's-complement binary numbers to a RB number is shown in Figure 1-14.

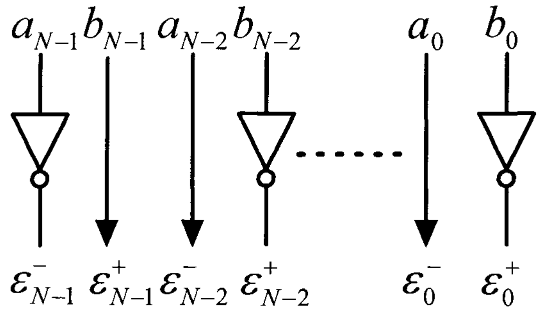


Figure 1-14. Mapping from the Subtraction of Two 2's-Complement Numbers to a RB Number

Based on this coding for RB numbers, the logic functions of a RBFA (RB full adder) and RBHA (RB half adder) are obtained [49] and shown in Table 1-C for the sum, z_i, with inputs x and y. Boolean variables, g and h, are used as intermediate variables to simplify the equations for the carry, c, and sum, z. Note, in Table 1-C, the notation used is the same as [49] but corrects the RBHA equations found in [49].

Table 1-C. Logic Functions for RBFA and RBHA

RBFA	RBHA
$g_i = (x_i^- \oplus x_i^+) \oplus (y_i^- \oplus y_i^+)$ $h_i = x_i^- x_i^+ + y_i^- y_i^+$ $z_i^- = g_i \oplus c_{i-1}^-$ $z_i^+ = c_{i-1}^+$ $c_i^- = (x_i^- + x_i^+)(y_i^- + y_i^+)$ $c_i^+ = g_i c_{i-1}^- + \overline{g_i} h_i$	$z_i^- = \overline{(x_i^- \oplus x_i^+)} \oplus y_i^-$ $z_i^+ = y_i^+$ $c_i^- = x_i^- + x_i^+$ $c_i^+ = \overline{(x_i^- \oplus x_i^+)} y_i^- + (x_i^- \oplus x_i^+) x_i^- x_i^+$

In this work, a novel, high-performance, fixed-point, inner-product processor based upon a redundant binary number system will be investigated. Similar to Baik's [28] methods, this scheme decreases the number of partial products by 50%, while achieving better speed and area performance and providing pipeline extension opportunities. When modified Booth encoding is used, partial products are reduced by almost 75%, thereby significantly reducing the multiplier addition depth. This design is well suited for VLSI implementation, and it can also be embedded as an inner-product core inside a general purpose DSP FPGA-based processor. This inner-product processor can be easily reconfigured for different computations, such as real number inner-product computations, parallel real number multipliers, complex number multipliers, complex number inner-product processors, redundant binary multipliers, redundant binary inner-product processors, etc. Chapter 2 proposes a fixed-point number inner-product processor. Computational structures for both real and complex number inner-products for both 2's-complement and unsigned integers is presented. A new division method using the IP structure is investigated in Chapter 3. Two convergence division methods — Goldschimdt and Newton-Raphson are compared. Chapter 4 discusses extended computations, such as parallel multiplications, inner-product processors using the inner-product processor for real, complex and redundant binary numbers. In Chapter 5, an improved redundant binary number to 2's-complement number converter is discussed. Chapter 6 provides a summary of contributions and future research directions for this work. The implementation of the redundant binary IP processor for real and complex number and the Goldschmidt division unit using the IP processor have been implemented using VHDL on Xilinx FPGA. The original contributions of this research are:

- IP processor reduces the number of partial products.
- A unified signed/unsigned 2's-complement/RB multiplier is developed using this IP structure.
- With the same IP structure, a novel Goldschmidt high-performance division circuit is developed.
- This IP structure can be used to build a multi-purpose dynamical processor for real, complex and redundant binary number computations.
- An improved 2's-complement to RB converter is proposed.

Chapter 2 Inner-Product Processor of Real, Complex and Redundant Binary Numbers

2.1 Real Number Inner-Product Computation

Consider an inner-product for an M dimensional (M=even), N-bit real vectors (N even), $\boldsymbol{A}$ and $\boldsymbol{B}$, where $\boldsymbol{A} = (A_0, A_1, A_2 .... A_{M-1})$ and $\boldsymbol{B} = (B_0, B_1, B_2 .... B_{M-1})$ with

$$A_i = (a_{N-1,i}\ a_{N-2,i}\ a_{N-3,i}\ ....a_{1,i}\ a_{0,i}) \text{ and } B_i = (b_{N-1,i}\ b_{N-2,i}\ b_{N-3,i}\ ....b_{1,i}\ b_{0,i}) \qquad (2.1)$$

where A_i and B_i are real numbers.

The real inner-product is defined as:

$$<\boldsymbol{A},\boldsymbol{B}> = (A_0\ A_1\ \cdots\ A_{M-2}\ A_{M-1}) \bullet (B_0\ B_1\ \cdots\ B_{M-2}\ B_{M-1}) = \sum_{i=0}^{M-1} A_i B_i \qquad (2.2)$$

Two basic approaches exist for performing the necessary inner-product multiplications using redundant binary arithmetic. The first method uses *inline* conversion or mapping of 2's-complement partial products into a redundant binary number for each multiplication of A_iB_i. The second method combines or maps equivalent 2's-complement partial products into a redundant binary number *across* the A_iB_i pairs [49]. Both approaches are considered in the following sections.

2.1.1 Inline Partial Product Redundant Binary Inner-Product

Considering the simple case of $M = 2$, $A_0B_0 + A_1B_1$, we first compute the redundant binary products for A_0B_0 and A_1B_1, and then add the RB products together to produce the inner-product. Redundant binary partial products are generated by mapping even/odd pairs of 2's-complement partial product sums. For N-bit numbers, the product of AB is expanded in the following equations:

$$\begin{aligned}
AB &= (-a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i)(-b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i) \\
&= (-a_{N-1}b_0 2^{N-1} + \sum_{i=0}^{N-2} a_i b_0 2^i)2^0 + (-a_{N-1}b_1 2^{N-1} + \sum_{i=0}^{N-2} a_i b_1 2^i)2^1 + \\
&\quad (-a_{N-1}b_2 2^{N-1} + \sum_{i=0}^{N-2} a_i b_2 2^i)2^2 + (-a_{N-1}b_3 2^{N-1} + \sum_{i=0}^{N-2} a_i b_3 2^i)2^3 + \\
&\quad \ldots\ldots\ldots \\
&\quad (-a_{N-1}b_{N-2} 2^{N-1} + \sum_{i=0}^{N-2} a_i b_{N-2} 2^i)2^{N-2} + (a_{N-1}b_{N-1} 2^{N-1} - \sum_{i=0}^{N-2} a_i b_{N-1} 2^i)2^{N-1}
\end{aligned}$$

Denote the 2's-complement partial products as:

$$\begin{aligned}
&PP_0 = (-a_{N-1}b_0 2^{N-1} + \sum_{i=0}^{N-2} a_i b_0 2^i)2^0,\ PP_1 = (-a_{N-1}b_1 2^{N-1} + \sum_{i=0}^{N-2} a_i b_1 2^i)2^1 \\
&PP_2 = (-a_{N-1}b_2 2^{N-1} + \sum_{i=0}^{N-2} a_i b_2 2^i)2^2,\ PP_3 = (-a_{N-1}b_3 2^{N-1} + \sum_{i=0}^{N-2} a_i b_3 2^i)2^3 \\
&\ldots\ldots \\
&PP_{N-2} = (-a_{N-1}b_{N-2} 2^{N-1} + \sum_{i=0}^{N-2} a_i b_{N-2} 2^i)2^{N-2},\ PP_{N-1} = (a_{N-1}b_{N-1} 2^{N-1} - \sum_{i=0}^{N-2} a_i b_{N-1} 2^i)2^{N-1}
\end{aligned} \tag{2.3}$$

Consider the first $N-2$ even/odd partial products, PP_{2j} and PP_{2j+1}, where j=0,1,2,…,$\frac{N-4}{2}$. To align the 2's-complement partial products, the sign of the even partial products is extended, and a low order zero is appended to the odd partial products:

$$\begin{aligned}
PP_{2j} &= (-a_{N-1}b_{2j}2^{N-1} + \sum_{i=0}^{N-2} a_i b_{2j} 2^i)2^{2j} \\
&= (-a_{N-1}b_{2j}2^N + a_{N-1}b_{2j}2^{N-1} + \sum_{i=0}^{N-2} a_i b_{2j} 2^i)2^{2j} \\
PP_{2j+1} &= (-a_{N-1}b_{2j+1}2^{N-1} + \sum_{i=0}^{N-2} a_i b_{2j+1} 2^i)2^{2j+1} \\
&= (-a_{N-1}b_{2j+1}2^N + a_{N-2}b_{2j+1}2^{N-1} + \sum_{i=1}^{N-2} a_{i-1} b_{2j+1} 2^i + 0)2^{2j}
\end{aligned} \tag{2.4}$$

Combining even/odd 2's-complement partial product pairs according to Equation (1.6), we have:

$$
\begin{aligned}
PP_{2j} + PP_{2j+1} &= \{(-a_{N-1}b_{2j} + \overline{a_{N-1}b_{2j+1}})2^N + \sum_{i=1}^{N-1}(a_i b_{2j} - \overline{a_{i-1}b_{2j+1}})2^i + a_0 b_{2j} - 1 - 1\}2^{2j} \\
&= (\alpha_{N,2j}2^N + \sum_{i=1}^{N-1}\alpha_{i,2j}2^i + \alpha_{0,2j} - 1)2^{2j}
\end{aligned}
\tag{2.5}
$$

where,

$$
\begin{aligned}
&\alpha_{N,2j} = -a_{N-1}b_{2j} + \overline{a_{N-1}b_{2j+1}} \\
&\alpha_{i,2j} = a_i b_{2j} - \overline{a_{i-1}b_{2j+1}}, \qquad 1 \le i \le N-1 \\
&\alpha_{0,2j} = a_0 b_{2j} - 1
\end{aligned}
\tag{2.6}
$$

Encoding the redundant binary coefficients, α_i, using two binary bits, all but the final RB partial product is encoded as:

$$
\begin{aligned}
&\text{For } j = 0,1,\ldots,\frac{N-4}{2}, \\
&\alpha^{+}_{N,2j} = \overline{a_{N-1}b_{2j}}, \quad \alpha^{-}_{N,2j} = \overline{a_{N-1}b_{2j+1}}, \\
&\alpha^{+}_{i,2j} = a_i b_{2j}, \quad \alpha^{-}_{i,2j} = a_{i-1}b_{2j+1}, \quad 1 \le i \le N-1 \\
&\alpha^{+}_{0,2j} = a_0 b_{2j}, \quad \alpha^{-}_{0,2j} = 0
\end{aligned}
\tag{2.7}
$$

Now, consider the last two 2's-complement partial products, PP_{N-2} and PP_{N-1}:

$$
\begin{aligned}
PP_{N-2}+PP_{N-1} &= (-a_{N-1}b_{N-2}2^N + a_{N-1}b_{N-2}2^{N-1} + \sum_{i=0}^{N-2} a_i b_{N-2}2^i)2^{N-2} \\
&\quad + (a_{N-1}b_{N-1}2^N - a_{N-2}b_{N-1}2^{N-1} - \sum_{i=1}^{N-2} a_{i-1}b_{N-1}2^i)2^{N-2} \\
&= \{(-a_{N-1}b_{N-2} + a_{N-1}b_{N-1})2^N + \sum_{i=1}^{N-1}(a_i b_{N-2} - a_{i-1}b_{N-1})2^i + a_0 b_{N-2} - 0\}2^{N-2} \\
&= \beta_N + \sum_{i=1}^{N-1}\beta_i 2^i + \beta_0
\end{aligned}
\tag{2.8}
$$

where,

$$
\beta_N = -a_{N-1}b_{N-2} + a_{N-1}b_{N-1},\ \beta_i = a_i b_{N-2} - a_{i-1}b_{N-1},\ (1 \le i \le N-1), \beta_0 = a_0 b_{N-2} - 0 \tag{2.9}
$$

Encoding the redundant binary coefficients, β_i, using two binary bits, the final RB partial product is:

$$\begin{aligned} &\beta_N^+ = \overline{a_{N-1}b_{N-2}}, && \beta_N^- = a_{N-1}b_{N-1}, && \\ &\beta_i^+ = a_i b_{N-2}, && \beta_i^- = \overline{a_{i-1}b_{N-1}}, && 1 \le i \le N-1 \\ &\beta_0^+ = a_0 b_{N-2}\,, && \beta_0^- = 1 && \end{aligned} \tag{2.10}$$

Figure 2-1 shows the RB implementation diagram of $A_0B_0 + A_1B_1$ for 8-bit numbers and Figure 2-2 shows the hardware implementation of A_0B_0+A_1B_1 with the inline redundant binary partial product generation (RBPPG) using 2's-complement even/odd partial products. If the final redundant binary adder (RBA) is bypassed, the circuit in Figure 2-2 can also perform the separate multiplications, A_0B_0 and A_1B_1.

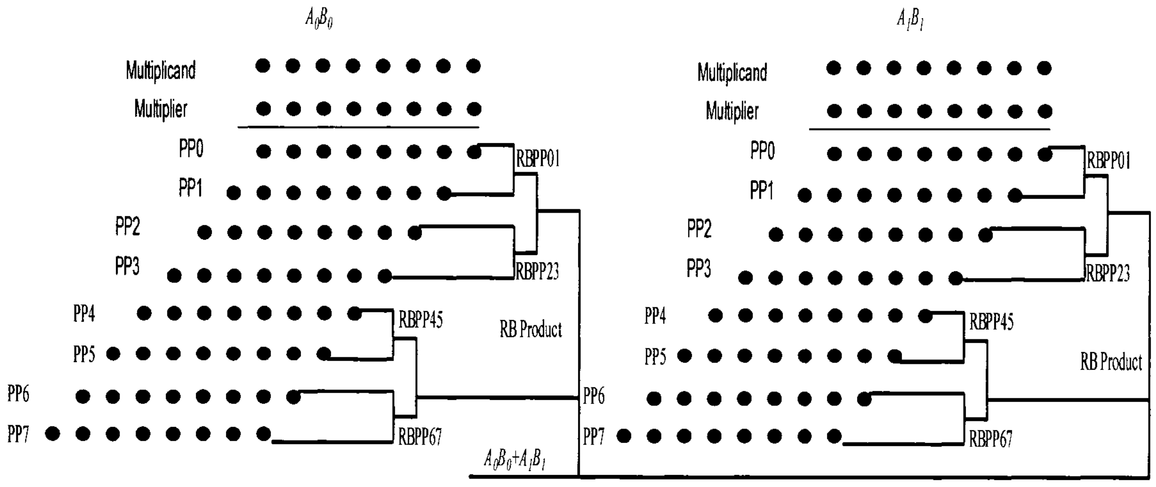
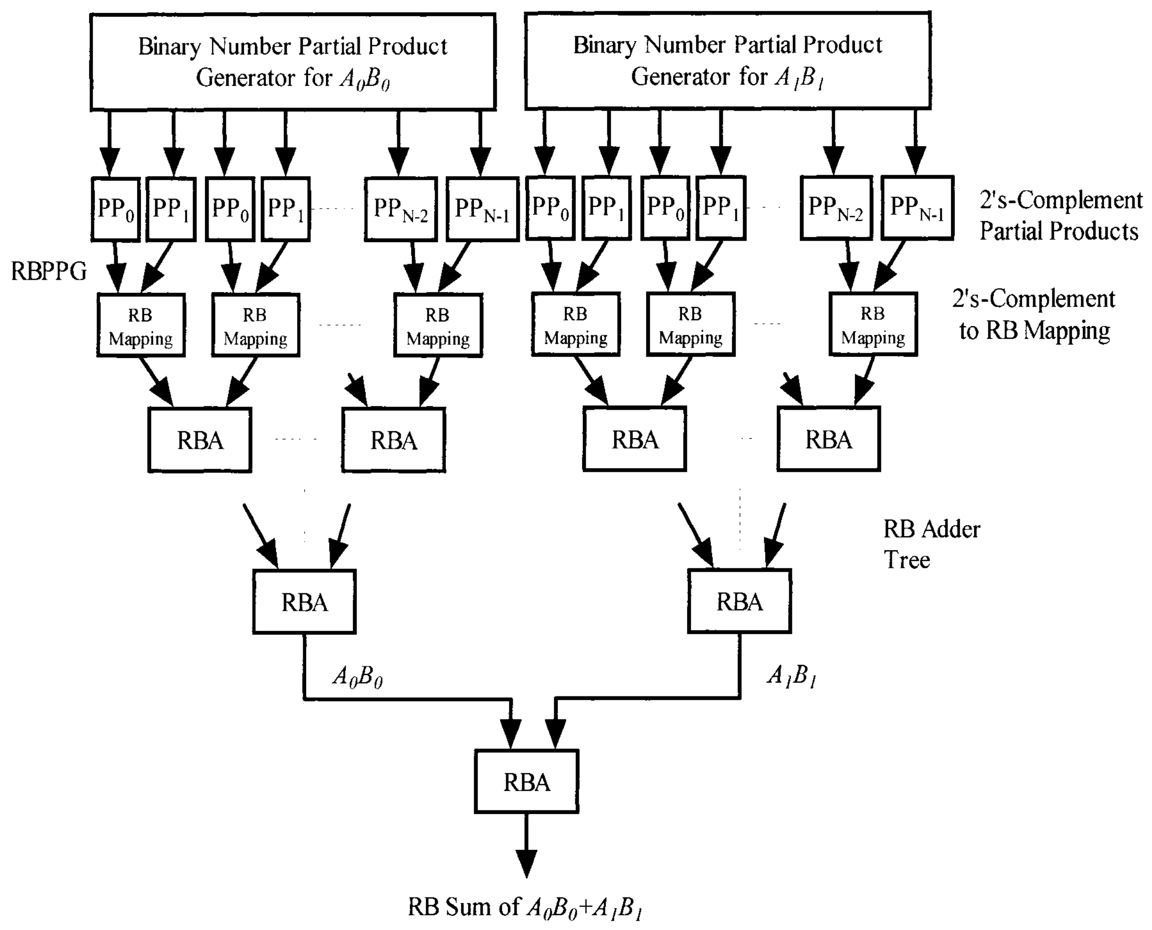


Figure 2-1. Inline Partial Product RB Implementation of $A_0B_0 + A_1B_1$

Defining the redundant binary number $\Delta_j = A_{2j}B_{2j} + A_{2j+1}B_{2j+1}$, the general form of the inline multiplication inner-product is given by:

$$<\boldsymbol{A},\boldsymbol{B}> = \sum_{j=0}^{M-1} A_i B_i = \sum_{j=0}^{\frac{M}{2}-1} (A_{2j}B_{2j} + A_{2j+1}B_{2j+1}) = \sum_{j=0}^{\frac{M}{2}-1} \Delta_j \tag{2.11}$$



Binary Number Partial Product Generator for A_0B_0
Binary Number Partial Product Generator for A_1B_1
PP_0
PP_1
PP_{N-2}
PP_{N-1}
2's-Complement Partial Products
RBPPG
RB Mapping
2's-Complement to RB Mapping
RBA
RB Adder Tree
A_0B_0
A_1B_1
RB Sum of $A_0B_0+A_1B_1$

Figure 2-2. Inline Partial Product Structure of $A_0B_0 + A_1B_1$

To realize the inner-product, all of the redundant binary numbers, Δ_j, are added using a redundant binary adder tree, and the final sum of the redundant binary numbers can be converted into a 2's-complement number using a RB-NB converter [5],[7],[12]. Figure 2-3 depicts the overall architecture of the RB inner-product circuit showing the RB adder tree, a RB accumulator, a RB-to-2's-complement converter, and additional data paths that might exist when the circuit is implemented in a DSP core.

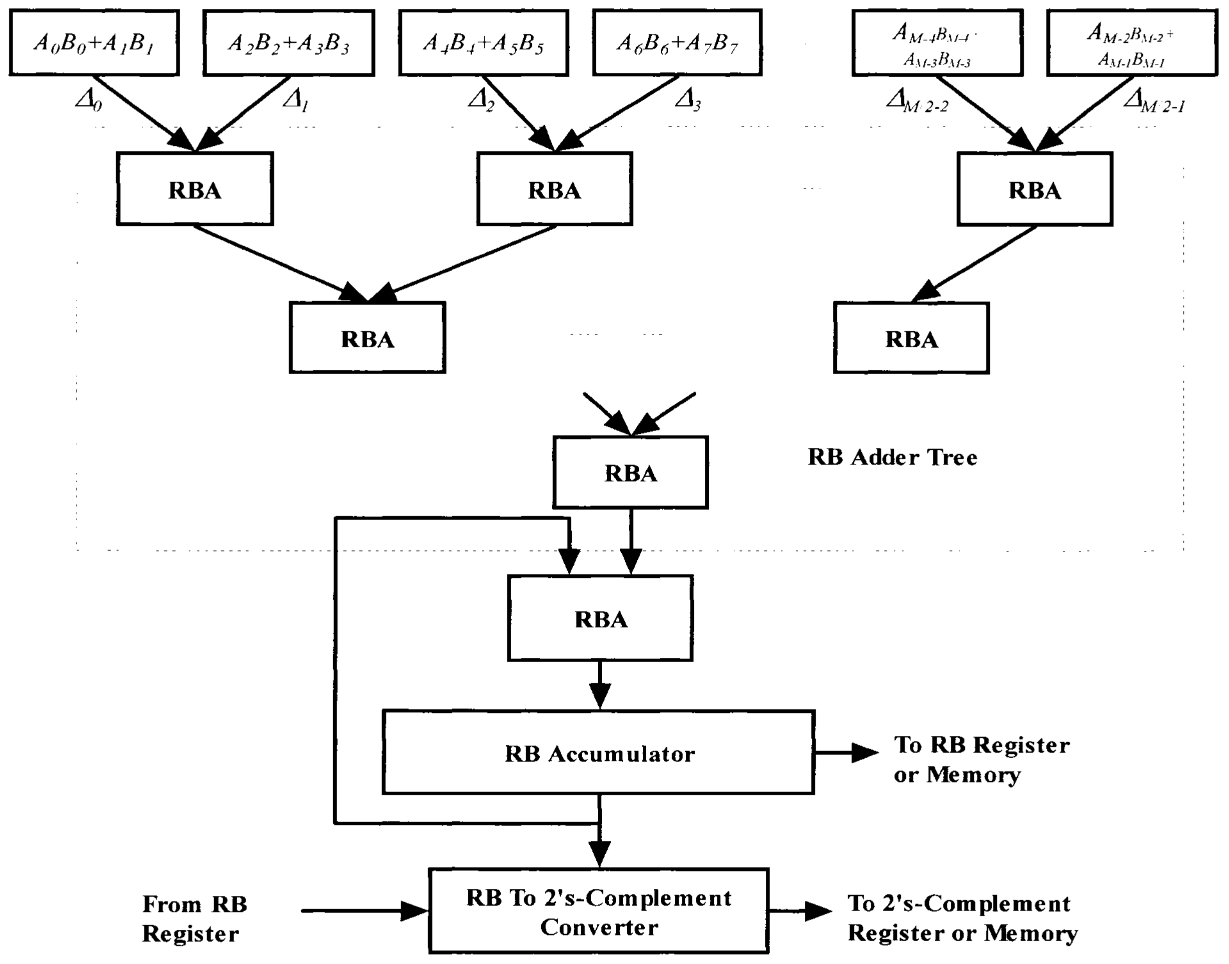


Figure 2-3. Overall Structure of the Redundant Binary Inner-Product

2.1.2 Cross Partial Product Redundant Binary Inner-Product

An alternative method for mapping 2's-complement partial products to redundant binary partial products is to combine like partial products across the A_jB_j and $A_{j+1}B_{j+1}$ pairs. The method derived here is similar to that provided by Shin and Jeon [50] for complex number multiplication. Consider the simple case of M=2, $A_0B_0+A_1B_1$, and expand it as:

$$
\begin{aligned}
&A_0B_0 + A_1B_1 \\
&= (-a_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2} a_{i,0}2^i)(-b_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2} b_{i,0}2^i) + \\
&\qquad (-a_{N-1,1}2^{N-1} + \sum_{i=0}^{N-2} a_{i,1}2^i)(-b_{N-1,1}2^{N-1} + \sum_{i=0}^{N-2} b_{i,1}2^i) \\
&= 2^{N-1}\{a_{N-1,0}b_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0})2^i + a_{N-1,1}b_{N-1,1}2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,1}b_{N-1,1})2^i\} + \\
&\sum_{i=0}^{N-2} 2^i\{b_{i,0}(-a_{N-1,0}2^{N-1} + \sum_{j=0}^{N-2} a_{j,0}2^j) + b_{i,1}(-a_{N-1,1}2^{N-1} + \sum_{j=0}^{N-2} a_{j,1}2^j)\} \\
&= 2^{N-1}\{(a_{N-1,0}b_{N-1,0} + a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0} - a_{i,1}b_{N-1,1})2^i\} + \\
&\sum_{i=0}^{N-2} 2^i\{(-a_{N-1,0}b_{i,0} - a_{N-1,1}b_{i,1})2^{N-1} + \sum_{j=0}^{N-2}(a_{j,0}b_{i,0} + a_{j,1}b_{i,1})2^j\}
\end{aligned}
\tag{2.12}
$$

Consider the first term of Equation (2.12),

$$
\begin{aligned}
&2^{N-1}\{(a_{N-1,0}b_{N-1,0} + a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0} - a_{i,1}b_{N-1,1})2^i\} \\
&= -2^{N-1}\{(-a_{N-1,0}b_{N-1,0})2^{N-1} + \sum_{i=0}^{N-2}(a_{i,0}b_{N-1,0})2^i \\
&\qquad + (-a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(a_{i,1}b_{N-1,1})2^i\} \\
&= 2^{N-1}\{(a_{N-1,0}b_{N-1,0} - \overline{a_{N-1,1}b_{N-1,1}})2^{N-1} \\
&\qquad + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0} + \overline{a_{i,1}b_{N-1,1}})2^i + 1\} \\
&= 2^{N-1}\{\kappa_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}\kappa_{i,0}2^i + 1\}
\end{aligned}
\tag{2.13}
$$

where $\kappa_{i,0}$ is a redundant binary number and ,

$\kappa_{N-1,0} = a_{N-1,0}b_{N-1,0} - \overline{a_{N-1,1}b_{N-1,1}}$, and $\kappa_{i,0} = -a_{i,0}b_{N-1,0} + \overline{a_{i,1}b_{N-1,1}}$ for $0 \le i \le N-2$ (2.14)

Encoding the redundant binary numbers, $\kappa_{N-1,0}$ and $\kappa_{i,0}$, the Boolean equations are

$$\kappa^-_{N-1,0} = a_{N-1,0}b_{N-1,0},\ \ \kappa^+_{N-1,0} = a_{N-1,1}b_{N-1,1},$$
$$\kappa^-_{i,0} = \overline{a_{i,0}b_{N-1,0}}\ ,\ \text{and}\ \kappa^+_{i,0} = \overline{a_{i,1}b_{N-1,1}} \tag{2.15}$$

Considering the second term of Equation (2.12), and using Equation (1.6),

$$\begin{aligned}
&\sum_{i=0}^{N-2} 2^i\{(-a_{N-1,0}b_{i,0} - a_{N-1,1}b_{i,1})2^{N-1} + \sum_{j=0}^{N-2}(a_{j,0}b_{i,0} + a_{j,1}b_{i,1})2^j\} \\
&= \sum_{i=0}^{N-2} 2^i\{-a_{N-1,0}b_{i,0} + \sum_{j=0}^{N-2}(a_{j,0}b_{i,0})2^j - (a_{N-1,1}b_{i,1})2^{N-1} + \sum_{j=0}^{N-2}(a_{j,1}b_{i,1})2^j\} \\
&= \sum_{i=0}^{N-2} 2^i\{(-a_{N-1,0}b_{i,0} + \overline{a_{N-1,1}b_{i,1}})2^{N-1} + \sum_{j=0}^{N-2}(a_{j,0}b_{i,0} - \overline{a_{j,1}b_{i,1}})2^j - 1\} \\
&= \sum_{i=0}^{N-2} 2^i\{\lambda_{i,N-1}2^{N-1} + \sum_{j=0}^{N-2}\lambda_{i,j}2^j - 1\}
\end{aligned} \tag{2.16}$$

where $\lambda_{i,j}$ $(0 \le i \le N-2,\ 0 \le j \le N-1)$ is a redundant binary number with

$$\lambda_{i,j} = a_{j,0}b_{i,0} - \overline{a_{j,1}b_{i,1}} \text{ for } 0 \le j \le N-2, \text{ and } \lambda_{i,N-1} = -a_{N-1,0}b_{i,0} + \overline{a_{N-1,1}b_{i,1}} \text{ for } j = N-1 . \tag{2.17}$$

Encoding $\lambda_{i,j}$ as two binary bits, $\lambda^-_{i,j}$ and $\lambda^+_{i,j}$, $(0 \le i \le N-2,\ 0 \le j \le N-1)$,

$$\lambda^-_{i,N-1} = \overline{a_{N-1,0}b_{i,0}},\ \ \lambda^+_{i,N-1} = \overline{a_{N-1,1}b_{i,1}},$$
$$\lambda^-_{i,j} = a_{j,0}b_{i,0}\ ,\ \text{and}\ \lambda^+_{i,j} = a_{j,1}b_{i,1}. \tag{2.18}$$

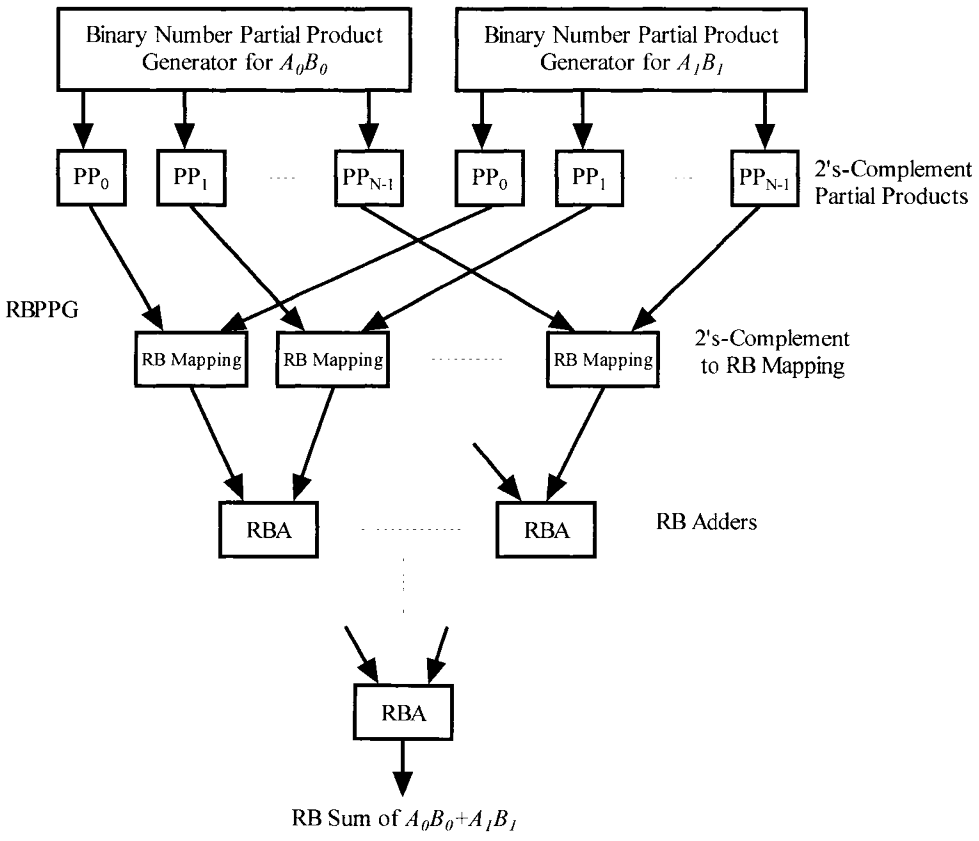
The overall inner-product is expressed as:

$$\begin{aligned}
&A_0B_0 + A_1B_1 \\
&= 2^{N-1}\{(a_{N-1,0}b_{N-1,0} + a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0} - a_{i,1}b_{N-1,1})2^i\} + \\
&\quad \sum_{i=0}^{N-2} 2^i\{(-a_{N-1,0}b_{i,0} - a_{N-1,1}b_{i,1})2^{N-1} + \sum_{j=0}^{N-2}(a_{j,0}b_{i,0} + a_{j,1}b_{i,1})2^j\} \\
&= 2^{N-1}\{\kappa_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}\kappa_{i,0}2^i + 1\} + \sum_{i=0}^{N-2}2^i\{\lambda_{i,N-1}2^{N-1} + \sum_{j=0}^{N-2}\lambda_{i,j}2^j - 1\} \\
&= 2^{N-1}\{\kappa_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}\kappa_{i,0}2^i\} + \sum_{i=0}^{N-2}2^i\{\lambda_{i,N-1}2^{N-1} + \sum_{j=0}^{N-2}\lambda_{i,j}2^j\} \\
&\quad + 2^{N-1} - \sum_{i=0}^{N-2}2^i
\end{aligned} \tag{2.19}$$

Since $2^{N-1} - \sum_{i=0}^{N-2} 2^i = 1$, Equation (2.19) becomes:

$$A_0B_0 + A_1B_1 = 2^{N-1}\{\kappa_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}\kappa_{i,0}2^i\} + \sum_{i=0}^{N-2}2^i\{\lambda_{i,N-1}2^{N-1} + \sum_{j=0}^{N-2}\lambda_{i,j}2^j\} + 1 \qquad (2.20)$$

The adjusting term, +1, can be applied as a carry-in to the LSD of the redundant binary full adder. Figure 2-4 shows the schematic structure of the $A_0B_0 + A_1B_1$ hardware implementation.



Binary Number Partial Product Generator for A_0B_0
Binary Number Partial Product Generator for A_1B_1
PP_0
PP_1
PP_{N-1}
PP_0
PP_1
PP_{N-1}
2's-Complement Partial Products
RBPPG
RB Mapping
RB Mapping
RB Mapping
2's-Complement to RB Mapping
RBA
RBA
RB Adders
RBA
RB Sum of $A_0B_0+A_1B_1$

Figure 2-4. Cross Partial Product Structure of $A_0B_0 + A_1B_1$ [50]

Defining the redundant binary number $\Phi_j = A_{2j}B_{2j} + A_{2j+1}B_{2j+1}$, the general form of the cross partial product method inner-product is given by:

$$\langle \boldsymbol{A},\boldsymbol{B}\rangle = \sum_{j=0}^{M-1} A_i B_i = \sum_{j=0}^{\frac{M}{2}-1} (A_{2j}B_{2j} + A_{2j+1}B_{2j+1}) = \sum_{j=0}^{\frac{M}{2}-1} \Phi_j \tag{2.21}$$

Again, all of the redundant binary numbers, Φ_j, are added using a redundant binary adder tree, and the final sum of the redundant numbers can be converted into a 2's-complement number using a RB-NB converter [5],[7],[12]. The same redundant binary adder tree used for the inline inner-product, shown in Figure 2-4, can be applied to the cross partial product method.

2.1.3 Booth Encoding Methods

To further reduce the number of partial products, the modified Booth encoding technique is used [96]. The modified Booth algorithm recodes an N-bit 2's-complement number, B, by the following equation:

$$\begin{aligned} B &= -b_{N-1}2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i = \sum_{i=0}^{\frac{N}{2}-1} (-2b_{2i+1} + b_{2i} + b_{2i-1})2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} Q_i 2^{2i} \end{aligned} \tag{2.22}$$

where $b_{-1} = 0$ and $Q_i \in \{-2,-1,0,+1,+2\}$ is determined according to the bit pattern of the 3-bit string of B as given in Table 2-A.

Table 2-A. Modified Booth Encoding Table [96]

b_{i+1}	b_i	b_{i-1}	Q_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

For the inline partial product method, we apply Booth coding to find the product of A and B (assuming N is even), using Equation (2.22),

$$\begin{aligned} AB &= (-a_{N-1}2^{N-1} + \sum_{k=0}^{N-2} a_k 2^k) \sum_{i=0}^{\frac{N}{2}-1} Q_i 2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} Q_i(-a_{N-1}2^{N-1} + \sum_{k=0}^{N-2} a_k 2^k) 2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} (C_i + g_i) 2^{2i} \end{aligned} \tag{2.23}$$

$$\begin{aligned} &\text{If } Q_i = 1,\ C_i = -a_{N-1}2^{N-1} + \sum_{k=0}^{N-2} a_k 2^k \text{ and } g_i = 0 \\ &\text{If } Q_i = 2,\ C_i = -a_{N-1}2^{N} + \sum_{k=0}^{N-2} a_k 2^{k+1} \text{ and } g_i = 0 \\ &\text{If } Q_i = -1,\ C_i = -\overline{a_{N-1}}2^{N-1} + \sum_{k=0}^{N-2} \overline{a_k} 2^k \text{ and } g_i = 1 \\ &\text{If } Q_i = -2,\ C_i = -\overline{a_{N-1}}2^{N} + \sum_{k=0}^{N-2} \overline{a_k} 2^{k+1} \text{ and } g_i = 1 \end{aligned} \tag{2.24}$$

Notice that C_i is a 2's-complement number. Mapping the product to redundant binary,

$$\begin{aligned} AB &= \sum_{i=0}^{\frac{N}{2}-1} (C_i + g_i) 2^{2i} \\ &= \sum_{i=0,2,4,\ldots\frac{N}{2}-2} \{(C_i + g_i)2^{2i} + (C_{i+1} + g_{i+1})2^{2i+2}\} \\ &= \sum_{i=0,2,4,\ldots\frac{N}{2}-2} \{(C_i + 4C_{i+1})2^{2i} + (g_i + 4g_{i+1})2^{2i}\} \\ &= \sum_{i=0,2,4,\ldots\frac{N}{2}-2} \{E_i 2^{2i} + (g_i + 4g_{i+1} - 1)2^{2i}\} \\ &= \sum_{i=0,2,4,\ldots\frac{N}{2}-2} (E_{i,i+1} + \gamma_i + \gamma_{i+1}) 2^{2i} \end{aligned} \tag{2.25}$$

where E_i is a redundant binary number and $E_i - 1 = C_i + 4C_{i+1}$. The correction factors, γ_i and γ_{i+1}, depend on the values of g_i and g_{i+1}, as shown in Table 2-B. Here, the two redundant binary Booth correction factors, γ_i and γ_{i+1}, are used since g_i and g_{i+1} can not be combined. The -1 is encoded into the γ_i correction factor.

Table 2-B. Booth Correction Factors for the Inline Multiplication Method

g_i	g_{i+1}	γ_i $(\gamma_i^+\gamma_i^-)$	γ_{i+1} $(\gamma_{i+1}^+\gamma_{i+1}^-)$
0	0	-1 (0 0)	0 (0 1)
0	1	-1 (0 0)	1 (1 1)
1	0	0 (0 1)	0 (0 1)
1	1	0 (0 1)	1 (1 1)

Referring to Booth coding Table 2-A, the Boolean equations for the correction factors are,

$$\begin{aligned}
\gamma_i^+ &= 0 \\
\gamma_i^- &= b_{i+1}\overline{b_i}\,\overline{b_{i-1}} + b_{i+1}\overline{b_i}\,b_{i-1} + b_{i+1}b_i\,\overline{b_{i-1}} \\
&= b_{i+1}\,\overline{b_i b_{i-1}} \\
\gamma_{i+1}^+ &= b_{i+3}\,\overline{b_{i+2}b_{i+1}} \\
\gamma_{i+1}^- &= 1
\end{aligned} \tag{2.26}$$

where $i = 0, 2, 4, \ldots \frac{N}{2} - 2$. After the redundant binary products of A_0B_0 and A_1B_1 are computed, a redundant binary adder is used to compute $A_0B_0 + A_1B_1$.

The general form of the inline Booth encoded inner-product is,

$$<\boldsymbol{A},\boldsymbol{B}> = \sum_{j=0}^{M-1} A_jB_j = \sum_{j=0}^{M-1} \sum_{i=0,2,4,\ldots\frac{N}{2}-2} (E_{j,(i,i+1)} + \gamma_{j,i} + \gamma_{j,i+1})2^{2i} \tag{2.27}$$

where the number of partial products is decreased to slightly above 25%, with consideration being given to the correction factors.

Applying Booth encoding for the cross partial product method, we must consider $A_0B_0 + A_1B_1$. Using Equation (2.22),

$$\begin{aligned} A_0B_0 + A_1B_1 &= \sum_{i=0}^{\frac{N}{2}-1} Q_{i,0}(-a_{N-1,0}2^{N-1} + \sum_{k=0}^{N-2} a_{k,0}2^k)2^{2i} \\ &+\sum_{i=0}^{\frac{N}{2}-1} Q_{i,1}(-a_{N-1,1}2^{N-1} + \sum_{k=0}^{N-2} a_{k,1}2^k)2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} (C_{i,0} + g_{i,0})2^{2i} + \sum_{i=0}^{\frac{N}{2}-1} (C_{i,1} + g_{i,1})2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} (C_{i,0} + C_{i,1} + g_{i,1} + g_{i,0})2^{2i} \end{aligned} \tag{2.28}$$

From Equation (1.6), the sum of two 2's-complement numbers can be considered as a redundant binary number minus 1. Equation (2.28) can be converted to:

$$\begin{aligned} A_0B_0 + A_1B_1 &= \sum_{i=0}^{\frac{N}{2}-1} (C_{i,0} + C_{i,1} + g_{i,1} + g_{i,0})2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} (\Gamma_{i,01} + g_{i,1} + g_{i,0} - 1)2^{2i} \\ &= \sum_{i=0}^{\frac{N}{2}-1} (\Gamma_{i,01} + \gamma_{i,01})2^{2i} \end{aligned} \tag{2.29}$$

where $\Gamma_{i,01}$ is a redundant binary number from the addition of $C_{i,0}$ and $C_{i,1}$, and the redundant binary number $\gamma_{i,01} = g_{i,0} + g_{i,1} - 1$. The correction factor, $\gamma_{j,01}$, depends on the values of $g_{i,0}$ and $g_{i,1}$, as shown in Table 2-C.

Table 2-C. Booth Correction Factors for the Cross Partial Product Method

$g_{i,0}$	$g_{i,1}$	$\gamma_{i,01} = g_{i,0} + g_{i,1} - 1$	$\gamma^{-}_{i,01}\ \gamma^{+}_{i,01}$
0	0	-1	0 0
0	1	0	0 1
1	0	0	1 0
1	1	1	1 1

From the Booth coding Table 2-A, the bit encoding Boolean equations are:

$$\begin{aligned} \gamma_{i,01}^{-} &= g_{i,0} = b_{i+1,0}\overline{b_{i,0}}\ \overline{b_{i-1,0}} + b_{i+1,0}\overline{b_{i,0}}\ b_{i-1,0} + b_{i+1,0}b_{i,0}\ \overline{b_{i-1,0}} \\ &= b_{i+1,0}\ \overline{b_{i,0}b_{i-1,0}} \\ \gamma_{i,01}^{+} &= g_{i,1} = b_{i+1,1}\overline{b_{i,1}}\ \overline{b_{i-1,1}} + b_{i+1,1}\overline{b_{i,1}}\ b_{i-1,1} + b_{i+1,1}b_{i,1}\ \overline{b_{i-1,1}} \\ &= b_{i+1,1}\ \overline{b_{i,1}b_{i-1,1}} \end{aligned} \tag{2.30}$$

Using Booth coding, the cross partial product inner-product method is given by

$$<\boldsymbol{A},\boldsymbol{B}> = \sum_{j=0}^{M-1} A_j B_j = \sum_{j=0}^{\frac{M}{2}-1} (A_{2j}B_{2j} + A_{2j+1}B_{2j+1}) = \sum_{j=0}^{\frac{M}{2}-1} \sum_{i=0}^{\frac{N}{2}-1} (\Gamma_{i,(2j,2j+1)} + \gamma_{i,(2j,2j+1)})2^{2i} \tag{2.31}$$

Again, the number of partial products is decreased to slightly above 25%, with consideration being given to the correction factors.

2.1.4 Implementation Comparison of Inline and Cross Inner-Product Methods

An 8-tap digital filter implementation in [28] uses a signed-amplitude system to encode a redundant binary number. The signed-amplitude method requires two gate delays for the conversion from 2's-complement to redundant binary. Examining Equations (1.7) and (1.8), only inverters are necessary for the inline partial product redundant binary mapping.

In [49], the cross partial product implementations of $A_0B_0 + A_1B_1$ and $A_0B_0 - A_1B_1$ are discussed for complex number multiplication. An equivalent derivation was provided in Section 2.1.2. The inline method, presented in Section 2.1.1, combines partial products *within* the partial products of A_0B_0 and A_1B_1, respectively. Figure 2-2 and Figure 2-4 depict these methods. For a qualitative comparison of these two designs, note that as feature sizes shrink in deep submicron VLSI technology, interconnection wires contribute a large portion of the total delay [49],[50]. The inline implementation provides more direct routing for vertical (horizontal) wires, while the cross partial product method [49],[50] will need crossing horizontal (vertical) wiring paths for partial product mapping, with the routing distance proportional to the word width. Therefore, the inline partial product method will result in improved performance, compared to the cross partial product method. In addition, the inline method offers more extended operational capability than the cross partial product scheme (see Chapter 4). The inline method requires more horizontal gates, primarily due to the overhead of the partial product alignment. Table 2-D and Table 2-E show comparisons for Xilinx FPGA implementations for $A_0B_0 + A_1B_1$, with and without Booth encoding. The Xilinx Virtex2 2V6000FF1517 device was targeted for the implementations using VHDL and Xilinx Foundation software. For this word length, the higher performance and area savings of the Booth encoded designs are evident (See Appendix for VHDL code availability).

Table 2-D. 16-Bit FPGA Implementations of $A_0B_0 + A_1B_1$ Without Booth Encoding

	Cross PP Method	Inline PP Method
Number of Slices	1094	1094
Number of LUTs	1992	2002
Equivalent Gate Count	11952	12012
Maximum Delay	30.014ns	28.084ns

Table 2-E. 16-Bit FPGA Implementations of $A_0B_0 + A_1B_1$ With Booth Encoding

	Cross PP Method	Inline PP Method
Number of Slices	846	905
Number of LUTs	1603	1717
Equivalent Gate Count	9618	10302
Maximum Delay	29.460ns	27.970ns

2.2 Complex Number Inner-Product Computation

2.2.1 Review of Complex Number Arithmetic

Complex number arithmetic computation is a key arithmetic feature in modern digital communication, radar systems and optical systems. Many algorithms based on convolutions, correlations, and complex filters require complex number multiplication, complex number division, and high-speed inner-products. These applications require efficient representation and manipulation of complex numbers together with real numbers. Among these computations, high-performance complex number multipliers and complex number inner-products are desirable in modern digital communication, optical systems, and radar systems. Recent research in hardware implementation of complex number arithmetic circuits is focused on utilization of radix-(2), as well as alternative radices, for the representation of complex numbers.

In this chapter, different complex radices are investigated and compared. It is found that the complex radices have no advantage in hardware implementations. Traditional radix-(2) redundant binary numbers are used to implement complex-number multiplication and inner-product processing. The investigated inline inner-product processor can be reconfigured/controlled to perform complex-number computations. The computational structures of $A_0B_0 + A_1B_1$ and $A_0B_0 - A_1B_1$ are developed for performing complex number inner-products. The implementation of $A_0B_0 + A_1B_1$ can be easily controlled to perform the computation of $A_0B_0 - A_1B_1$. A complex number inner-product processor is realized, based upon a unified structure for $A_0B_0 \pm A_1B_1$.

To represent a complex number other than radix-(2), several representations have been proposed. Knuth [63],[64] described a "quater-imaginary" number system with radix-(2*j*). Dao [58] further analyzed this quater-imaginary system for complex-radix arithmetic. Penney [66] proposed a complex number representation with the base of *j*-1. Slekys [67] defined arithmetic operations on radix- $(j\sqrt{2})$. Recently, further investigations examined the arithmetic algorithms and hardware implementation of these representations. Aoki [31],[97],[98], and McIlhenny [99],[100] investigated complex number arithmetic in a redundant radix-(*2j*) number system. Jamil [62] and Blest [55] further analyzed the complex number computations in the radix-($j-1$) number system and included proposed arithmetic methods for addition, subtraction, multiplication and division. Frougny [60] and Koren [65] provided a theoretic investigation of complex number arithmetic for complex numbers in the bases $j\sqrt{b}$ and $-b+j$. Stepanenko [68] also investigated the complex number arithmetic in radix-($j\sqrt{2}$).

Multiplication is an essential operation for high-speed hardware implementation of complex number computations. It can be used to compare the complexity of complex number arithmetic with different complex radices. The analysis of complex number multiplication in these various radices will provide one metric for comparison.

To compute the product of two complex numbers, the conventional method is to use four binary multiplications, one addition, and one subtraction, as shown in Figure 2-5. Define two complex numbers as:

$$\begin{aligned} A &= A_r + jA_i \\ B &= B_r + jB_i \end{aligned} \tag{2.32}$$

where $j=\sqrt{-1}$, and A_r, A_i, B_r, and B_i are the real and imaginary parts of the complex numbers, A and B. Multiplication of A and B is given by:

$$A \times B = (A_r + jA_i) \times (B_r + jB_i) = A_rB_r - A_iB_i + j(A_rB_i + A_iB_r) \tag{2.33}$$

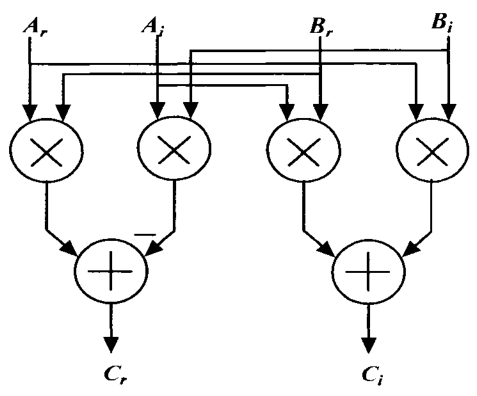


Figure 2-5. Basic Diagram of Complex Number Multiplication

In this direct implementation method, four multiplications plus two additions are required. To reduce the arithmetic complexity of the complex number multiplication, an algebraic transform, given in Equation (2.34), is proposed by Blahut [101]. This method saves one real number multiplication at the expense of three more additions:

$$\begin{aligned}&(A_r + jA_i)\times(B_r + jB_i)\\ &= \{(A_r - A_i)B_i + A_r(B_r - B_i)\} + j\{(A_r - A_i)B_i + A_i(B_r + B_i)\}\end{aligned} \tag{2.34}$$

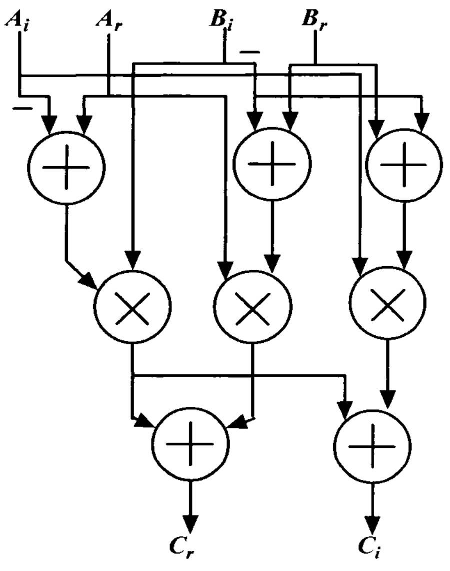


Figure 2-6. Blahut's Complex Number Multiplier [101]

As shown in Figure 2-6, this method requires pre-addition of $B_r + B_i$ and pre-subtractions, $A_r - A_i$ and $B_r - B_i$, before the binary multiplications, resulting in an increase of critical path delay. Although addition is generally less expensive in area than multiplication, the overall savings in hardware does not offset the non-trivial critical path delay. Therefore, the complex multiplication scheme given in Equation (2.33) will be utilized in this research.

2.2.2 Comparison of Different Complex Radices

Representing complex numbers with a complex radix implies that the complex numbers can be manipulated without separating the real and the imaginary part. It is supposed that in these complex radices, the complex-number computational arithmetic will be simplified. For example, complex number multiplication may only need one complex radix multiplication and hence provide a major performance improvement. Can a complex radix system really achieve such improvement? In the analysis, the various complex radices are compared, although it is interesting that no significant improvement is achieved compared to the traditional 2's-complement binary representation of complex numbers with the real and imaginary parts treated separately. Further, many of the alternative complex radix representations are unbalanced or fractal, thereby providing, in limited precision hardware, a significant representation issue for the range of the real and imaginary number components as well as the positive and negative fixed-point values. A review of these alternative complex radices follows.

2.2.2.1 Radix-(2j)

As early as 1960, Knuth [63],[64] proposed radix-(*2j*) which leads to an interesting system called "quater-imaginary" (by analogy with "quaternary" or base-4). In this system every complex number is represented with the digits 0, 1, 2, and 3 without a sign. For example:

$$(11210.31)_{2j} = 16 + (-8j) + 2\times(-4) + (2j) + 3\times(-\frac{1}{2}j) + (-\frac{1}{4}) = 7\frac{3}{4} - 7\frac{1}{2}j \qquad (2.35)$$

Here the number $(a_{2N}\mathrm{L}\ a_1 a_0.a_{-1}\mathrm{L}\ a_{-2K})_{2j}$ is equal to

$$(a_{2N}\mathrm{L}\ a_2 a_0.a_{-2}\mathrm{L}\ a_{-2K})_{-4} + 2j(a_{2N-1}\mathrm{L}\ a_3 a_1.a_{-1}\mathrm{L}\ a_{-2K+1})_{-4} \qquad (2.36)$$

Conversion to and from quater-imaginary radix reduces to the conversion to and from negative quaternary representation of the real and imaginary parts. In his book [64], Knuth proposed that the interesting property of this system is that it allows the multiplication and division of complex numbers to be done in a fairly unified manner without treating real and imaginary parts separately. For example, we can multiply two numbers in this system, much as we may do with any radix, by merely using a different carry rule. Whenever a sum digit exceeds 4, we subtract 4 from the sum digit and carry −1 two columns to the left; when a sum digit is negative, we add four to it and carry +1 two columns to the left.

Representing complex numbers in radix-(*2j*) is the same as representing the real and imaginary parts in radix-(-4). Although there is no sign to deal with in radix-(-4), the number system is imbalanced. The imbalance of the negative-base number system in Zohar's work [102] isn't correct. Zohar's results [102] are shown in Equations (2.37) and (2.38):

For radix $q<0$, the maximum positive number is given by

$$P = \frac{q^{2[(D+1)/2]}-1}{|q|+1} \tag{2.37}$$

and the minimum negative number is given by

$$N = q\frac{q^{2[D/2]}-1}{|q|+1} \tag{2.38}$$

Here are the correct results. Consider a system that uses D digits to represent numbers in the base $q<0$. When D is even, the largest representable integer is the positive number P, whose representation is:

$$P = (0,|q|-1,\mathrm{L}\ ,0,|q|-1,0,|q|-1). \tag{2.39}$$

Its value is given by,

$$P = (|q|-1)\frac{1-q^{D-2}q^{2}}{1-q^{2}} = \frac{|q|^{D}-1}{1+|q|} \tag{2.40}$$

Similarly, the smallest integer (N) is:

$$N = (|q|-1,\mathrm{L}\ ,0,|q|-1,0,|q|-1,0). \tag{2.41}$$

Its value is given by,

$$N = (1-|q|)\frac{|q|-|q|^{D-1}|q|^{2}}{1-|q|^{2}} = \frac{|q|(1-|q|^{D})}{1+|q|} \tag{2.42}$$

The number of integers contained in the closed interval defined this way is $P-N+1$. That is:

$$P+N-1 = P = \frac{|q|^{D}-1}{1+|q|} + \frac{|q|(|q|^{D}-1)}{1+|q|} + 1 = |q|^{D} \tag{2.43}$$

The result is very similar when D is odd. The largest representable integer is the positive number P whose representation is

$$P = (|q|-1, \mathrm{L}\ , 0, |q|-1, 0, |q|-1). \tag{2.44}$$

Its value is given by:

$$P = (|q|-1)\frac{1-q^{D-1}q^{2}}{1-q^{2}} = \frac{|q|^{D+1}-1}{1+|q|} \tag{2.45}$$

and the smallest integer (*N*) is

$$N = (0, |q|-1, \mathrm{L}\ , 0, |q|-1, 0, |q|-1, 0). \tag{2.46}$$

Its value is given by:

$$N = (1-|q|)\frac{|q|-|q|^{D-2}|q|^{2}}{1-|q|^{2}} = \frac{|q|(1-|q|^{D-1})}{1+|q|} \tag{2.47}$$

The number of integers contained in the closed interval defined this way is $P-N+1$. That is:

$$P+N-1 = P = \frac{|q|^{D+1}-1}{1+|q|} + \frac{|q|(|q|^{D-1}-1)}{1+|q|} + 1 = |q|^{D} \tag{2.48}$$

This, however, is the number of different configurations of the *D* digits. We conclude, therefore, that *D* digits span all the integers from *N* to *P*; regardless of *D* being even or odd. A troubling result is that the closed interval (*N,P*) is quite asymmetrical. A simple example will illustrate these statements.

Assume

$$q = -10 \text{ and } D = 3$$

Then the largest number is $(909)_{-10} = 909$, and the smallest is $(90)_{-10} = -90$.

Assume

$$q = -10 \text{ and } D = 4$$

Then the largest number is $(0909)_{-10} = 909$, and the smallest is $(9090)_{-10} = -9090$.

2.2.2.1.1 Complex Number Addition in Radix-(2*j*)

Dao [58] proposed a hardware implementation for the radix-(2*j*) addition. The adding of two numbers X and Y in the quater-imaginary system, as in any positional representation, consists of adding digits of the same weight. The modulo-(-4) result produces a sum digit and a carry digit. In this radix, the carry is -1 and has a weight equal to that of the digit two digit positions to the left:

$$x_i(2j)^i + y_i(2j)^i = (x_i + y_i)(2j)^i$$

where

$$\begin{aligned} &4 \le (x_i + y_i) \le 6 \\ &x_i + y_i = c_i(2j)^2 + s_i \end{aligned} \tag{2.49}$$

For example:

$$\begin{array}{rrrr} & 5 & + & 10j \\ + & 8 & + & 2j \\ \hline \\ \\ & 13 & + & 12j \end{array} \qquad \begin{array}{cccccc} 1 & 1 & 3 & 3 & 1 & 1 \\ 0 & 1 & 0 & 2 & 1 & 0 \\ \hline 1 & 2 & 3 & 1 & 2 & 1 \\ & -1 & & & & \\ \hline 1 & 1 & 3 & 1 & 2 & 1 \end{array}$$

Figure 2-7. An Example of Addition in Radix-(2*j*) [58]

Actually, the radix-(2*j*) adder is a radix-(-4) adder in the separate even and odd digit positions. The negative radix addition for real numbers is further investigated in [103],[104]. We conclude that the radix-(2*j*) addition reduces to the radix-(-4) addition in the even and odd positions separately.

2.2.2.1.2 Complex Number Subtraction in Radix-*(2j)*

In negative bases like (-4), no explicit sign digit is required in the representation. The negation of a number is obtained by taking the 4's-complement of each non-zero digit together with a positive carry digit of 1 two positions ahead:

$$\begin{aligned}-x_k(2j)^k &= (-4+\overline{x_k})(2j)^k \text{ with } \overline{x_k} = 4 - x_k \\ &= (2j)^{k+2} + \overline{x_k}(2j)^k\end{aligned} \tag{2.50}$$

The subtraction of a number X is reduced to adding its 4's-complement with proper carry propagation:

Similar to addition, the implementation of radix-(2*j*) subtraction is radix-(-4) subtraction in the even and odd positions separately.

```
     5  +  10j           1  1  3  3  1  1
 -  (  8  +  2j  )       0  3  0  2  3  0
 ----------------           1  1           carry from 4'S-complement
                     +     -1 -1           carry from radix-(-4) addition
                         -----------------
   -  3  +  8j           1  0  3  1  0  1
```

Figure 2-8. An Example of Subtraction in Radix-(2*j*) [58]

2.2.2.1.3 Complex Number Multiplication in Radix-*(2j)*:

Serial multiplication, i.e., one digit of the multiplier at a time, proceeds as in the binary case. Given,

$$X = x_0(2j)^0 + \mathrm{L} + x_{N-1}(2j)^{N-1}$$
$$Y = y_0(2j)^0 + \mathrm{L} + y_{N-1}(2j)^{N-1}$$

the product is:

$$Z = XY = \sum_{k=0}^{N-1}\sum_{i=0}^{N-1}[y_k x_i (2j)^i](2j)^k \tag{2.51}$$

The digit product $y_k x_i$ can generate a carry (0,-1,-2), which must be added to the digit position two places ahead in the partial sum:

$$\begin{aligned} y_k x_i &= -c_{k,i}(-4) + d_{k,i} = -c_{k,i}(2j)^2 + d_{k,i} \\ Z &= \sum_{k=0}^{N-1}\sum_{i=0}^{N-1}[-c_{k,i}(2j)^{i+2} + d_{k,i}(2j)](2j)^k \end{aligned} \tag{2.52}$$

The terms inside the bracket represent the partial product from the lower digit of the multiplier.

Notice that shifting a number X one position to the left is equivalent to rotating the vector x by 90 degrees and doubling its length, while shifting to the right one position results in a –90 degree rotation and halving the length.

```
     5 + 10j                        1  1  3  3  1  1
X    8 +  2j                        0  1  0  2  1  0
-----------                    -------------------
    20 + 90j                     1  1  3  3  1  1  0   Partial Product
                              2  2  2  2  2  2         Partial Product
                             -1 -1
                             ----------------------
                              1  1  2  1  1  3  1  0
                        1  1  3  3  1  1               Partial Product
                       -1 -1
                       ----------------------------
                                    3  2  1  3  1  0
```

Figure 2-9. An Example of Multiplication in Radix-(2*j*) [58]

The implementation of radix-(2*j*) multiplication results in a radix-(2*j*) partial product generator with even and odd positions separately followed by a radix-(-4) addition tree to generate the final product (see example in Figure 2-9).

An analysis of Knuth's "quater-imaginary" radix shows that there are several disadvantages of this imaginary number system:

- Since the numbers obtained from sensors and digital systems are normally 2's-complement binary numbers, a conversion from 2's-complement to radix-(2*j*) must be conducted. Our research [105] shows that the implementation of this conversion procedure will require a delay on the order of a carry-lookahead adder and will add an additional computational delay in the critical path.

- A key computation for complex numbers in radix-(2*j*) will invariably be multiplication. Without further developments, radix-(2*j*) is slow, compared to 2's-complement binary multiplication that uses Wallace trees, redundant binary number addition, Booth encoding, or array multipliers.
- Representing complex numbers in radix-(2*j*) is the same as representing the real and imaginary parts in radix-(-4). As previously shown in Equations (2.40), (2.42), (2.45) and (2.47), the radix-(-4) is an imbalanced system while a traditional positive radix system is a balanced number system.

Recently, Aoki [31],[97],[98] and McIlhenny [99],[100] investigated the redundant complex radix-(2*j*) arithmetic for high-speed signal processing with emphasis on complex number addition and multiplication. The addition of two numbers, $X = (x_{K-1} \mathrm{L}\ x_i \mathrm{L}\ x_{-L})$ and $Y = (y_{K-1} \mathrm{L}\ y_i \mathrm{L}\ y_{-L})$ in the redundant complex number system (2*j*,3), where $x_i, y_i \in \{-3,-2,-1,\ldots 1,2,3\}$, is performed by the following three steps for each digit:

$$\begin{aligned} &\text{Step 1: } z_i = x_i + y_i \\ &\text{Step 2: } -4c_i + w_i = z_i \\ &\text{Step 3: } s_i = w_i + c_{i-2} \end{aligned} \tag{2.53}$$

Here z_i is the linear sum, w_i is called the intermediate sum, and c_i is the carry. This so-called radix-(2*j*) redundant number system is actually a radix-(4) redundant number system with the real and imaginary parts treated separately.

In summary, this analysis shows that the non-redundant radix-(2*j*) number system has several disadvantages and the redundant radix-(2*j*) complex number system is a radix-(4) redundant binary system, with real and imaginary parts treated separately.

Therefore, radix-(2*j*) offers no overall hardware implementation advantage over a conventional binary number system.

2.2.2.2 Radix-($j\sqrt{2}$)

A system similar to radix-(2*j*) that uses only the digits 0 and 1 is based on $j\sqrt{2}$. This scheme, however, requires an infinite non-repeating expansion for the simple number $0+1j$. Slekys [67] defined arithmetic operations on a modified bi-imaginary number system based on radix-$(j\sqrt{2})$. If a modified bi-imaginary complex system is used to encode each complex number $a+jb$ as $a+j\sqrt{2}c$, then the number $(a_{2N}\mathrm{L}\ a_1a_0.a_{-1}\mathrm{L}\ a_{-2K})_{\sqrt{2}j}$ is equal to:

$$(a_{2N}\mathrm{L}\ a_2a_0.a_{-2}\mathrm{L}\ a_{-2K})_{-2}+\sqrt{2}j(a_{2N-1}\mathrm{L}\ a_3a_1.a_{-1}\mathrm{L}\ a_{-2K+1})_{-2}$$

The conversion to and from radix-$(j\sqrt{2})$ notation reduces to the conversion to and from a negative-2 representation of the real and imaginary parts separately. Slekys defined algorithms for complex number addition, subtraction, multiplication and division. However, since Slekys's radix- $(j\sqrt{2})$ system is not redundant, the computational arithmetic requires additional operations comparable to the conventional 2's-complement representation of complex numbers. To see this, multiplication in this system is considered.

Define the multiplier and multiplicand as the complex numbers Z_1 and Z_2 respectively, where:

$$Z_1 = A + jB$$
$$Z_2 = C + jD$$

Using a modified bi-imaginary representation let,

$$Z_1' = A + j\sqrt{2}B$$
$$Z_2' = C + j\sqrt{2}D$$

Then the multiplication of Z_1 and Z_2 will be:

$$Z_1 \bullet Z_2 = Z_1' \bullet Z_2' + B \bullet D \tag{2.54}$$

Equation (2.54) shows that in the bi-imaginary complex-radix system the multiplication of two complex numbers will be composed of one complex number multiplication requiring four multiplications and two additions, plus one real number multiplication and one complex number addition.

For complex radix-($j\sqrt{2}$), we can also use a redundant binary system for the computational arithmetic. Similar to the case of radix-(2*j*), this redundant complex radix-($j\sqrt{2}$) is just a radix-(2) redundant binary system with the real and imaginary parts treated separately.

2.2.2.3 ***Radix-(j-1)***

A binary complex number system is also obtained by using the base (*j*-1), as first suggested by Penney [66]. Further studies of this radix were conducted by Jamil [62] and Blest [55]. Jamil shows that the conversion from 2's-complement to radix-(*j*-1) is actually the conversion from radix-(2) to radix-(-4) for the real and imaginary parts separately, with a (*j*-1)-based addition needed to complete the conversion procedure. The multiplication and division of complex numbers based on this radix are also presented in Jamil's [62] and Blest's work [55]. Hardware implementations were not specifically addressed. In fact, it appears that the hardware for this radix will possess considerable latency and gate count due the necessary carry detection logic requirement for the addition operation.

In radix-(j-1), there exists a carry propagation problem in complex number addition that further exacerbates the partial product additions in the multiplication hardware. To deal with this carry propagation problem for high-speed parallel hardware, a zero-detector is required for each digit. The zero-detector adds additional latency and gate complexity.

The value of an N-bit binary number $A = (a_{N-1}, a_{N-2}, \cdots a_1, a_0)$ with radix-(j-1) can be written in the form of a power series as follows:

$$A = a_{N-1}(-1+j)^{N-1} + a_{N-2}(-1+j)^{N-2} + \cdots + a_1(-1+j) + a_0 \tag{2.55}$$

where the coefficients $a_{N-1}, a_{N-2}, \cdots a_1, a_0 \in \{0,1\}$. As an example, if N is a 16 bit number, the powers of $-1+j$ associated with the coefficients will be (from bottom to top, right to left, in groups of four):

[Row 4] (-128-j128), (0+j128), (64-j64), (-64+j0)
[Row 3] $(32+j32)$, $(0-j32)$, $(-16+j16)$, $(16+j0)$
[Row 2] $(-8-j8)$, $(0+j8)$, $(4-j4)$, $(-4+j0)$
[Row 1] $(2+j2)$, $(0-j2)$, $(-1+j1)$, $(1+j0)$

To describe the hardware implementation difficulties for this system, we need only consider the addition operation. In radix-$(-1+j)$, we have:

$$(1)_{-1+j} + (1)_{-1+j} = (2)_{-1+j} = (1100)_{-1+j} \tag{2.56}$$

In radix-$(-1+j)$ there are two carries from one bit position. The addition truth table for one bit position is shown in the Table 2-F.

From the truth table, it is seen that there is a carry propagation problem in the addition of numbers in radix- $(-1+j)$. Figure 2-10 shows an example for $x_i = 1, y_i = 1, c_{i-2} = 1$ and $c_{i-3} = 1$.

In this example, $x_i = y_i = c_{i-2} = c_{i-3} = 1$, from Table 2-F and Equation (2.56), the carry-out c_{i+2} and c_{i+3} to the 2 and 3 digit positions to the left are $c_{i+2} = c_{i+3} = 2$. Then carry $c_{i+2} = 2$ will further propagate to $c_{i+4} = c_{i+5} = 1$ and carry $c_{i+3} = 2$ will further propagate to $c_{i+5} = c_{i+6} = 1$. Thus, carry $c_{i+5} = 1+1 = 2$ will propagate to $c_{i+7} = c_{i+8} = 1$.

Table 2-F. Truth Table for Radix-$(-1+j)$ One-Bit Addition

x_i	y_i	c_{i-2}	c_{i-3}	z_i	c_{i+2}	c_{i+3}
0	0	0	0	0	0	0
0	0	1	1	0	1	1
0	1	1	1	1	1	1
1	1	1	1	0	2	2
1	1	1	2	1	2	2
....						
1	1	2	2	0	3	3
....						
1	1	3	3	0	4	4
....						

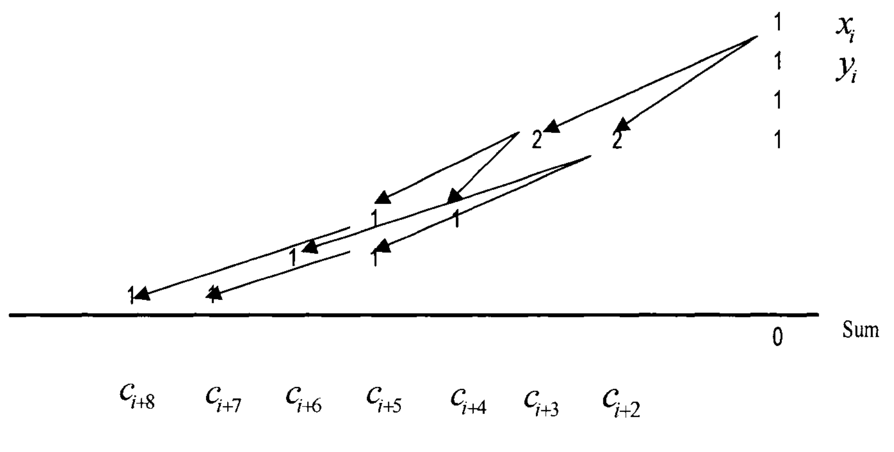


Figure 2-10. An Example for Radix-(j-1) Carry Propagation

A possible hardware method to work around this problem is a zero-detector based on the equality:

$$(11)_{-1+j} + (111)_{-1+j} = (0)_{-1+j} \tag{2.57}$$

However, the zero-detector is expensive, serial in nature, and produces high latency. From this analysis, the radix- $(-1+j)$ has no advantage in the complex number computational hardware for addition and multiplication.

In summary, different complex radices such as radix-$(2j)$, radix-$(-1+j)$ and radix-$(j\sqrt{2})$ are studied. It is shown that these complex radices have no advantage over traditional binary number systems in hardware implementation. Chang's research [56],[57] also supports this conclusion. In Chang's research, a RIA (Real Imaginary Alternate) complex number system is proposed. In essence, his system represents complex numbers in 2's-complement binary form with interleaved real and imaginary parts. Therefore, based upon traditional binary number representation and the previously discussed real-number inner-product processor, a high-performance complex multiplier and complex number inner-product processor is developed in the following sections.

2.2.3 Complex Number Multiplier and Inner-Product Computation

2.2.3.1 ***RB Complex Number Multiplier***

Applying the inline multiplication method to the complex inner-product requires no modification to Figure 2-2 to produce the redundant binary imaginary part, $j(A_R B_I + A_I B_R)$. However, the real part, $A_R B_R - A_I B_I$, requires a final redundant binary subtraction, rather than an addition. The subtraction is easily implemented by modifying the redundant binary adder (RBA) to add the complement of the redundant binary number $(-A_lB_l)_{\mathrm{RB}}$, since $(-A_lB_l)_{\mathrm{RB}} = \overline{(A_lB_l)_{\mathrm{RB}}}$, where $\overline{-1} = 1$, $\bar{1} = -1$, and $\bar{0} = 0$. In the actual hardware implementation of Figure 2-2, the addend or its RB complement are multiplexed into the RBA, thereby converting it to an adder/subtracter. Defining the control signal Real_Img as Real_Img=1, for the $A_0B_0 - A_1B_1$ computation and Real_Img=0 for the $A_0B_0 + A_1B_1$ computation, and then the inline implementation for $A_0B_0 \pm A_1B_1$ is shown in Figure 2-11. For Booth encoding of the inline method, no further modification is required.

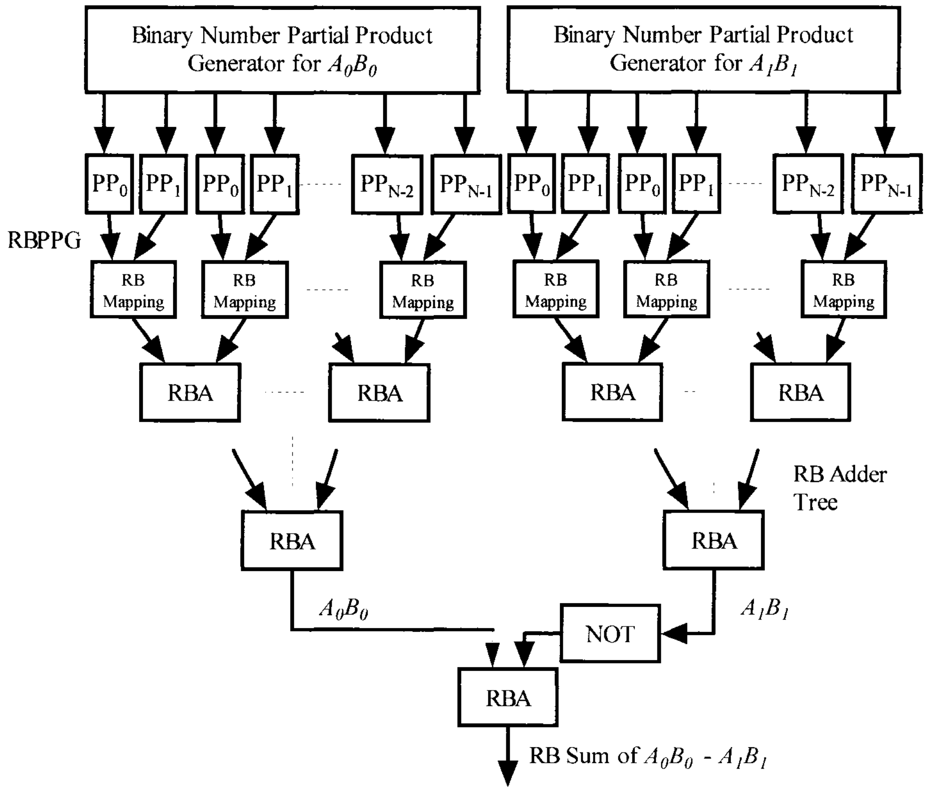

Binary Number Partial Product Generator for A_0B_0
Binary Number Partial Product Generator for A_1B_1
PP_0
PP_1
PP_{N-2}
PP_{N-1}
RBPPG
RB Mapping
RBA
RB Adder Tree
A_0B_0
A_1B_1
NOT
RB Sum of A_0B_0 - A_1B_1

Figure 2-11. Inline Implementation of A_0B_0 - A_1B_1

The cross partial product method is slightly more difficult and requires mapping the difference of two 2's-complement numbers into a redundant binary number to compute the real part of the complex product, $A_R B_R - A_I B_I$. The result given here is similar to that provided in [49] and is derived much the same as given in Section 2.1.1 for $A_0 B_0 + A_1 B_1$.

$$\begin{aligned}
A_0B_0 - A_1B_1 &= (-a_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2} a_{i,0}2^i)(-b_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2} b_{i,0}2^i) - \\
&\quad (-a_{N-1,1}2^{N-1} + \sum_{i=0}^{N-2} a_{i,1}2^i)(-b_{N-1,1}2^{N-1} + \sum_{i=0}^{N-2} b_{i,1}2^i) \\
&= 2^{N-1}(a_{N-1,0}b_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0})2^i - a_{N-1,1}b_{N-1,1}2^{N-1} + \sum_{i=0}^{N-2}(a_{i,1}b_{N-1,1})2^i) + \\
&\quad \sum_{i=0}^{N-2} 2^i\{b_{i,0}(-a_{N-1,0}2^{N-1} + \sum_{j=0}^{N-2} a_{j,0}2^j) + b_{i,1}(a_{N-1,1}2^{N-1} - \sum_{j=0}^{N-2} a_{j,1}2^j)\} \\
&= 2^{N-1}\{(a_{N-1,0}b_{N-1,0} - a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0} + a_{i,1}b_{N-1,1})2^i\} + \\
&\quad \sum_{i=0}^{N-2} 2^i\{(-a_{N-1,0}b_{i,0} + a_{N-1,1}b_{i,1})2^{N-1} + \sum_{j=0}^{N-2}(a_{j,0}b_{i,0} - a_{j,1}b_{i,1})2^j\}
\end{aligned} \tag{2.58}$$

Considering the first term of Equation (2.58),

$$\begin{aligned}
&2^{N-1}\{(a_{N-1,0}b_{N-1,0} - a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0} + a_{i,1}b_{N-1,1})2^i\} \\
&= 2^{N-1}\{(a_{N-1,0}b_{N-1,0})2^{N-1} + \sum_{i=0}^{N-2}(-a_{i,0}b_{N-1,0})2^i\} + \{(-a_{N-1,1}b_{N-1,1})2^{N-1} + \sum_{i=0}^{N-2}(a_{i,1}b_{N-1,1})2^i\} \\
&= 2^{N-1}\{\mu_{N-1,0}2^{N-1} + \sum_{i=0}^{N-2}\mu_{i,0}2^i\}
\end{aligned} \tag{2.59}$$

where $\mu_{i,0}$ $(0 \le i \le N-1)$ is a redundant binary number with

$$\mu_{N-1,0} = a_{N-1,0}b_{N-1,0} - a_{N-1,1}b_{N-1,1} \text{ and } \mu_{i,0} = -a_{i,0}b_{N-1,0} + a_{i,1}b_{N-1,1} \text{ for } 0 \le i \le N-2. \tag{2.60}$$

Encoding $\mu_{i,0}$ as two binary bits for $0 \le i \le N-2$:

$$\begin{aligned}
&\mu^-_{N-1,0} = a_{N-1,0}\, b_{N-1,0}, \quad \mu^+_{N-1,0} = \overline{a_{N-1,1}\, b_{N-1,1}}, \\
&\mu^-_{i,0} = \overline{a_{i,0}b_{N-1,0}}, \text{ and } \quad \mu^+_{i,0} = a_{i,1}b_{N-1,1}.
\end{aligned} \tag{2.61}$$

Converting the last term of Equation (2.58) to redundant binary,

$$\begin{aligned}&\sum_{i=0}^{N-2} 2^i\{(-a_{N-1,0}b_{i,0}+a_{N-1,1}b_{i,1})2^{N-1}+\sum_{j=0}^{N-2}(a_{j,0}b_{i,0}-a_{j,1}b_{i,1})2^j\}\\ &=\sum_{i=0}^{N-2} 2^i\{\nu_{i,N-1}2^{N-1}+\sum_{j=0}^{N-2}\nu_{i,j}2^j\}\end{aligned} \tag{2.62}$$

where $\nu_{i,j}$ $(0\le i\le N-2,\ 0\le j\le N-1)$ is a redundant binary number and

$$\nu_{i,j}=a_{j,0}b_{i,0}-a_{j,1}b_{i,1} \text{ for } 0\le j\le N-2, \text{ and } \nu_{i,N-1}=-a_{N-1,0}b_{i,0}+a_{N-1,1}b_{i,1}. \tag{2.63}$$

Encoding $\nu_{i,j}$ $(0\le i\le N-2,\ 0\le j\le N-1)$ as a redundant binary number using

two binary bits:

$$\begin{aligned}&\nu^-_{i,N-1}=\overline{a_{N-1,0}b_{i,0}}\ ,\ \ \nu^+_{i,N-1}=a_{N-1,1}b_{i,1},\\ &\nu^-_{i,j}=a_{j,0}b_{i,0}, \text{ and }\ \nu^+_{i,j}=\overline{a_{j,1}b_{i,1}}.\end{aligned} \tag{2.64}$$

The general redundant binary equation for the summation of $A_0B_0-A_1B_1$ is:

$$A_0B_0-A_1B_1 = 2^{N-1}\{\mu_{N-1,0}2^{N-1}+\sum_{i=0}^{N-2}\mu_{i,0}2^i\}+\sum_{i=0}^{N-2}2^i\{\nu_{N-1,0}2^{N-1}+\sum_{i=0}^{N-2}\nu_{i,0}2^i\}. \tag{2.65}$$

Now consider the inner-product, $A_0B_0-A_1B_1$, using modified Booth encoding:

$$\begin{aligned}A_0B_0-A_1B_1 &= \sum_{i=0}^{\frac{N}{2}-1} Q_{i,0}(-a_{N-1,0}2^{N-1}+\sum_{k=0}^{N-2}a_{k,0}2^k)2^{2i}\\ &\quad-\sum_{i=0}^{\frac{N}{2}-1} Q_{i,1}(-a_{N-1,1}2^{N-1}+\sum_{k=0}^{N-2}a_{k,1}2^k)2^{2i}\\ &=\sum_{i=0}^{\frac{N}{2}-1}(C_{i,0}+g_{i,0})2^{2i}-\sum_{i=0}^{\frac{N}{2}-1}(C_{i,1}+g_{i,1})2^{2i}\\ &=\sum_{i=0}^{\frac{N}{2}-1}(C_{i,0}-C_{i,1}+g_{i,0}-g_{i,1})2^{2i}\end{aligned} \tag{2.66}$$

From Equation (1.10), the subtraction of two 2's-complement numbers can be considered as a redundant binary number. So Equation (2.66) becomes

$$
\begin{aligned}
A_0B_0 - A_1B_1 &= \sum_{i=0}^{\frac{N}{2}-1} (C_{i,0} - C_{i,1} + g_{i,0} - g_{i,1})2^{2i} \\
&= \sum_{i=0}^{\frac{N}{2}-1} (Y_{i,01} + g_{i,0} - g_{i,1})2^{2i} \\
&= \sum_{i=0}^{\frac{N}{2}-1} (Y_{i,01} + \gamma_{i,01})2^{2i}
\end{aligned}
\tag{2.67}
$$

where $Y_{i,01}$ is a redundant binary number from the subtraction of $C_{i,0}$ and $C_{i,1}$ with $Y_{i,01} = C_{i,0} - C_{i,1}$ and $\gamma_{j,01} = g_{j,0} - g_{j,1}$. The correction factor, $\gamma_{i,01}$, depends upon the value $g_{i,0}$ and $g_{i,1}$, as shown in Table 2-G.

Table 2-G. Booth Correction Factors for Redundant Binary Partial Product Generation of $A_0B_0 - A_1B_1$

$g_{i,0}$	$g_{i,1}$	$\gamma_{i,01} = g_{i,0} - g_{i,1}$	$\gamma_{i,01}^{-}$ $\gamma_{i,01}^{+}$
0	0	0	0 1
0	1	-1	0 0
1	0	1	1 1
1	1	0	1 0

From the Booth coding Table 2-A,

$$
\begin{aligned}
\gamma_{i,01}^{-} = g_{i,0} &= b_{i+1,0}\overline{b_{i,0}}\ \overline{b_{i-1,0}} + b_{i+1,0}\overline{b_{i,0}}\ b_{i-1,0} + b_{i+1,0}b_{i,0}\ \overline{b_{i-1,0}} \\
&= b_{i+1,0}\ \overline{b_{i,0}b_{i-1,0}} \\
\gamma_{i,01}^{+} = \overline{g_{i,1}} &= \overline{b_{i+1,1}\overline{b_{i,1}}\ \overline{b_{i-1,1}} + b_{i+1,1}\overline{b_{i,1}}\ b_{i-1,1} + b_{i+1,1}b_{i,1}\ \overline{b_{i-1,1}}} \\
&= \overline{b_{i+1,1}\ \overline{b_{i,1}b_{i-1,1}}}
\end{aligned}
\tag{2.68}
$$

Again, the "+" in the equations above is the Boolean OR operation.

2.2.3.2 *RB Complex Number Inner-Product Processor*

The inner-product of complex numbers $C_0, C_1, \cdots C_{M-1}$ and $D_0, D_1, \cdots D_{M-1}$,

$$C_0 = A_0 + jA_1,\ C_1 = A_2 + jA_3, \ldots C_{M-1} = A_{2M-2} + jA_{2M-1}$$
$$D_0 = B_0 + jB_1,\ D_1 = B_2 + jB_3, \ldots D_{M-1} = B_{2M-2} + jB_{2M-1} \tag{2.69}$$

is

$$\begin{aligned}\langle \boldsymbol{C}, \boldsymbol{D} \rangle &= \sum_{j=0}^{M-1} C_j D_j = \sum_{j=0}^{M-1} (A_{2j} + jA_{2j+1})(B_{2j} + jB_{2j+1}) \\ &= \sum_{j=0}^{M-1} (A_{2j}B_{2j} - A_{2j+1}B_{2j+1}) + j\sum_{j=0}^{M-1} (A_{2j}B_{2j+1} + A_{2j+1}B_{2j})\end{aligned} \tag{2.70}$$

The real part of the complex number inner-product is shown as Figure 2-12.

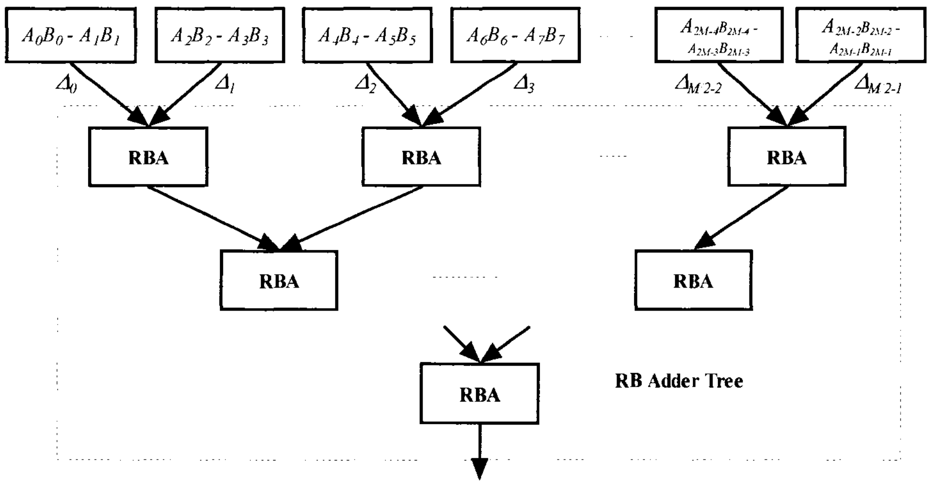


Figure 2-12. The Real Part of the Complex Number Inner-Product

The imaginary part of the complex number inner-product is shown as Figure 2-13:

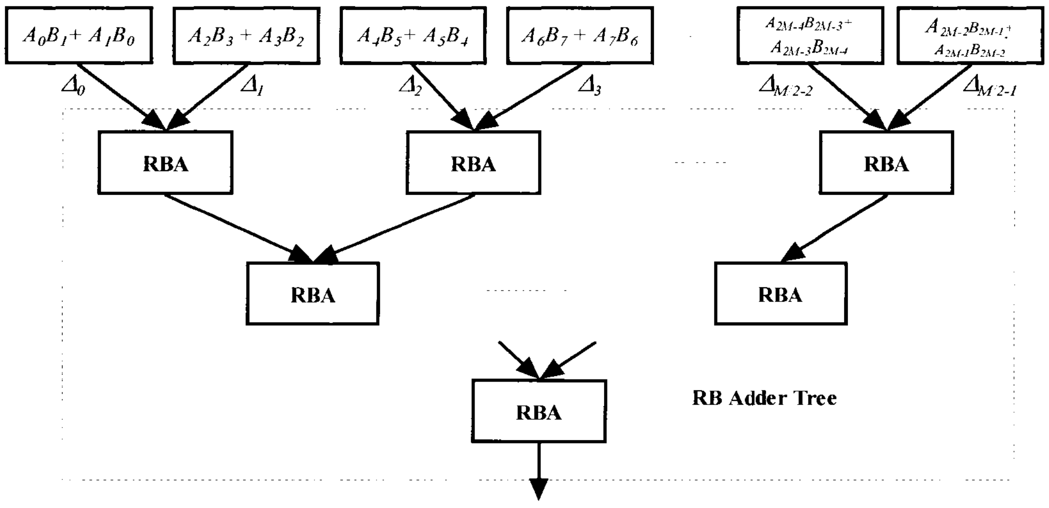


Figure 2-13. The Imaginary Part of Complex-Number Inner-Product

Defining a control signal Real_Img=1/0 for $AB-CD/AB+CD$ computation, then the overall structure of a unified inner-product processor for $AB \pm CD$ is shown in Figure 2-14:

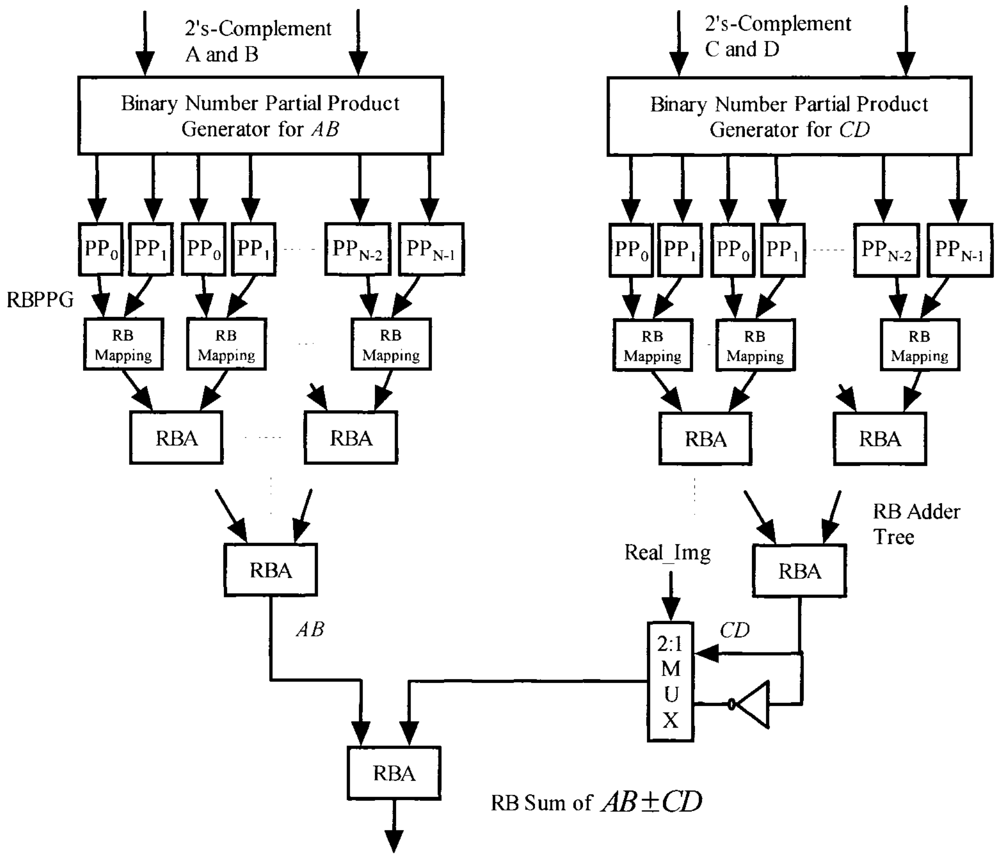


Figure 2-14. Unified RB IP Processor for $AB \pm CD$

Using the above structures, the real and imaginary parts of the complex number inner-products are computed in redundant binary form. Finally, a RB to 2's-complement converter is needed to convert the redundant binary inner-product real and imaginary parts to 2's-complement form, if required.

2.3 Inner-Product Computation Comparison

In this section, the computation time for real number inner-product processing is compared between the Texas Instruments TMS320C6000 series and the RB inner-product processor. The sample inner-product assembly code using TMS320C6000 [92] is shown in Figure 2-15:

```
            MVK .S1 100, A1        ; set up loop counter
            ZERO .L1 A7            ; zero out accumulator
     LOOP:
            LDH .D1 *A4++,A2       ; load ai from memory
            LDH .D1 *A3++,A5       ; load bi from memory
            NOP 4                  ; delay slots for LDH
            MPY .M1 A2,A5,A6       ; ai * bi
            NOP                    ; delay slot for MPY
            ADD .L1 A6,A7,A7       ; sum += (ai * bi)
            SUB .S1 A1,1,A1        ; decrement loop counter
    [A1]    B .S2 LOOP             ; branch to loop
            NOP 5                  ; delay slots for branch
            ; Branch occurs here
```

Figure 2-15. An Example Code of Fixed-Point Inner-Product [92]

To analyze the execution clock cycles of this sample, a dependency graph is very useful. Dependency graphs can help analyze loops by showing the flow of instructions and data in an algorithm. These graphs also show how instructions depend on one another. The following terms are used in defining a dependency graph:

- A *node* is a point on a dependency graph with one or more data paths flowing in and/or out.
- The *path* shows the flow of data between nodes. The numbers beside each path represent the number of cycles required to complete the instruction.
- An instruction that writes to a variable is referred to as a parent instruction and defines a *parent node.*

- An instruction that reads a variable written by a parent instruction is referred to as its child and defines a *child node.*

Use the following steps to draw a dependency graph:

1) Define the nodes based on the variables accessed by the instructions.
2) Define the data paths that show the flow of data between nodes.
3) Add the instructions and latencies.
4) Add the functional units.

Figure 2-16 shows the dependency graph for the fixed-point inner-product assembly instructions, shown in Figure 2-15 and their corresponding register allocations:

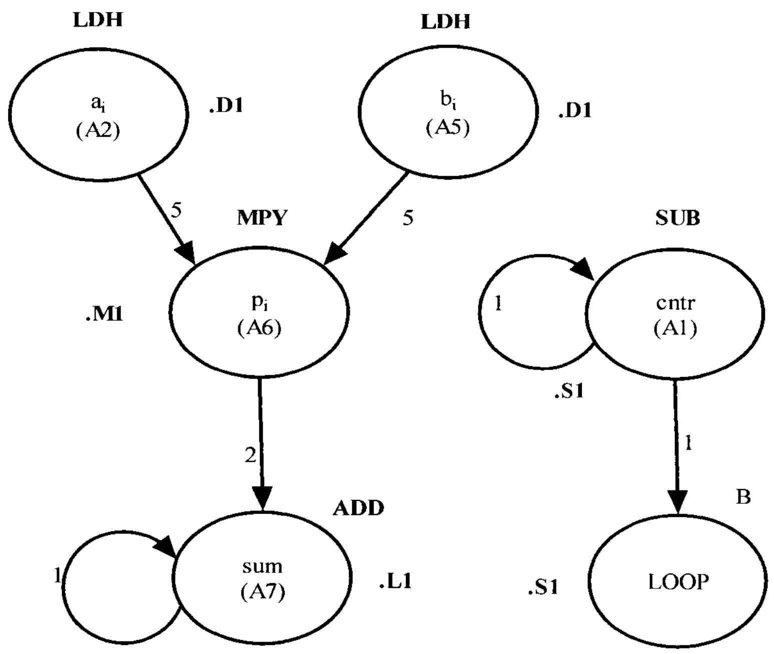


Figure 2-16. Dependency Graph of Fixed-Point Inner-Product [92]

Figure 2-16 provides the following observations:

- The two LDH instructions, which write the values of a_i and b_i, are parents of the MPY instruction. Five cycles for the parent (LDH) instruction are needed. Therefore, if the LDH is scheduled on cycle i, then its child (MPY) cannot be scheduled until cycle $i + 5$.

- The MPY instruction, which writes the product p_i, is the parent of the ADD instruction. The MPY instruction takes two cycles to complete.
- The ADD instruction adds p_i (the result of the MPY) to the sum. The output of the ADD instruction feeds back to become an input on the next iteration and, thus, creates a loop carry path.

The dependency graph for this inner-product algorithm has two separate parts since the decrement of the loop counter and the branch do not read or write any variables from the other part. The loop counter graph shows the following:

- The SUB instruction writes to the loop counter, cntr. The output of the SUB instruction feeds back and creates a loop carry path.
- The branch (B) instruction is a child of the loop counter.

Executing this inner-product code serially requires 16 cycles for each iteration plus two cycles to set up the loop counter and initialize the accumulator, thus 100 iterations require 1602 cycles. For the fixed-point TMS320C62X ('C62X) devices, which are operated typically at a 200 MHz clock (5 ns) frequency, 100 iterations require:

$$1602 \times 5 = 8010ns \qquad (2.71)$$

For the RB inner-product processor, Figure 2-17 shows the structure to implement the real number inner-product computation. For an accurate and fair comparison, we use only one RB multiplier and RB accumulator.

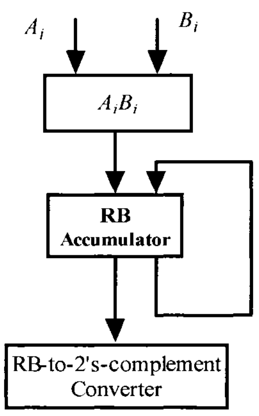


Figure 2-17. RB Inner-Product Implementation

A CMOS implementation of the RB multiplier [3] with 0.5 μm fabrication shows that a 54×54 bit multiplier achieves 8.8 ns delay, which includes 2.4 ns delay for the RB-to-2's-complement converter. The actual delay of the RB multiplier is only 7.2 ns. Compared to the implementation of TMS320C62X processor, if two-stage pipelines are used for the multiplication and the RB multiplier is employed in the TMS320C62X, then the clock cycle can be reduced to 6.4/2=3.2ns. Assuming that all the other instruction operations (LDH, ADD, SUB, etc) take the same time in the RB IP processor, then for 100 iterations of inner-product computations, the total time required is

$$1602 \times 3.2 = 5126.4ns \tag{2.72}$$

Table 2-H shows the comparison result using TMS320C62X and the RB inner-product processor for 100 iterations of inner-product computations:

Table 2-H. Comparison of IP Computation between TMS320C62X and RB Inner-Product Processor

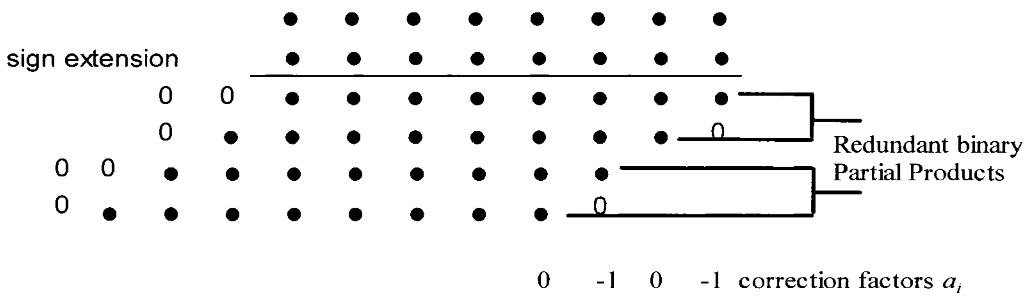
	100 Iterations
TMS320C62X	8010 ns
RB Inner-Product Processor	5126.4 ns

2.4 Implementation of Unified Signed/Unsigned Multiplier

In this section, a unified signed/unsigned multiplier is developed using the RB inner-product core without and with Booth encoding. An unsigned binary number can be considered as a 2's-complement number with an extra sign bit '0' padded before the MSB (Most Significant Bit). For example, an unsigned binary number 10001111 can be considered as a signed binary number 010001111 with an extra sign bit '0'.

2.4.1 Unifled Signed/Unsigned Muitiplier Without Booth Coding

From Equation (1.6), a RB number, Δ, can be derived from the addition of a pair of 2's-complement numbers. Thus for an unsigned $N \times N$ multiplier of $A \times B$, N unsigned partial products are generated. These N unsigned partial products are converted to N+1 signed partial products with extra bit 0 padded before MSB and are mapped to $N/2$ RB partial products with correction factors as shown in Figure 2-18.



sign extension
0 0
0
0 0
0
0
0
Redundant binary Partial Products
0 -1 0 -1 correction factors a_i

Figure 2-18. Unsigned Multiplier with Partial Product Generation

Consider the partial products of PP_{N-2} and PP_{N-1} for signed multiplier of A and B as:

$$PP_{N-2} = (-a_{N-1}b_{N-2}2^N + a_{N-1}b_{N-2}2^{N-1} + \sum_{i=0}^{N-2} a_i b_{N-2} 2^i)2^{N-2}$$
$$PP_{N-1} = -(-a_{N-1}b_{N-1}2^N + a_{N-2}b_{N-1}2^{N-1} + \sum_{i=1}^{N-2} a_{i-1} b_{N-1} 2^i)2^{N-2} \quad (2.73)$$

Where PP_{N-2} and $-PP_{N-1}$ are 2's-complement binary numbers, the sum of PP_{N-2} and PP_{N-1} are mapped into a RB number according to Equations (1.10), (1.11), (1.12) and Figure 1-14. The mapping structure is shown in Figure 2-19.

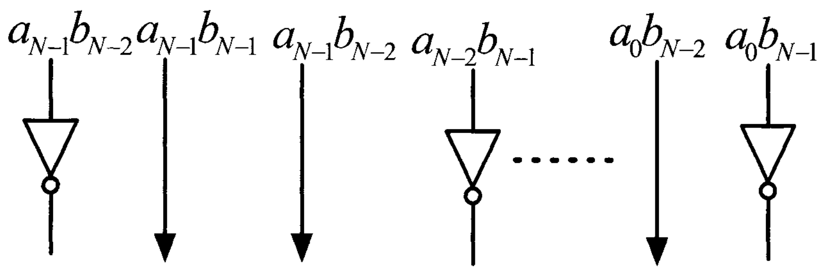


Figure 2-19. Mapping of PP_{N-2} and PP_{N-1} for Signed Multiplier into a RB Digit

Consider the partial products of PQ_{N-2} and PQ_{N-1} for unsigned multiplier of A and B as:

$$\begin{aligned} PQ_{N-2} &= (0\square 2^{N+1} + a_{N-1}b_{N-2}2^{N} + a_{N-1}b_{N-2}2^{N-1} + \sum_{i=0}^{N-2} a_i b_{N-2}2^{i})2^{N-2} \\ PQ_{N-1} &= (0\square 2^{N+1} + a_{N-1}b_{N-1}2^{N} + a_{N-2}b_{N-1}2^{N-1} + \sum_{i=1}^{N-2} a_{i-1} b_{N-1}2^{i})2^{N-2} \end{aligned} \tag{2.74}$$

Where PQ_{N-2} and PQ_{N-1} are 2's-complement binary numbers, the sum of PQ_{N-2} and PQ_{N-1} are mapped into a RB number according to Equations (1.6), (1.7), (1.8) and Figure 1-13 with an extra correction factor -1. The mapping structure is shown in Figure 2-20.

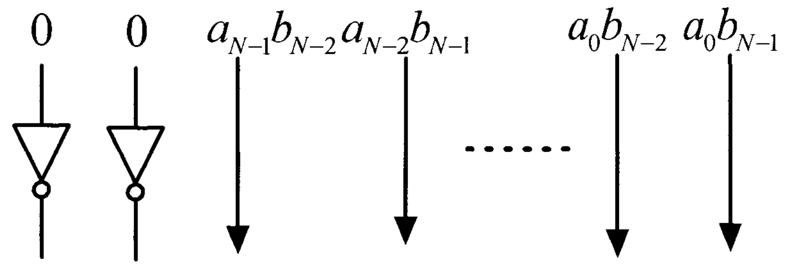


Figure 2-20. Mapping of PQ_{N-2} and PQ_{N-1} for Signed Multiplier into a RB Digit

Define a control signal, *SIGN*, where *SIGN*=1 for signed multiplication and *SIGN*=0 for unsigned multiplication, then the combined partial product PP'_{N-1} is realized in Figure 2-21.

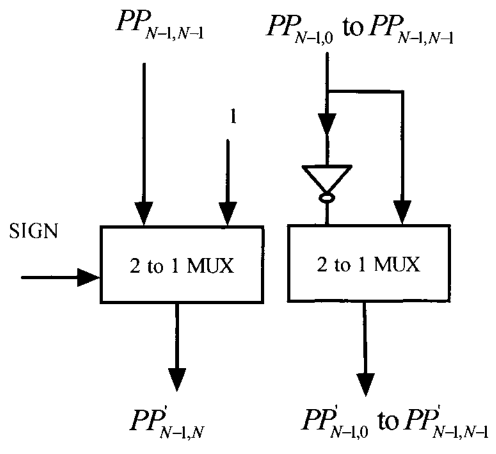

$PP_{N-1,N-1}$
$PP_{N-1,0}$ to $PP_{N-1,N-1}$
1
SIGN
2 to 1 MUX
2 to 1 MUX
$PP'_{N-1,N}$
$PP'_{N-1,0}$ to $PP'_{N-1,N-1}$

Figure 2-21. Circuit Realization of the Last Partial Product PP'_{N-1} for Signed/Unsigned Multiplier

Thus, for the implementation of *N*-bit unsigned multiplier, the correction factors are {0, -1, 0, -1, 0, -1, ...-1}. That is:

$$\begin{aligned} \alpha_{N-2} &= \alpha_{N-4} = \cdots = \alpha_2 = \alpha_0 = -1 \\ \alpha_{N-1} &= \alpha_{N-3} = \cdots = \alpha_3 = \alpha_1 = 0 \end{aligned} \tag{2.75}$$

For unsigned multiplication, all the correction factors except $\alpha_{N-2} = -1$ can be added in the RB addition tree. Here the α_{N-2} factor is combined with the first partial product, $PP0_{N-1}\ PP0_{N-2} \cdots PP0_1\ PP0_0$, which is shown in Figure 2-22:

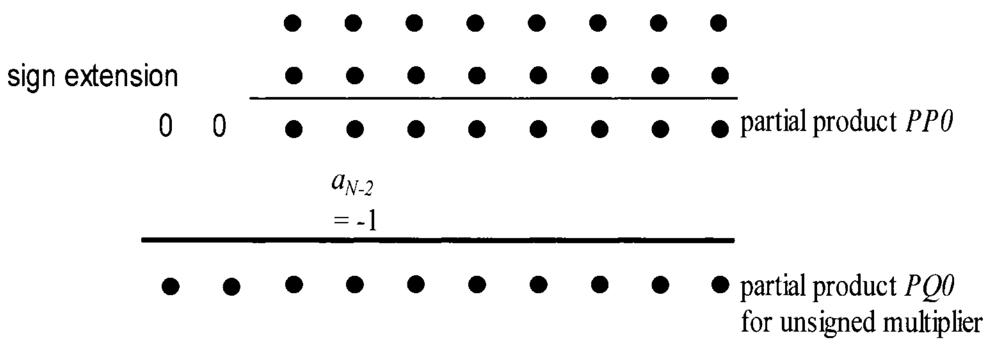


Figure 2-22. First Partial Product *PQ0* for Unsigned Multiplier

Table 2-I to Table 2-L are the truth tables for the first partial product *PQ0* of the unsigned multiplier:

Table 2-I. Partial Product $PQ0_0$ to $PQ0_{N-3}$ for Unsigned Multiplier

First Partial Products PP_0 $PP0_0$ to $PP0_{N-3}$		First Partial Products PQ_0 $PQ0_0$ to $PQ0_{N-3}$	
0	1	0	1

Table 2-J. Partial Product $PQ0_{N-2}$ for Unsigned Multiplier

First Partial Products PP_0 $PP0_{N-2}$		First Partial Products PQ_0 $PQ0_{N-2}$	
0	1	1	0

Table 2-K. Partial Product $PQ0_{N-1}$ for Unsigned Multiplier

$PP0_{N-2}$	$PP0_{N-1}$	First Partial Products PQ_0 $PQ0_{N-1}$
0	0	1
0	1	0
1	0	0
1	1	1

Table 2-L. Partial Product $PQ0_N$ and $PQ0_{N+1}$ for Unsigned Multiplier

$PP0_{N-2}$	$PP0_{N-1}$	First Partial Products PQ_0 $PQ0_N$ and $PQ0_{N+1}$
0	0	1
0	1	0
1	0	0
1	1	0

The logic equation of the partial product $PQ0$ for unsigned multiplier is as follows:

$$\begin{aligned}
&PQ0_0 = PP0_0 \\
&PQ0_1 = PP0_1 \\
&\ldots\ldots \\
&PQ0_{N-3} = PP0_{N-3} \\
&PQ0_{N-2} = \overline{PP0_{N-2}} \\
&PQ0_{N-1} = \overline{PP0_{N-1} \oplus PP0_{N-2}} \\
&PQ0_{N+1} = PQ0_N = \overline{PP0_{N-1} + PP0_{N-2}}
\end{aligned} \tag{2.76}$$

Figure 2-23 shows the circuit realization of the combined partial product PP_0'.

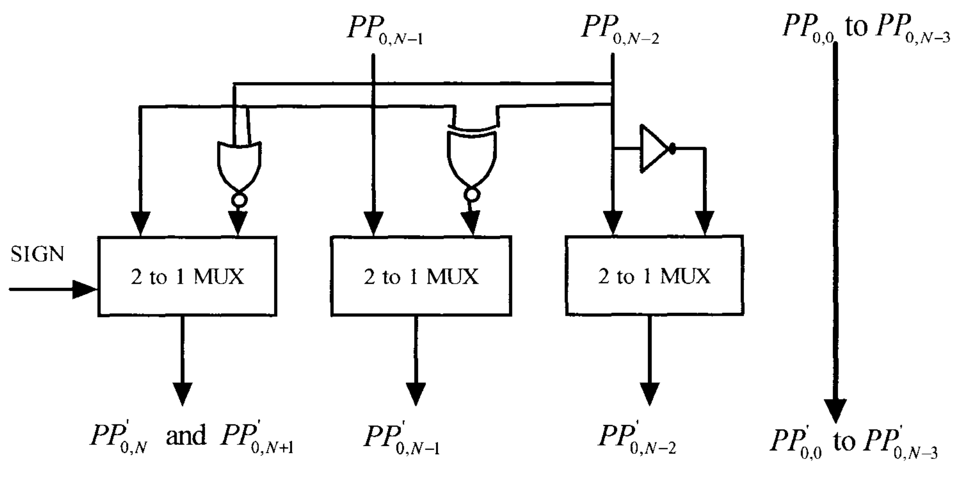


Figure 2-23. Circuit of the First Partial Product PP'_0 for Signed/Unsigned Multiplier

For the partial products of PQ_1 and PQ_{N-2} for unsigned multiplier, an extra bit 0 is padded before the MSB. Figure 2-24 shows the circuit realization of the combined partial products from PP'_1 to PP'_{N-2}.

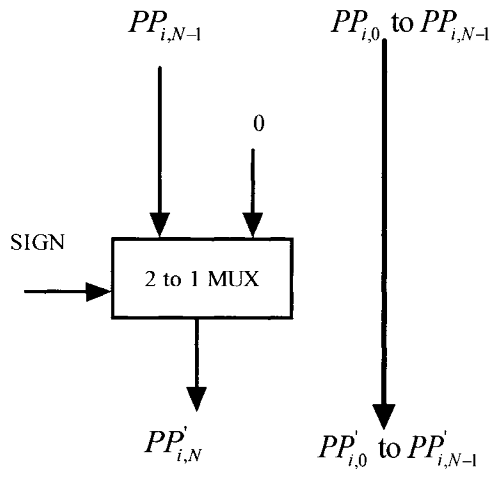


Figure 2-24. Circuit of the Partial Products from PP'_1 to PP'_{N-2} for Signed/Unsigned Multiplier

Figure 2-25 shows a unified signed/unsigned multiplier with the control signal *SIGN*.

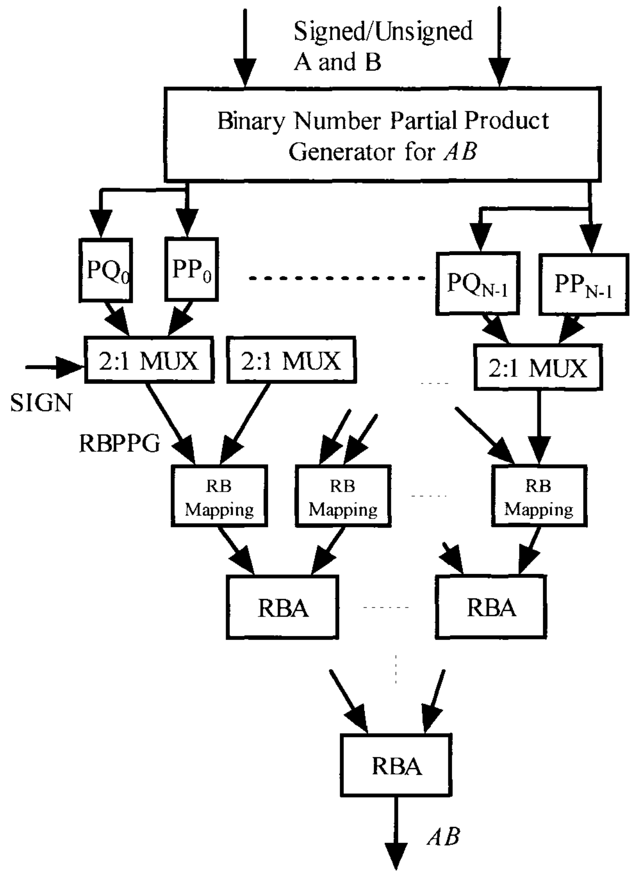


Figure 2-25. A Unified Sign/Unsigned Multiplier

Both the signed and unsigned multiplier have the same structure of the RB adder tree. The partial products are controlled to switch between signed and unsigned multiplication. Then the combined binary partial products are mapped to RB partial products and added using the RB adder tree to compute the signed/unsigned RB product.

2.4.2 Unified Signed/Unsigned Multiplier With Booth Coding

For the unsigned multiplier for A and B,

$$\begin{aligned} A &= \sum_{i=0}^{N-1} a_i 2^i = -0 \times 2^N + a_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i = A' \\ B &= \sum_{i=0}^{N-1} b_i 2^i = b_{N-1} 2^N - b_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} b_i 2^i = b_{N-1} 2^N + B' \end{aligned} \tag{2.77}$$

where A' and B' are 2's-complement binary numbers.

Thus, the product of AB is calculated as:

$$\begin{aligned} AB &= A'(b_{N-1} 2^N + B') \\ &= A' b_{N-1} 2^N + A'B' \\ &= \sum_{i=0}^{N-1} a_i b_{N-1} 2^i 2^N + A'B' \end{aligned} \tag{2.78}$$

The product of $A'B'$ using Booth encoding was previously discussed in Section 2.1.3. The extra $\sum_{i=0}^{N-1} a_i b_{N-1} 2^i 2^N$ value can be combined with the correction factors γ_i. The new correction factors are

$$\begin{aligned} \gamma_i^+ &= 0 \\ \gamma_i^- &= b_{i+1} \overline{b_i}\ \overline{b_{i-1}} + b_{i+1} \overline{b_i}\ b_{i-1} + b_{i+1} b_i\ \overline{b_{i-1}} \\ &= b_{i+1}\ \overline{b_i b_{i-1}} \\ \gamma_{i+1}^+ &= b_{i+3}\ \overline{b_{i+2} b_{i+1}} \\ \gamma_{i+1}^- &= 1 \end{aligned} \tag{2.79}$$

where $i = 0, 2, 4, \ldots \frac{N}{2} - 2$ and

$$\gamma_i^+ = a_i b_{N-1} \quad \gamma_i^- = 1 \tag{2.80}$$

where $i = N, N+1, \ldots, 2N-1$.

2.5 The Implementation of a Unified Signed/Unsigned Inner-Product Processor for $AB \pm CD$

The overall structure of a unified signed/unsigned inner-product processor is described for the computations of $AB + CD$ and $AB - CD$. In Section 2.2.3.1 and Figure

2-11. The $AB-CD$ implementation is developed using the structure for $AB+CD$ with inverters added in the RBA tree and a unified structure for $AB \pm CD$ is developed. In Section 2.4, a unified sign/unsigned multiplier was developed using RB representations. Define two control signals, SIGN=1/0 for signed/unsigned multiplication, and Real_Img=1/0 for the $AB-CD/AB+CD$ computation. Then the overall structure of a unified signed/unsigned inner-product processor for $AB \pm CD$ is shown in Figure 2-26, where the PQ_0 to PQ_{N-1} of AB and CD refers to the unsigned partial products discussed in Section 2.4.

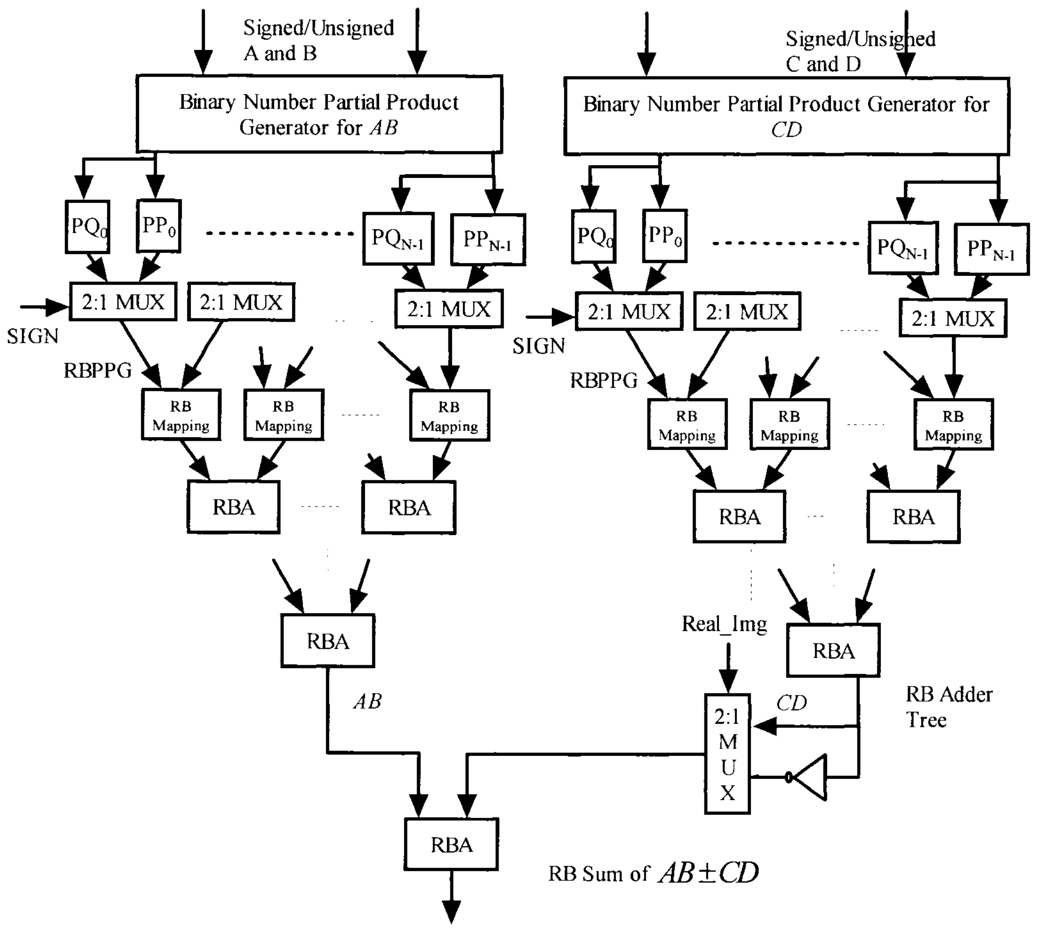


Figure 2-26. Unified Signed/Unsigned IP Processor for $AB \pm CD$

2.6 The Implementation of a Redundant Binary Multiplier

Currently, numerous floating-point unit designs incorporating a fast multiplier make iterative use of the multiplier for implementing fast algorithms for division, square root, and/or transcendental function computations by extended polynomial approximation [107]-[112]. If multipliers are to be used iteratively for RB computations, it is advantageous for the multiplier to accept redundant binary coded input directly, in addition to the initial 2's-complement numbers. A multiplier capable of accepting both 2's-complement and RB inputs avoids the excessive RB to 2's-complement delay. To our knowledge, no prior multiplier design exists with this capability. Recently a new floating point arithmetic unit was proposed [113]. A redundant number system is used to achieve IEEE compliant results. All operations in the arithmetic units are carried in redundant form with conversion back to the standard IEEE format performed only when an operand is written to memory. In [113], it is argued that the proposed floating point unit could achieve better performance across all of the required functions. In all these cases, a fast multiplier that can accept either 2's-complement or RB inputs is advantageous, i.e., the multiplicand and multiplier are both redundant binary numbers with the product produced in redundant binary form, as shown in Figure 2-27:

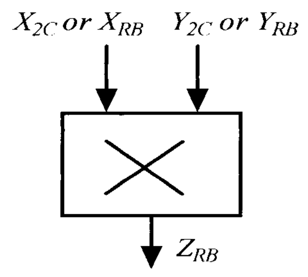


Figure 2-27. A RB Multiplier Diagram

2.6.1 Direct Implementation of Redundant Binary Multiplier

To implement the dual input multiplier, we first consider a RB multiplier. Figure 2-28 shows an example of RB multiplication.

Let A_{RB}=[1 -1 0 1], B_{RB}=[-1 1 0 -1] . Then, the product of A_{RB} and B_{RB} is computed as:

```
                     1    -1     0     1
                    -1     1     0    -1
                   ----------------------
                    -1     1     0    -1
               0     0     0     0
         1    -1     0     1
  -1     1     0    -1
-----------------------------------------
   0     0    -1    -1     0     0    -1
```

Figure 2-28. An Example of RB Multiplication

The RB partial product is generated according to Table 2-M, where α_i and β_i are the RB signed digits of A_{RB} and B_{RB}, respectively.

Table 2-M. RB Partial Product Generation

α_i	β_i	$\alpha_i\beta_i$
-1	-1	1
-1	0	0
-1	1	-1
0	-1	0
0	0	0
0	1	0
1	-1	-1
1	0	0
1	1	1

Encoding the RB digits 1=(11), 0=(01)=(10), -1=(00), Table 2-N shows the encoded RB partial products.

Table 2-N. Encoded RB Partial Product Generation

α_i	β_i	$\rho_i = \alpha_i \beta_i$
-1 (0,0)	-1 (0,0)	1 (1,1)
-1 (0,0)	0 (0,1) (1,0)	0 (0,1)(1,0)
-1 (0,0)	1 (1,1)	-1 (0,0)
0 (0,1) (1,0)	-1 (0,0)	0 (0,1)(1,0)
0 (0,1) (1,0)	0 (0,1)(1,0)	0 (0,1)(1,0)
0 (0,1) (1,0)	1 (1,1)	0 (0,1)(1,0)
1 (1,1)	-1 (0,0)	-1 (0,0)
1 (1,1)	0 (0,1) (1,0)	0 (0,1) (1,0)
1 (1,1)	1 (1,1)	1 (1,1)

From Table 2-N, Equation (2.81) is derived to find the encoded RB partial product:

$$\begin{aligned} \rho_i^- &= \overline{\alpha_i^-}\ \overline{\alpha_i^+}\ \overline{\beta_i^-}\ \overline{\beta_i^+} + \alpha_i^-\ \alpha_i^+\ \beta_i^-\ \beta_i^+ \\ \rho_i^+ &= \overline{\overline{\alpha_i^-}\ \overline{\alpha_i^+}\ \beta_i^-\ \beta_i^+ + \alpha_i^-\ \alpha_i^+\ \overline{\beta_i^-}\ \overline{\beta_i^+}} \end{aligned} \tag{2.81}$$

Another way to implement a RB multiplier is to use the RB inline inner-product processor core. In Section 2.4, and Section 2.5, the implementation of $A_0B_0+A_1B_1$ and $A_0B_0 - A_1B_1$ for both 2's-complement and unsigned numbers was discussed. Here the reuse of these cores is investigated to implement the RB multiplier.

2.6.2 Redundant Binary Multiplier Implementation Using Inner-Product Processor

Let $Z_{RB} = \sum_{i=0}^{N-1} \zeta_i 2^i$, where Z_{RB} is a RB number and ζ_i is encoded as two binary bits $(\zeta_i^+ \zeta_i^-)$ (refer to Table 1-B). In this research, the RB encoding is $\zeta_i = \zeta_i^+ - \overline{\zeta_i^-}$. Therefore,

$$\begin{aligned} Z_{RB} &= \sum_{i=0}^{N-1} \zeta_i 2^i = \sum_{i=0}^{N-1} (\zeta_i^+ - \overline{\zeta_i^-}) 2^i \\ &= \sum_{i=0}^{N-1} \zeta_i^+ 2^i - \sum_{i=0}^{N-1} \overline{\zeta_i^-} 2^i \end{aligned} \tag{2.82}$$

Let $Z^+ = \sum_{i=0}^{N-1} \zeta_i^+ 2^i$ and $Z^- = \sum_{i=0}^{N-1} \zeta_i^- 2^i$ then

$$Z_{RB} = Z^+ - \overline{Z^-} \tag{2.83}$$

where Z^+ and Z^- are unsigned binary numbers.

For two RB numbers, A_{RB} and B_{RB}, we have:

$$\begin{aligned} A_{RB} &= \sum_{i=0}^{N-1} \alpha_i 2^i,\ A^+ = \sum_{i=0}^{N-1} \alpha_i^+ 2^i \text{ and } A^- = \sum_{i=0}^{N-1} \alpha_i^- 2^i \\ B_{RB} &= \sum_{i=0}^{N-1} \beta_i 2^i,\ B^+ = \sum_{i=0}^{N-1} \beta_i^+ 2^i \text{ and } B^- = \sum_{i=0}^{N-1} \beta_i^- 2^i \end{aligned} \tag{2.84}$$

Here α_i and β_i are encoded as $(\alpha_i^+\ \alpha_i^-)$ and $(\beta_i^+\ \beta_i^-)$, with $\alpha_i = \alpha_i^+ - \overline{\alpha_i^-}$ and $\beta_i = \beta_i^+ - \overline{\beta_i^-}$.

From Equation (2.83),

$$\begin{aligned} A_{RB} &= A^+ - \overline{A^-} \\ B_{RB} &= B^+ - \overline{B^-} \end{aligned} \tag{2.85}$$

so $A_{RB}B_{RB}$ becomes

$$\begin{aligned} A_{RB}B_{RB} &= (A^+ - \overline{A^-})(B^+ - \overline{B^-}) \\ &= (A^+B^+ - \overline{A^-}B^+) + (\overline{A^-}\ \overline{B^-} - A^+\overline{B^-}) \end{aligned} \tag{2.86}$$

Where A^+, A^-, B^+ and B^- are all unsigned binary numbers, so the product of RB A_{RB} and B_{RB} can be realized, using two unsigned binary numbers computations with $A^+B^+ - \overline{A^-}B^+$ and $\overline{A^-}\ \overline{B^-} - A^+\overline{B^-}$. The diagram is shown in Figure 2-29:

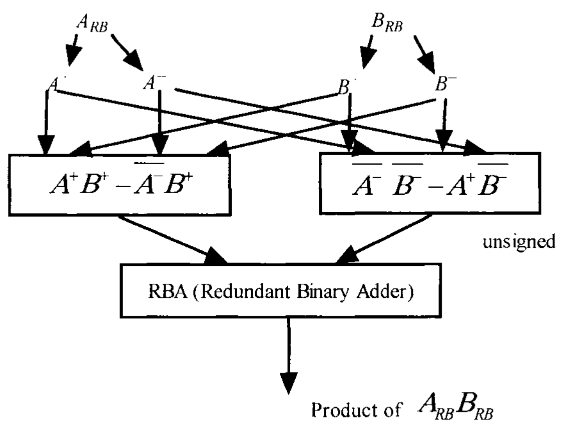

A_{RB}
B_{RB}
A^+
A^-
B^+
B^-
$A^+B^+ - \overline{A^-B^+}$
$\overline{A^-}\ \overline{B^-} - \overline{A^+B^-}$
unsigned
RBA (Redundant Binary Adder)
Product of $A_{RB}B_{RB}$

Figure 2-29. Implementation of RB Multiplier $A_{RB}B_{RB}$

Using this method to split the RB encoding bits and utilizing the unsigned feature of the multiplier (see Figure 2-26), the basic IP computing core will generate RB products with RB multiplicand and multiplier inputs.

2.7 Redundant Binary Inner-Product Computation

With the development of a RB multiplier in Section 2.6, an inner-product processor which can accept RB numbers input is easily designed. For example, Figure 2-30 shows the implementation structure to find the inner-product of $AB+X\Delta$, where A, B, X and Δ are RB numbers.

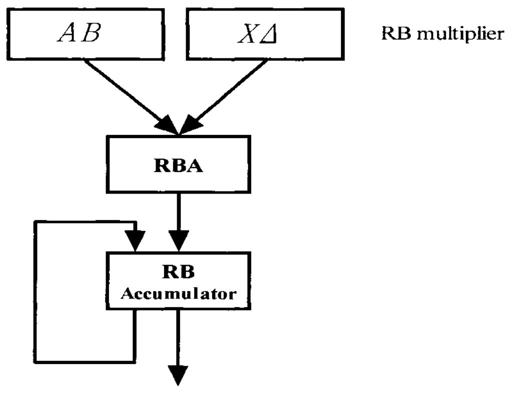


Figure 2-30. IP Implementation for RB Number $AB + X\Delta$

Chapter 3 Implementations of Division Method

3.1 Division Algorithm Review

The notation is used in the discussion here of division algorithms:

Z Dividend	$z_{2N-1}z_{2N-2}...z_1z_0$
D Divisor	$d_{N-1}d_{N-2}...d_1d_0$
Q Quotient	$q_{N-1}q_{N-2}...q_1q_0$
S Remainder $[Z-(D\times Q)]$	$s_{N-1}s_{N-2}...s_1s_0$

Division algorithms can generally be divided into the following classes: digit recurrence (restoring or non-restoring), functional iteration, table look-up and variable latency. The basis for these classes is the difference in the hardware operations used in their implementations, such as multiplication, subtraction, and table look-up. Many practical division algorithms are not pure forms of a particular class but rather are combinations of multiple classes. For example, a high performance algorithm may use table look-up to gain an initial approximation of the reciprocal, then use a function iteration algorithm to converge quadratically to the quotient. Table look-up may be impractical for general applications. The division method of table look-up requires a large RAM size for longer divisor size. The size of RAM increases exponentially with the word length of the divisor. The variable latency method results in a complex design for the control circuit and requires an asynchronous design method. The latency of variable latency division method depends on the value of the divisor. For different values of divisor, the latency is different. The two most popular division methods are digital recurrence and functional iteration.

Digit recurrence is the oldest class of high-speed division algorithms and, as a result, a significant quantity of literature exists proposing digit recurrence algorithms, implementations, and techniques. The most common implementation of digit recurrence division in modern processors was named *SRT* division by Freiman [81], taking its name from the initials of Sweeney, Robertson, and Tocher, who developed the algorithm independently at approximately the same time. Atkins [78] did fundamental research on division by digit recurrence, which was the first major analysis of SRT algorithms. Tan [89] derived and presented the theory of high-radix SRT division and an analytic method of implementing SRT look-up tables. Ercegovac and Lang [79] presented a comprehensive treatment of division by digit recurrence. Kuninobu [83], Aoki [77], and Srinivas [88] investigated the digit-recurrence division method with the redundant binary representation of the remainders. Basically the equation of the digit-recurrence division method in radix-(r) is:

$$s_j = rs_{j-1} - q_{N-j}(r^N D) \tag{3.1}$$

Digit recurrence algorithms deal with how to represent the remainder and quotient, how to choose the quotient, and choice of radix. Convergence of digit-recurrence is linear and has order *N*. A high performance quadratically convergent method, function iteration, was proposed which included the Goldschmidt [82] and Newton-Raphson [85] methods. Both methods first find the reciprocal and then use multiplication to compute the quotient. The functional iteration method is discussed below.

To compute the radio $Q = Z / D$, one can repeatedly multiply Z and D by a sequence of M multipliers $X_0, X_1 \ldots, X_{M-1}$:

$$Q = \frac{Z}{D} = \frac{ZX_0, X_1 \ldots, X_{M-1}}{DX_0, X_1 \ldots, X_{M-1}} \qquad (3.2)$$

If this is done in such a way that the denominator $DX_0, X_1 \ldots, X_{M-1}$ converges to 1, the numerator $QX_0, X_1 \ldots, X_{M-1}$ will converge to Q. This process does not yield a remainder, but the remainder S (if needed) can be computed, via an additional multiplication and a subtraction, using $S = Z - QD$.

To perform division based on the preceding idea, we face two questions:

1. How should we select the multipliers X_i so that the denominator does in fact converge to 1?
2. Given a selection rule for the multipliers X_i how many iterations are needed?

In the following discussion, we answer these questions in turn, but first, we formulate this process as a convergence computation.

Assume a bit-normalized fractional divisor, *D,* and dividend, *Z,* in [1/2 1). If this condition is not satisfied initially, it can be made to hold by appropriately shifting *Z* and/or *D*. The corresponding convergence computation is formulated as follows [82]:

$$\begin{array}{ll} D_{i+1} = D_i X_i & \text{Set } D_0 = D; \text{ make } D_M \text{ converge to 1} \\ Z_{i+1} = Z_i X_i & \text{Set } Z_0 = Z; \text{ obtain } Z/D = Q \approx Z_M \end{array} \qquad (3.3)$$

We now answer the first question posed above by selecting,

$$X_i = 2 - D_i \qquad (3.4)$$

This choice transforms the recurrence equations into:

$$\begin{array}{ll} D_{i+1} = D_i(2 - D_i) & \text{Set } D_0 = D; \text{ iterate until } D_M \approx 1 \\ Z_{i+1} = Z_i(2 - D_i) & \text{Set } Z_0 = Z; \text{ obtain } Z/D = Q \approx Z_M \end{array} \qquad (3.5)$$

Thus, computing the functions *f* and *g* consists of determining the 2's-complement of D_i and two multiplications by the result $2 - D_i$.

Now to address the second question: How quickly does D_i converge to 1? In other words, how many multiplications are required to perform division? Noting that

$$D_{i+1} = D_i(2 - D_i) = 1 - (1 - D_i)^2 \tag{3.6}$$

It is concluded that [82]:

$$1 - D_{i+1} = (1 - D_i)^2 \tag{3.7}$$

Thus, if D_i is already close to 1 (i.e. $1 - D_i \le \varepsilon$), D_{i+1} will be even closer to 1 (i.e. $1 - D_{i+1} \le \varepsilon^2$). This property is known as *quadratic convergence* and leads to a logarithmic number, *M*, of iterations to complete the process.

Another way to compute $Q = Z / D$ is to first find $1/D$ and then multiply the result by *Z*. If several divisions by the same divisor *D* need to be performed, this method [85] is particularly efficient. One method for computing $1/D$ is based on the Newton-Raphson iteration to determine a root of *f(x)=0*. We start with some initial estimate X_0 for the root and then iteratively refine the estimate using the recurrence:

$$X_{i+1} = X_i - \frac{f(X_i)}{f'(X_i)} \tag{3.8}$$

where $f'(X_i)$ is the derivative of *f(x)*. To apply the Newton-Raphson method to reciprocation, we use *f(x)=1/x-d* which has a root at *x=1/d*. Then $f'(x) = -1/x^2$, leading to the recurrence.

$$X_{i+1} = X_i(2 - X_i D) \tag{3.9}$$

Computationally, two multiplications and a 2's-complement step are required per iteration.

Let $\delta_i = 1/D - X_i$ be the error at the ith iteration. Then:

$$\delta_{i+1} = 1/D - X_{i+1} = 1/D - X_i(2 - X_i D) = D(1/D - X_i)^2 = D(\delta_i)^2 \tag{3.10}$$

Since $D<1$, we have $\delta_{i+1} < (\delta_i)^2$, thus this functional iteration based upon Newton-Raphson converges quadratically.

3.2 *Further Studies of the Goldschmidt and Newton-Raphson Methods*

In this section the algorithm of the Goldschmidt and the Newton-Raphson method are compared and studied. We show that these two methods are theoretically equivalent, but are often treated separately in the literature. Further studies of the Goldschmidt method are presented. Next, the RB inner-product processor core is investigated for performing the division computations for both real and complex numbers. We show how to control and/or reconfigure the RB inner-product processor to provide high-performance division.

3.2.1 Comparison of the Goldschmidt and Newton-Raphson Methods

For the initial divisor, D, and dividend, Z, the Goldschmidt iteration equations are:

$$\begin{aligned} &D_{i+1} = D_i(2 - D_i), \text{ Set } D_0 = D; \text{ iterate until } D_M \approx 1 \\ &Z_{i+1} = Z_i(2 - D_i), \text{ Set } Z_0 = Z; \text{ obtain } Z/D = Q \approx Z_M \end{aligned} \tag{3.11}$$

where D_i is the iterated divisor, Z_i is the iterated dividend, and Q is the quotient.

For the Newton-Raphson method, the iteration equations are:

$$X_{i+1} = X_i(2 - X_i D) \tag{3.12}$$

where X_i is the approximate reciprocal of *D*.

Multiplying *Z* on Equation (3.12) on both sides, we have:

$$ZX_{i+1} = ZX_i(2 - X_i D) \tag{3.13}$$

Comparing Equations (3.13) with (3.11), we notice that in Equation (3.13), ZX_i is the approximate quotient which is gradually close to *Z/D* after each iteration. Therefore,

$$Z_i = ZX_i \tag{3.14}$$

Then Equation (3.13) becomes:

$$Z_{i+1} = ZX_{i+1} = ZX_i(2 - X_i D) = Z_i(2 - X_i D) \tag{3.15}$$

If we define $D_i = X_i D$, then Equation (3.15) is:

$$Z_{i+1} = Z_i(2 - X_i D) = Z_i(2 - D_i) \tag{3.16}$$

Note that,

$$D_{i+1} = X_{i+1}D = X_i D(2 - X_i D) = D_i(2 - D_i) \tag{3.17}$$

and

$$D_0 = X_0 D, \quad Z_0 = ZX_0 \tag{3.18}$$

Letting $X_0 = 1$, then

$$D_0 = X_0 D = D \quad Z_0 = ZX_0 = Z \tag{3.19}$$

Under the condition of $X_0 = 1$, Equations (3.12) and (3.13) are equivalent to Equation (3.11). However, from the standpoint of implementation, Goldschmidt and Newton-Raphson methods are different. For the Goldschmidt method, as shown in Figure 3-1, two parallel multiplications plus two complement operations are required. For the Newton-Raphson method, as shown in Figure 3-2, two sequential multiplications and one complement operation are required. Therefore, the critical time delay in the Newton-

Raphson method is two multiplications plus one addition, while for the Goldschmidt method, only one multiplication step (parallel multiplications) and one addition are necessary. As far as the implementation area is concerned, the Goldschmidt method needs only one extra complement operation to implement. From this, we conclude that there are performance advantages for using the Goldschmidt method rather than Newton-Raphson method.

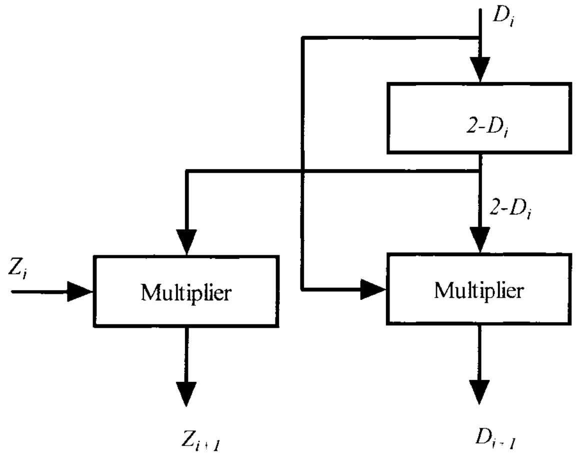


Figure 3-1. Goldschmidt Divisor Implementation [82]

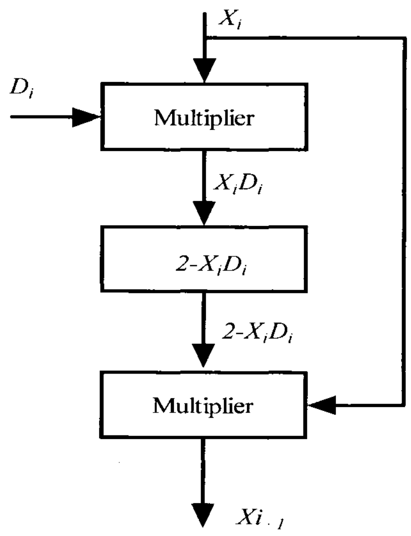


Figure 3-2. Newton-Raphson Divider Implementation [85]

3.2.2 Further Discussion of the Goldschmidt Method

In [85], it is claimed that the number of clock cycles for the Goldschmidt division method is $\log_2 N$, where N is the word length for dividend and divisor. Further study of our circuit implementations show that the actual number of clock cycles to achieve precise accuracy in the LSB of the quotient is $\log_2 N+1$. In the implementation of the Goldschidmt method, two additional guard bits are required to get the quotient precision of N bits.

After $M = \log_2 N$ iterations,

$$D_M = DX_0X_1 \cdots X_{M-2} = 1 - 2^{-N} \tag{3.20}$$

so,

$$X_0X_1 \cdots X_{M-2} = \frac{1-2^{-N}}{D} \tag{3.21}$$

and,

$$Z_M = ZX_0X_1 \cdots X_{M-1} = \frac{Z}{D}(1-2^{-N}) = Q(1-2^{-N}) \tag{3.22}$$

The error between the actual quotient and untruncated quotient Z_M is

$$\varepsilon_1 = |Q - Z_M| = |Q - Q(1-2^{-N})| = Q2^{-N} \tag{3.23}$$

For $Z < D$, we have $Q<1$, so $\varepsilon_1 < 2^{-N}$.

The approximate quotient is taken by truncating Z_M to N bits as $(Z_M)_T$, so

$$\varepsilon_2 = |Z_M - (Z_M)_T| < 2^{-N} \tag{3.24}$$

The error between the actual quotient and the computed one is:

$$\begin{aligned}\varepsilon &= |Q-(Z_M)_T| = |Q - Z_M + Z_M - (Z_M)_T| \\ &\le |Q - Z_M| + |Z_M - (Z_M)_T| \\ &= 2^{-N} + 2^{-N} = 2^{-N+1} \\ &\text{i.e. } \varepsilon \le 2^{-N+1}\end{aligned} \tag{3.25}$$

After $M = \log_2 N$ iterations, the precision of the quotient found from the Goldschimdt method is *N-1* bits. In order to reach the precision of *N* bits, $\log_2 N + 1$ iterations are needed.

If we want to achieve the computation error of the division using the RB IP processor to satisfy $\varepsilon \le 2^{-N}$, from Equations (3.23)(3.24) and (3.25), the following conditions must be satisfied as:

$$\varepsilon_1 \le 2^{-N-1} \text{ and } \varepsilon_2 \le 2^{-N-1} \tag{3.26}$$

That is, the dividend z_i and divisor d_i during the iteration must be truncated to *N*+1 bits instead of *N* bits, so one extra guard bit is required for the iteration.

In the same way, if the computation error of the division is required to meet $\varepsilon < 2^{-N}$, then two guard bits are required to compute the quotient using the RB IP processor.

3.2.3 Implementation of the Goldschmidt Division

Here we will explore how to implement the Goldschmidt division method using the RB inner-product structure. From Equation (3.5) in order to implement the high-speed divisor, all the intermediate dividend, divisor are in RB forms. Therefore, a RB-complement operation, $2 - A_{RB}$, similar to 2's-complement operation, and a RB multiplier must be developed. In Section 2.6, the RB multiplier is studied using the RB IP structure, so only the RB-complement operation requires development.

Let a real number A with precision N be represented in RB form, that is:

$$A = \sum_{i=-L}^{N-L-1} \alpha_i 2^i = \sum_{i=0}^{N-L-1} \alpha_i 2^i + \sum_{i=-L}^{-1} \alpha_i 2^i \tag{3.27}$$

where $\alpha_i = \{-1, 0, 1\}$, then

$$2 - A = 2 - \left(\sum_{i=0}^{N-L-1} \alpha_i 2^i + \sum_{i=-L}^{-1} \alpha_i 2^i\right) = (0010)_2 + \sum_{i=0}^{N-L-1} (-\alpha_i) 2^i + \sum_{i=-L}^{-1} (-\alpha_i) 2^i \tag{3.28}$$

Using the RB coding system, 1=(1 1), 0=(1,0)=(0,1) and –1=(0,0), notice that if α_i is encoded as $(\alpha_i^+ \ \alpha_i^-)$, then $-\alpha_i$ will be encoded as $(\overline{\alpha_i^+} \ \overline{\alpha_i^-})$. The implementation of 2-A_{RB} is shown in Figure 3-3:

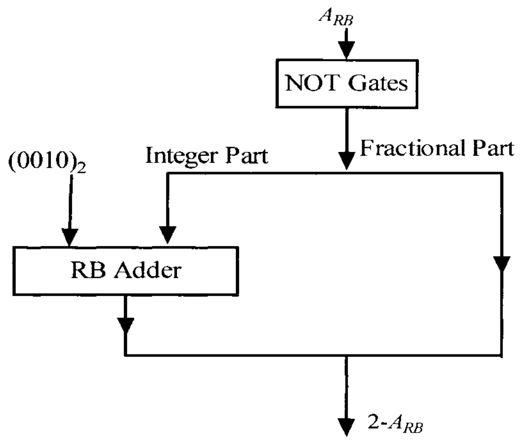


Figure 3-3. Implementation of 2-A_{RB}

3.3 Real Number Division Implementation

First, the dividend Z and the divider D are normalized to satisfy Z and $D \in [0.5 \ 1)$. For the normalization circuit, see references [116]-[118].

Then for the first iteration equation,

$$\begin{aligned} D_1 &= D_0(2 - D_0) \qquad && \text{Set } D_0 = D \\ Z_1 &= Z_0(2 - D_0) \qquad && \text{Set } Z_0 = Z \end{aligned} \tag{3.29}$$

where Z_0 and D_0 are both 2's-complement numbers. From Equation (1.9), Z_0 and D_0 can be mapped into RB digit numbers $(Z_0)_{RB}$ and $(D_0)_{RB}$ directly. Thus a RB multiplier which can accept the inputs $(D_0)_{RB}$, $(Z_0)_{RB}$, and $(2-D_0)_{RB}$ can realize the first iteration, as is shown in Figure 3-4.

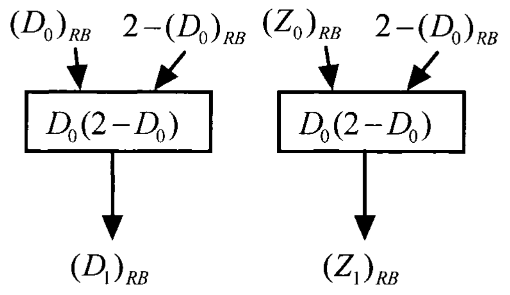


Figure 3-4. First Iteration Implementation of the Goldschmidt Division

Notice that the output Z_1 and D_1 of the first iteration are redundant binary numbers. Thus the successive iterations can be implemented as shown in Figure 3-5.

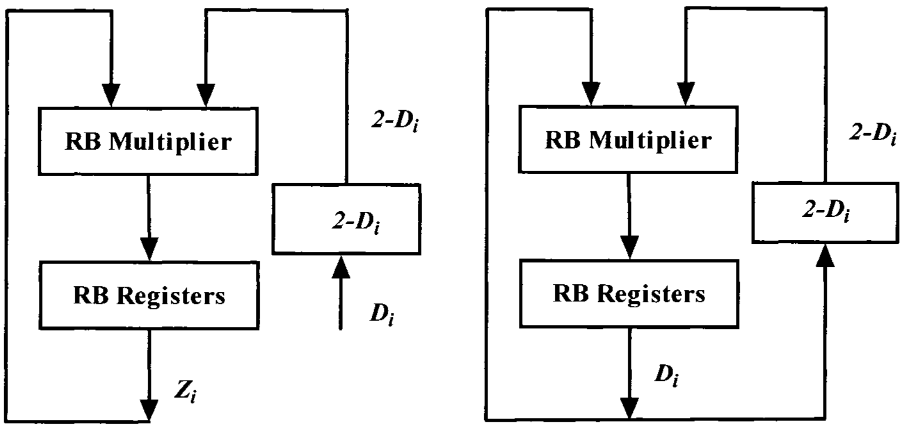


Figure 3-5. Implementation of Successive Iteration Computation for *Z* and *D*

After $\log_2 N+1$ iterations are carried out, where N is the number of bit precision, a RB-to-2's-complement converter is required to convert the quotient back to 2's-complement, if required. Four unified structures of $AB \pm CD$ are required to realize the real number division. Figure 3-6 shows the overall structure of the divider using RB IP processor.

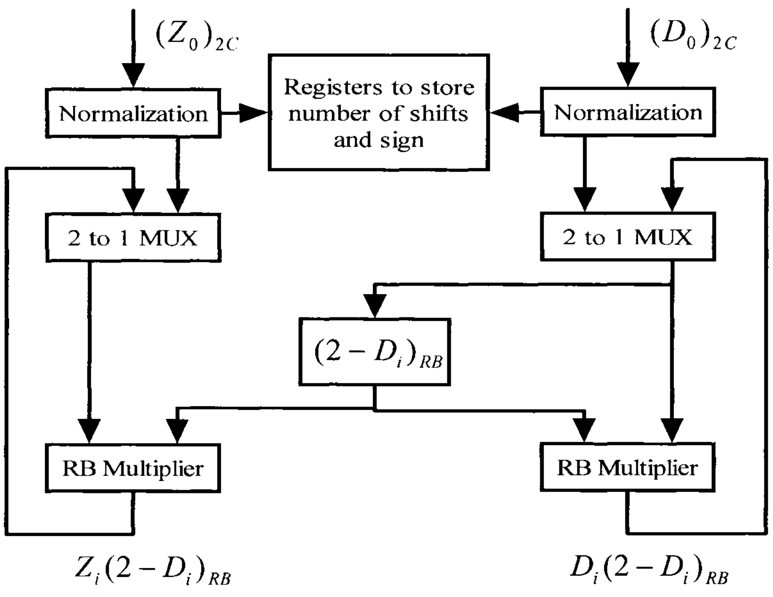


Figure 3-6. Overall Structure of Divider Using RB IP Processor

3.4 Comparison of the Implementations of Division

The implementation time required for division is compared between the Pentium Processor and the division implementation using a RB processor. Division implemented on the Pentium processor uses the SRT method. The 8-bit unsigned division implemented on the Pentium requires 17 clock cycles [85],[114],[115]. If the VLSI fabrication in [3] is implemented to realize the RB inner-product processor, then one iteration for the RB multiplication requires 8.8-2.4=6.4ns [3]. For 8-bit division, 4 iterations are required. The total time required for 8-bit division using the RB IP processor is:

$$(\log_2 8 + 1) \times 6.4 = 25.6ns \qquad (3.30)$$

If this division implementation result is compared to the Pentium processor, the equivalent clock cycle will be $25.6/17 = 1.505ns$, and is equivalent to 660Mhz clock frequency of Pentium processor.

3.5 Complex Number Division Implementation

To find the quotient of the complex number $\frac{A+jB}{C+jD}$,

$$Q=\frac{A+jB}{C+jD}=\frac{(A+jB)(C-jD)}{(C+jD)(C-jD)}=\frac{AC+BD}{C^2+D^2}+j\frac{BC-AD}{C^2+D^2} \tag{3.31}$$

For the implementation of a complex number divisor, $AC+BD$, $BC-AD$ and C^2+D^2 need to be computed. These computations can be realized by the unified signed/unsigned $AB \pm CD$ IP structure. For C^2+D^2, let $A=B$ and $C=D$. Notice that the outputs are in RB form for $AC+BD$, $BC-AD$ and C^2+D^2. For the Goldschmidt division method, both the dividend and the divisor need to be normalized. To normalize a RB digit, a RB-to-2's-complement converter is required to convert it back to 2's-complement. A normalization circuit is required to normalize $AC+BD$, $BC-AD$ and C^2+D^2 into [0.5 1). The diagram is shown in Figure 3-7.

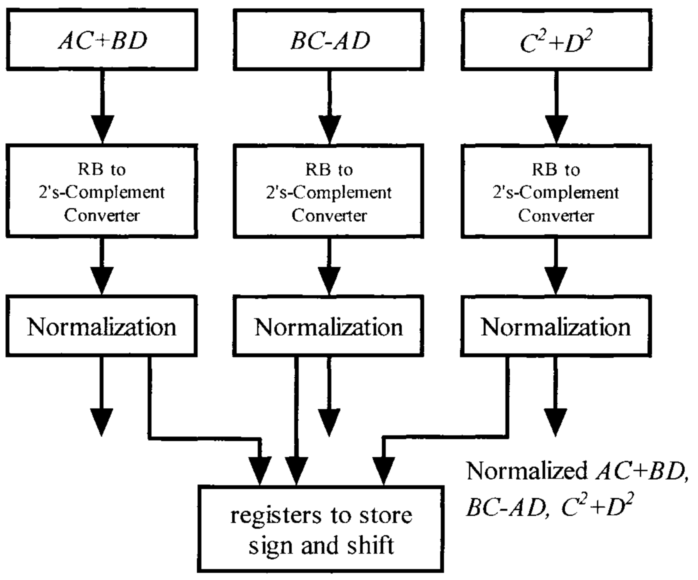

AC+BD
BC-AD
C²+D²
RB to 2's-Complement Converter
RB to 2's-Complement Converter
RB to 2's-Complement Converter
Normalization
Normalization
Normalization
registers to store sign and shift
Normalized AC+BD, BC-AD, C²+D²

Figure 3-7. Complex-Number Division Implementation Initial Process

Following the derivation of the normalized 2's-complement values of $AC+BD$, $BC-AD$ and C^2+D^2 in the first iteration, the real-number division implementation procedure is utilized to develop the quotient.. Six blocks of unified IP structure $AB \pm CD$ are needed to compute the complex number division since the implementation of divisors for both $\frac{AC+BD}{C^2+D^2}$ and $\frac{BC-AD}{C^2+D^2}$ can share the same computing structure for C^2+D^2. For normalization circuits, see [116]-[118].

Chapter 4 Computational Extensions

The inner-product structures described in Chapter 2 can be extended to provide a rich set of real, complex, RB and mixed real, complex and RB number computations. The inline partial product method [39] allows more extensions than the cross partial product scheme and will be used for illustrating added capabilities. Together with the basic inner-product operation, the computational capabilities afforded can be implemented using control signals or accomplished with circuit reconfiguration if configurable hardware is used. All of these extended computational capabilities are targeted for implementation in a Complex Arithmetic Signal Processor (CASP).

Referring to Figure 4-1, up to eight accumulator segments are required to support the following operations for $M = 8$.

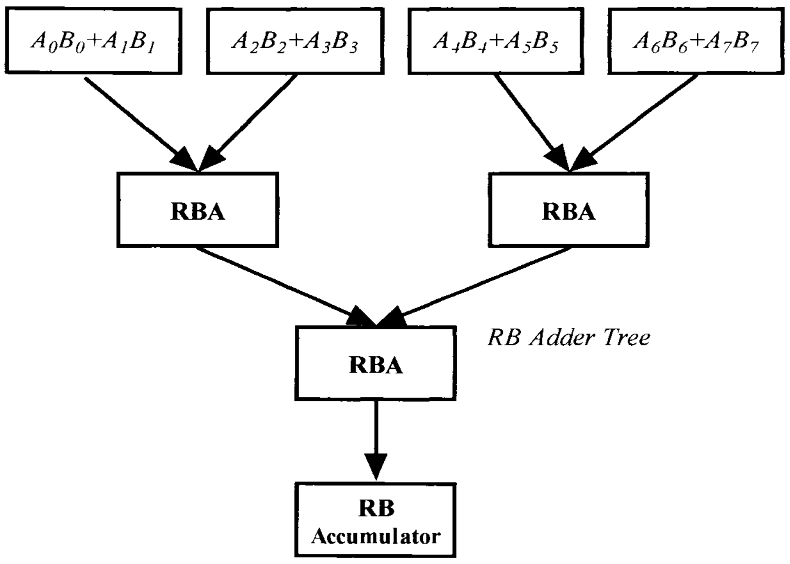


Figure 4-1. An Example of a Redundant Number Adder Tree

The structure in Figure 4-1 can support the following real number computations:

1. 8-element real number inner-product computation using a single RB accumulator segment.

2. Dual 4-element real number inner-product using two RB accumulator segments.
3. Quad 2-element real inner-product using four RB accumulator segments.
4. Eight parallel multipliers with or without eight accumulator segments.

The structure in Figure 4-1 can support the following complex number computations:

1. Single 2-element complex number inner-products using one RB accumulator segment.
2. Dual single complex number inner-products using four RB accumulator segments.
3. Two parallel complex number multipliers with or without two real and imaginary accumulator segments.

The structure in Figure 4-1 can support the following redundant binary number computations:

1. Single element redundant binary number inner-product computation using one accumulator segment.
2. Dual 2-element RB inner-product using two RB accumulator segments.
3. Four parallel RB multipliers using 4 RB accumulator segments.

Mixed real and complex number operations and mixed real/complex, 2's-complement/RB operations are also possible using the same 8-element IP structure. All of the extended computations are performed by bypassing some or all of the RB adder tree shown in Figure 4-1. The basic inner-product structure has the highest latency since the entire RB adder tree is utilized. When implementing one or more of the extended operations using control signals, design choices should be carefully considered since additional multiplexers are necessary for a multiple operation capability.

For a general purpose complex number DSP core, a key element of the design is the segmented accumulator and the ability to provide both overflow and saturation arithmetic. The design of the segmented accumulator and its associated final RBA for implementing the extended operations in a CASP device is beyond the scope of this dissertation and is the subject of continuing research.

4.1 Real-Number Computational Extensions

4.1.1 8-Element Real Number Inner-Product Computation

This structure is developed in Chapter 2 and provides the basic computational foundation for extended calculations. Refer to Figure 2-2 and Figure 4-1.

4.1.2 Dual 4-Element Real Number Inner-Product

Figure 4-2 shows the structure to perform dual 4-element real number inner-product using two RB accumulator segments. This calculation requires two accumulators, one for each inner-product.

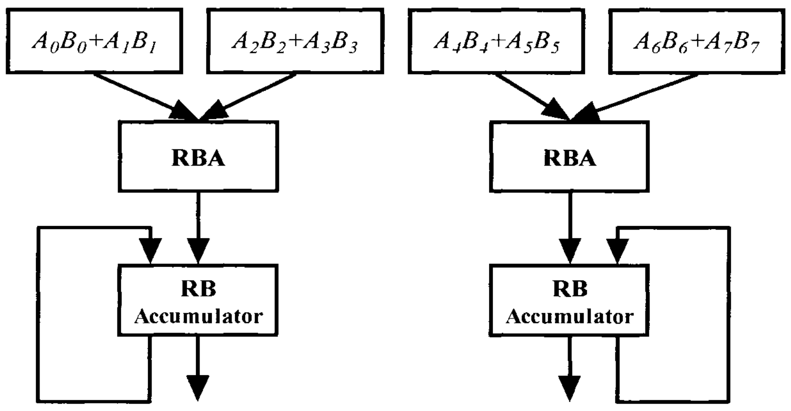


Figure 4-2. Dual 4-Element Real Number Inner-Product

4.1.3 Quad 2-Element Real Inner-Product Using Four Redundant Binary Accumulators

For this computation, the RB adder tree requires reconfiguration as shown in Figure 4-3. Here four RB accumulators are required:

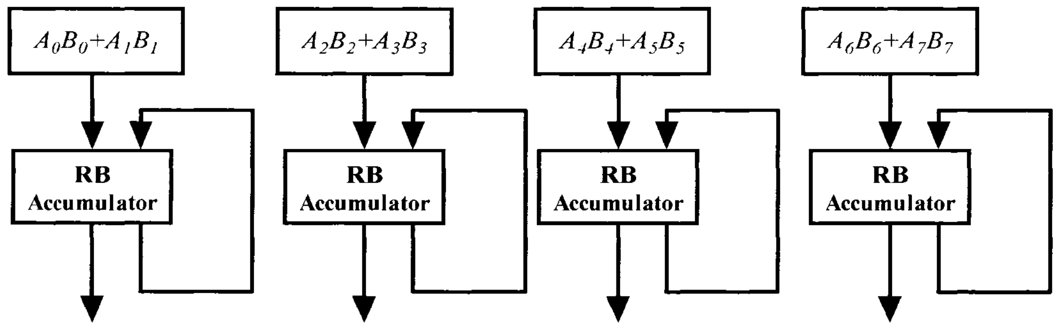


Figure 4-3. Quad 2-Element Real Number Inner-Product

4.1.4 Eight Parallel Multipliers Using 8 Redundant Binary Accumulators

For this computation, the RB adder tree needs to be controlled as shown in Figure 4-4. Here eight RB accumulators are required, if the structure is used for computing eight inner-products; otherwise, the accumulators are bypassed.

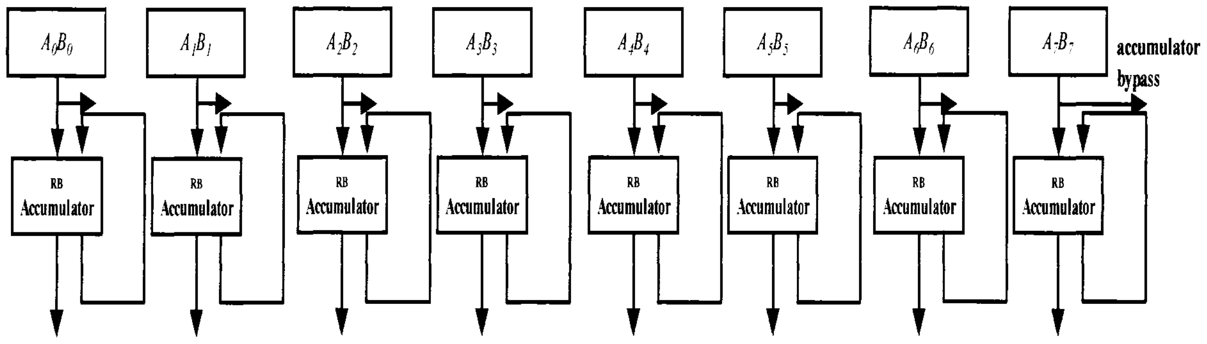


Figure 4-4. Eight Parallel Multipliers Using 8 RB Accumulators

4.2 Complex-Number Computational Extensions

Defining four complex numbers as: $C_0 = A_0 + jB_0$, $C_1 = A_1 + jB_1$, $C_2 = A_2 + jB_2$, $C_3 = A_3 + jB_3$, the computational extensions for complex number computations are depicted as follows.

4.2.1 Single 2-Element Complex Number Inner-Product Computation Using One Real/Imaginary Redundant Binary Accumulator

Figure 4-5 shows the structure of a single 2-element complex number inner-product using a RB accumulator segment for the real and imaginary parts separately.

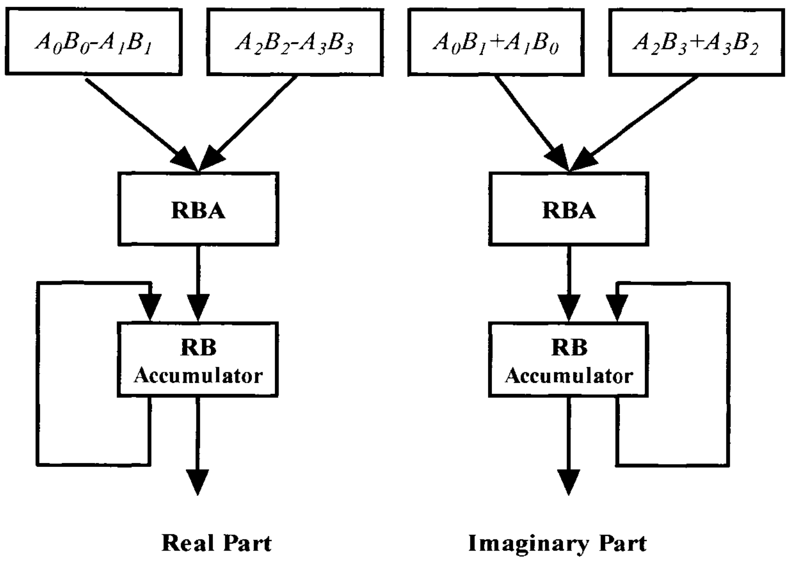


Figure 4-5. Single 2-Element Complex Number IP Using One Real/Imaginary RB Accumulator

4.2.2 Dual Single-element Complex Number Inner-Product Computation Using Four Redundant Binary Accumulators

Figure 4-6 shows the structure of dual 2-element complex number inner-products using four RB accumulators.

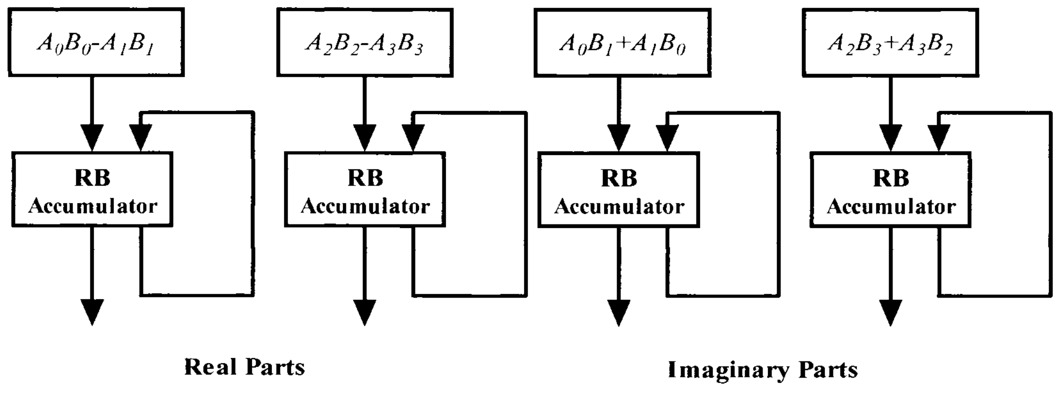


Figure 4-6. Dual 2-Element Complex Number Inner-Products Using Four RB Accumulators

4.2.3 Two Parallel Complex Number Multipliers

Figure 4-7 shows the structure of two parallel complex number multipliers using four RB accumulators.

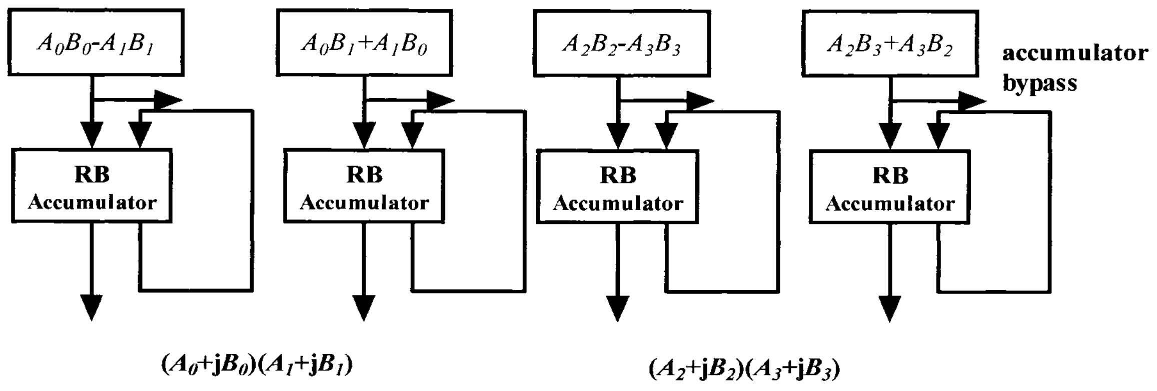


Figure 4-7. Two Parallel Complex Number Multipliers

4.3 Redundant Binary Number Computational Extensions

4.3.1 Single Element Redundant Binary Number inner-Product Computation

Figure 4-8 shows the structure of a 4-element redundant binary inner-product computation, where Φ_0 to Φ_3 and Γ_0 to Γ_3 are redundant binary numbers. The structure of the RB multiplier for $\Phi_0\Gamma_0$ to $\Phi_3\Gamma_3$ is discussed in Section 2.6.

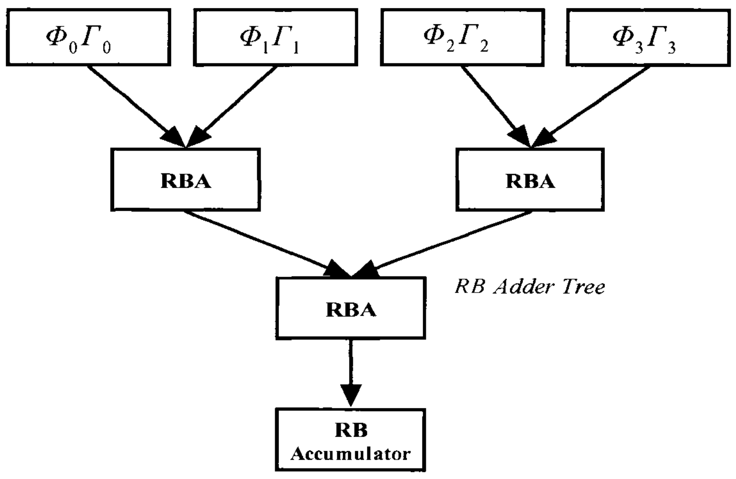


Figure 4-8. 4-Element Redundant Binary Inner-Product

4.3.2 Dual 2-Element RB Inner-Product plus Two Redundant Binary Accumulators

Figure 4-9 shows the structure of dual 2-element RB inner-product computation using two RB accumulators.

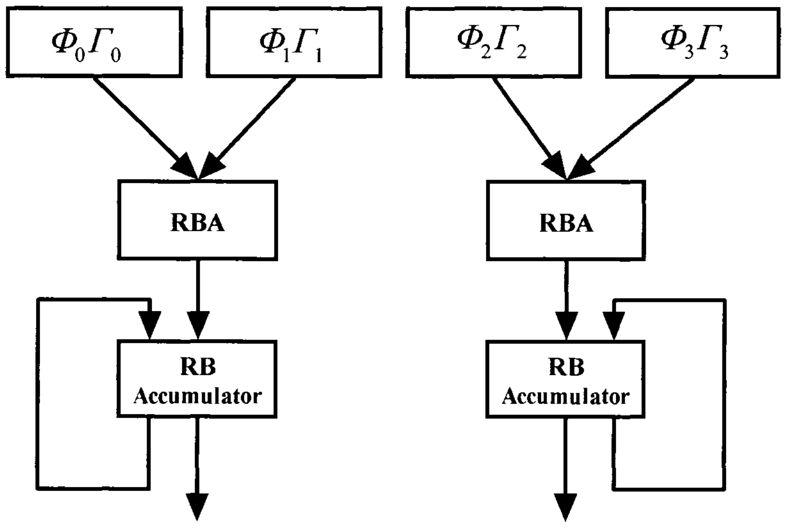


Figure 4-9. Dual 2-Element RB Inner-Product

4.3.3 Four Parallel Redundant Binary Multipliers Using Four Redundant Binary Accumulators

Figure 4-10 shows the structure of four parallel RB multipliers using 4 RB accumulators:

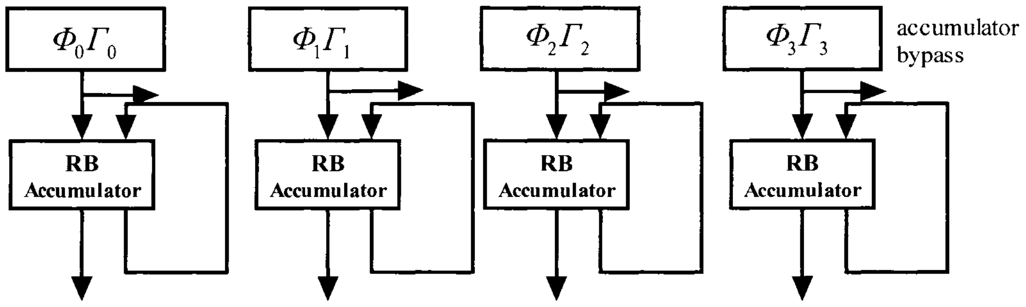


Figure 4-10. Four Parallel RB Multipliers Using 4 RB Accumulators

4.4 Pipeline Extensions

In this section, the possible pipeline design alternatives of the RB inner-product processor are investigated. The 0.5 μm CMOS time delay model from [3] for an 8-bit RB multiplier is used for the discussion and is shown as Table 4-A:

Table 4-A. Time Delay Model of RB Multiplier [3]

	Time Delay
2's-complement to RB Mapping	2-Nand gate=200ps=0.2ns
RBA (RB adder)	0.9 ns
RB to 2's-complement converter	1.6ns

Consider the 8-word 8-bit RB IP processor in Figure 4-1 and re-draw it as shown in Figure 4-11:

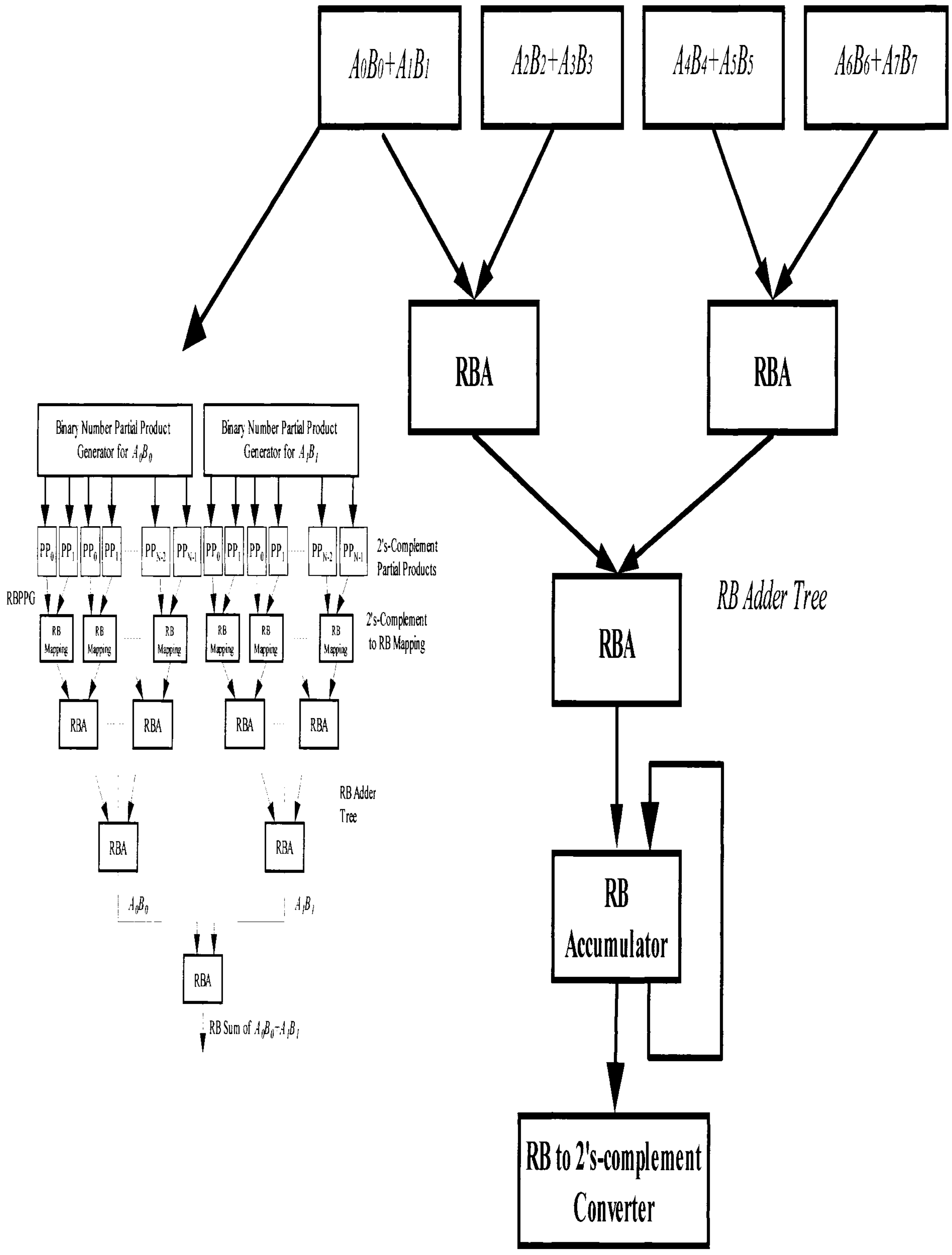

$A_0B_0+A_1B_1$
$A_2B_2+A_3B_3$
$A_4B_4+A_5B_5$
$A_6B_6+A_7B_7$
RBA
RBA
RB Adder Tree
RBA
RB Accumulator
RB to 2's-complement Converter
Binary Number Partial Product Generator for A_0B_0
Binary Number Partial Product Generator for A_1B_1
PP_0
PP_1
PP_{N-2}
PP_{N-1}
2's-Complement Partial Products
RBPPG
RB Mapping
2's-Complement to RB Mapping
RBA
RB Adder Tree
A_0B_0
A_1B_1
RB Sum of A_0B_0+A_1B_1

Figure 4-11. 8-Word 8-Bit RB IP Processor

The implementation of $A_0B_0 + A_1B_1$ consists of 2's-complement to RB mapping and 4 RB adders. If a two-stage pipeline structure is used, the resulting pipeline RB IP processor is depicted in Figure 4-12.

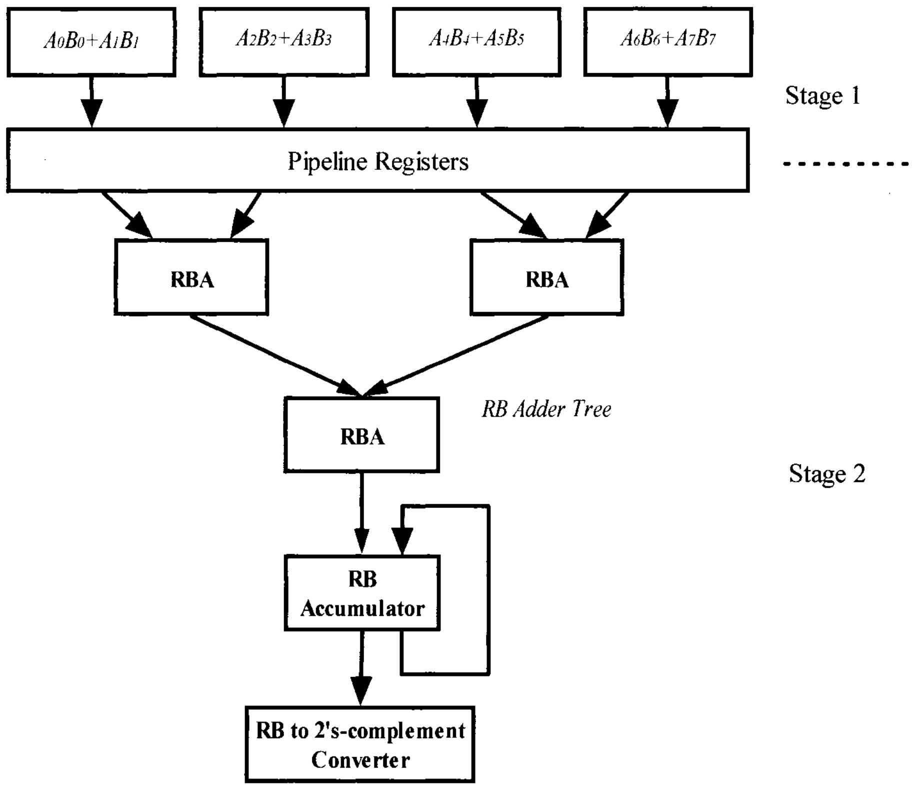


Figure 4-12. Two-Stage Pipelined RB IP Processor

The structure for stage 1 consists of the 2's-complement to RB mapping and 4 RB adders. The time delay is for the first stage is

$$0.2 + 4 \times 0.9 = 3.8ns \tag{4.1}$$

The structure for stage 2 consists of 3 RB adders and one RB to 2's-complement converter. The latency for the second stage is

$$3 \times 0.9 + 1.6 = 4.3ns \tag{4.2}$$

The difference of the time delay between these two stages is

$$4.3 - 3.8 = 0.5ns \qquad (4.3)$$

and can be considered as a balanced pipelined structure.

If a three-stage pipeline structure is employed, then the resulting structure is shown as Figure 4-13.

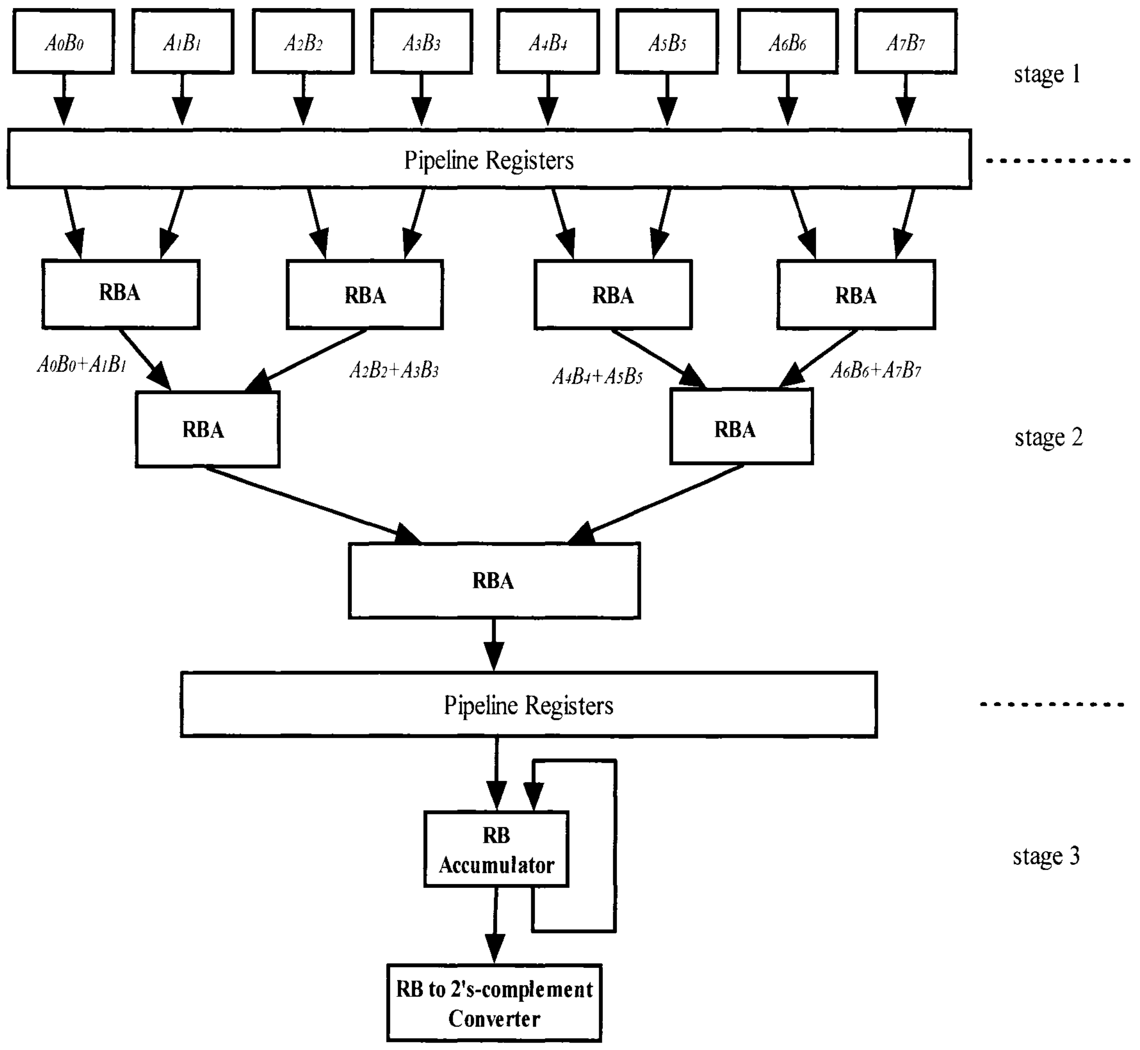


Figure 4-13. Three-Stage Pipelined RB IP Processor

The time delay for each stage is:

Stage 1

$$0.2 + 3 \times 0.9 = 2.9ns \qquad (4.4)$$

Stage 2

$$3 \times 0.9 = 2.7ns \tag{4.5}$$

Stage 3

$$0.9 + 1.6 = 2.5ns \tag{4.6}$$

The maximum difference in time delay among the three stages is 0.4 ns. This pipeline structure can be considered to be balanced.

Chapter 5 Redundant Binary to 2's-Complement Number Conversion

Since the inner-product processor produces results in a redundant binary form, a RB-NB (redundant binary to normal binary) converter may be required to provide a 2's-complement representation. Ruiz [5] proposed a carry-look-ahead RB-to-2's-complement converter which is similar to the structure of a carry-look-ahead adder. Rajashekhara [71] proposed a similar converter that is based upon a borrow-look-ahead structure. In his paper [7], Yen gave a novel definition of carry in the proposed RB to 2's-complement converter. An on-the-fly converter was discussed in [72] which converts serial RB inputs to a 2's-complement number. Choo [74] claimed a breakthrough of a new converter which has no latency proportional to the word length. However, according to the proof in [70], it is equivalent of 2's-complement addition in the conversion of redundant binary to 2's-complement. Choo's converter never works correctly. Ling [106] proposed a high-speed adder which is currently the fastest known binary adder. In this chapter, based on Yen's method [7] and Ling's adder scheme [106], we propose an improved RB-NB converter.

5.1 An Improved Redundant Binary to 2's-Complement Converter

Define a new variable carry c_i [7] as follows:

1) c_i=1 means that, for the current RB digit position *i*, there is at least one –1 to the right of the current bit position and no +1s between the –1 and the current position.

2) c_i=0 otherwise.

Table 5-A shows the conversion rules at stage *i*, where x_i is the redundant binary bit, c_i is the carry-in from next lower order position, s_i is the 2's-complement binary bit output, and c_{i+1} is the carry-out to the next higher bit position. Example 1 shows a conversion from RB to 2's-complement based on the foregoing rules.

Table 5-A. Conversion Rules in Stage *I* [7]

Input		Output	
Redundancy bit x_i	Carry in c_i	Binary bit s_i	Carry out c_{i+1}
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0
-1	0	1	1
-1	-1	0	1

Example 1: Letting the RB number $X_{RB} = [-1\ \ 1\ \ 0\ \ -1\ \ 0\ \ -1\ \ 1\ \ 0\ \ -1\ \ 0\ \ 0\ \ 0]$, then

$X_{RB} = [-1\ \ 1\ \ 0\ \ -1\ \ 0\ \ -1\ \ 1\ \ 0\ \ -1\ \ 0\ \ 0\ \ 0] = -1320$

$C\ = [\ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0]$

and

$S = [1\ \ 1\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0] = -1320$

For example, at bit position 0, $x_0 = 0$ and $c_0 = 0$; then according to Table 5-A, $s_0 = 0$ and $c_1 = 0$. For bit position 3, $x_3 = -1$, $c_3 = 0$; then according to Table 5-A, $s_3 = 1$ and $c_4 = 1$.

Using the encoding provided in Table 1-B to encode Table 5-A, the RB-NB conversion is shown in Table 5-B.

Table 5-B. Conversion Truth Table for RB-NB

Input		Output	
Redundancy bit $x_i = (x_i^- \ x_i^+)$	Carry in c_i	Binary bit s_i	Carry out c_{i+1}
0 (0 1) (1 0)	0	0	0
0 (0 1) (1 0)	1	1	1
1 (1 1)	0	1	0
1 (1 1)	1	0	0
-1 (0 0)	0	1	1
-1 (0 0)	1	0	1

According to this conversion truth table, we derive the following equation:

$$\begin{aligned} s_i &= \overline{c_i \oplus (x_i^- \oplus x_i^+)} \\ c_{i+1} &= \overline{x_i^- + x_i^+} + c_i \overline{x_i^- x_i^+} \end{aligned} \tag{5.1}$$

For the c_{i+1} equation above, define the signals, carry propagate, $p_i = \overline{x_i^- x_i^+}$, and carry generate, $g_i = \overline{x_i^- + x_i^+}$. Then:

$$c_{i+1} = g_i + c_i p_i \tag{5.2}$$

Unrolling the carry equations, we get:

$$\begin{aligned} c_4 &= g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3 \\ c_3 &= g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2 \\ c_2 &= g_1 + g_0 p_1 + c_0 p_0 p_1 \\ c_1 &= g_0 + c_0 p_0 \end{aligned} \tag{5.3}$$

Based on the Ling adder [106], a more efficient RB-NB converter can be designed.

Define the signal, carry transfer, $t_i = p_i + g_i$, (carry is not annihilated) and $h_i = c_i + c_{i-1}$, then

$$\begin{aligned} c_i &= g_{i-1} + c_{i-1}p_{i-1} = g_{i-1} + c_{i-1}t_{i-1} \\ &= g_{i-1} + g_{i-2}t_{i-1} + g_{i-3}t_{i-2}t_{i-1} + g_{i-4}t_{i-3}t_{i-2}t_{i-1} + c_{i-4}t_{i-4}t_{i-3}t_{i-2}t_{i-1} \end{aligned} \tag{5.4}$$

Ling's modification consists of propagating $h_i = c_i + c_{i-1}$ instead of c_i. To understand the following derivations, we note that g_{i-1} implies c_i (c_i if g_{i-1}=1), which in turn implies h_i.

$$\begin{aligned} p_i g_i &= \overline{x_i^- x_i^+}\ \overline{x_i^- + x_i^+} = (\overline{x_i^-} + \overline{x_i^+})(\overline{x_i^-}\ \overline{x_i^+}) \\ &= \overline{x_i^-}\ \overline{x_i^+} = \overline{x_i^- + x_i^+} = g_i \end{aligned}$$

$$\begin{aligned} c_i &= g_{i-1} + c_{i-1}p_{i-1} \\ &= g_{i-1}h_i + g_{i-1} + c_{i-1}p_{i-1} \qquad (g_{i-1}h_i + g_{i-1} = g_{i-1}) \\ &= g_{i-1}h_i + c_{i-1}p_{i-1} + p_{i-1}g_{i-1} \quad (p_{i-1}g_{i-1}{=}g_{i-1}) + c_{i-1}p_{i-1}p_{i-1}\text{(repeated term)} \\ &= g_{i-1}h_i + c_{i-1}p_{i-1} + p_{i-1}(g_{i-1} + c_{i-1}p_{i-1}) \\ &= g_{i-1}h_i + c_{i-1}p_{i-1} + c_i p_{i-1} \\ &= g_{i-1}h_i + p_{i-1}(c_{i-1} + c_i) \\ &= g_{i-1}h_i + p_{i-1}h_i = h_i t_{i-1} \end{aligned}$$

$$\begin{aligned} h_i &= c_i + c_{i-1} = (g_{i-1} + c_{i-1}p_{i-1}) + c_{i-1} \\ &= g_{i-1} + c_{i-1} \\ &= g_{i-1} + h_{i-1}t_{i-2} \end{aligned}$$

Unrolling the recurrence for h_i, we get:

$$\begin{aligned} h_i &= g_{i-1} + t_{i-2}h_{i-1} = g_{i-1} + t_{i-2}(g_{i-2} + h_{i-2}t_{i-3}) \\ &= g_{i-1} + g_{i-2} + h_{i-2}t_{i-2}t_{i-3} \ \{\text{Since } t_{i-2}g_{i-2} = g_{i-2}\} \\ &= g_{i-1} + g_{i-2} + g_{i-3}t_{i-3}t_{i-2} + h_{i-3}t_{i-4}t_{i-3}t_{i-2} \\ &= g_{i-1} + g_{i-2} + g_{i-3}t_{i-2} + g_{i-4}t_{i-3}t_{i-2} + h_{i-4}t_{i-5}t_{i-4}t_{i-3}t_{i-2} \end{aligned}$$

$$\begin{aligned} h_1 &= g_0 + t_{-1}h_0 \\ h_2 &= g_1 + g_0 + h_0 t_0 t_{-1} \\ h_3 &= g_2 + g_1 + g_0 t_1 + h_0 t_{-1} t_0 t_1 \\ h_4 &= g_3 + g_2 + g_1 t_2 + g_0 t_1 t_2 + h_0 t_{-1} t_0 t_1 t_2 \end{aligned} \tag{5.5}$$

Now, the expression for the converter output is:

$$\begin{aligned} s_i &= \overline{c_i \oplus (x_i^- \oplus x_i^+)} \\ &= \overline{(h_i t_{i-1}) \oplus (x_i^- \oplus x_i^+)} \end{aligned} \tag{5.6}$$

where,

$$\begin{aligned} t_i = p_i + g_i &= \overline{x_i^- x_i^+} + \overline{x_i^- + x_i^+} = (\overline{x_i^-} + \overline{x_i^+}) + (\overline{x_i^-}\ \overline{x_i^+}) \\ &= \overline{x_i^-} + \overline{x_i^+} = \overline{x_i^- x_i^+} = p_i \end{aligned}$$

Here, h_0 and h_4 are the 4-bit converter's h_{in} and h_{out}, respectively. A carry network based on the preceding equations can be used in conjunction with 2-input NANDs, producing the t_i signals, 2-input NORs, producing g_i, and 3-input XNORs, producing the sum bits, to build a 4-bit binary RB-NB converter. Note that since h_4 does not affect the computation of the sum bits, it can be derived based on the simpler equation:

$$h_4 = g_3 + h_3 t_2, \tag{5.7}$$

with a slight speed penalty. The resulting carry network is depicted in Figure 5-1.

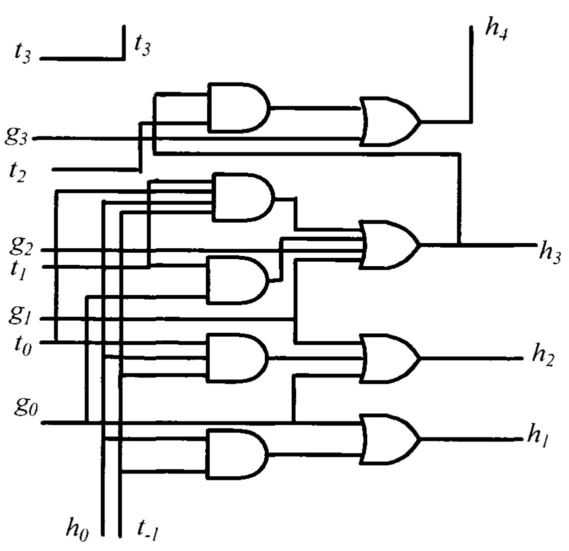


Figure 5-1. Four-Bit Carry-Lookahead RB-NB Converter (Similar to [85])

Compared to Equation (5.3), Equation (5.5) for h_4 contains only 12 terms, while in Equation (5.3), c_4 has 15 terms. The cost is that the sum is obtained by a slightly more complex expression in (5.6), as compared to Equation (5.1).

Given the design represented in Equations (5.5), the group "block generate" and "block propagate" signals can be derived as follows:

$$\begin{aligned} g[_{i,i+3}] &= g_{i+3} + g_{i+2} + g_{i+1}t_{i+2} + g_i t_{i+1} t_{i+2} \\ t[_{i,i+3}] &= t_{i-1} t_i t_{i+1} t_{i+2} \end{aligned} \qquad (5.8)$$

Figure 5-2 shows a schematic diagram of a 4-bit carry-lookahead block carry generator based on Ling's design.

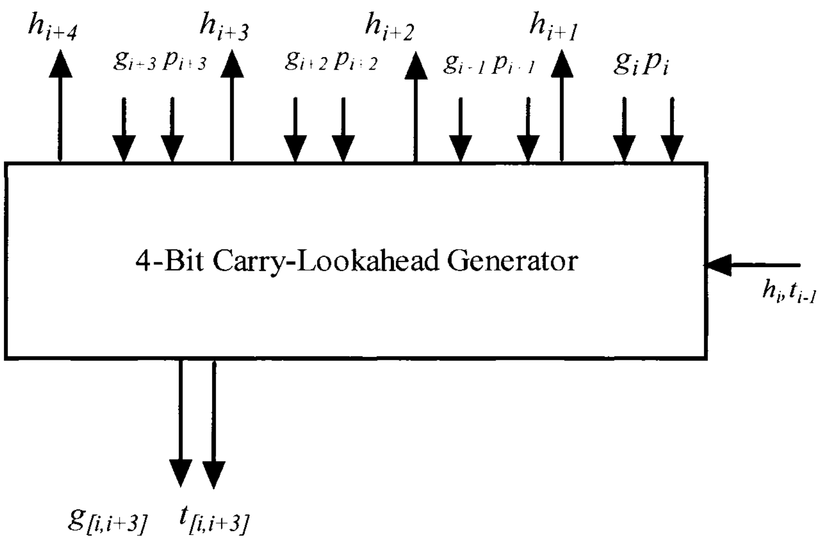


Figure 5-2. Diagram of a 4-Bit Carry-Lookahead RB-NB Carry Generator (Similar to [85])

Given the 4-bit carry-lookahead generator from Figure 5-1 and Figure 5-2, the construction of a multilevel-lookahead circuit is straightforward. For example, to construct a two-level 16-bit carry-lookahead RB-NB converter, we need four 4-bit RB-NB converters and a 4-bit carry-lookahead generator, connected together as shown in Figure 5-3.

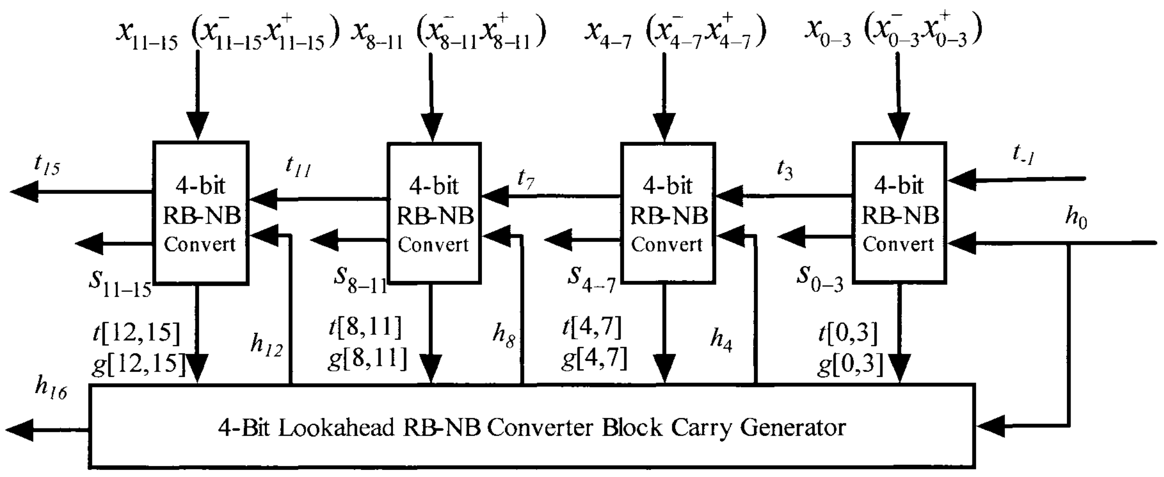


Figure 5-3. Two-Level 16-bit RB-NB Converter (Similar to [85])

5.2 Comparison Result

In this section, the novel converter is compared to a traditional carry-look-ahead based converter. A 4-bit converter is used for comparison. Assume only 2-inputs OR or AND gates can be used to build such a converter. For a traditional carry-look-ahead converter, the longest latency is defined by Equation (5.3). That is,

$$c_4 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3 + c_0 p_0 p_1 p_2 p_3$$

If only two-input gates are allowed, then it requires 14 gates to the realized c_4. The critical path delay is six gates level.

For the converter investigated here, the critical path delay is defined by Equation (5.5). That is,

$$h_4 = g_3 + g_2 + g_1 t_2 + g_0 t_1 t_2 + h_0 t_{-1} t_0 t_1 t_2$$

If only two-input gates are allowed, then this requires 10 gates to the realized h_4 and the critical path delay is only five gates level.

Chapter 6 Summary and Conclusions

Inner-product computations play a central role in digital signal processing, especially in the areas of digital filters, signal correlation, convolution, FFT, etc. Complex number arithmetic computation is a key arithmetic feature required in modern digital communication, radar systems and optical systems. Many algorithms based on convolutions, correlations, and complex number filters require complex number multiplication and high-speed inner-product computation. The overall motivation for this work is the design of a high-performance complex arithmetic processor (CASP) capable of offering novel extended inner-product operations.

The CASP design relies on the high-speed multiplication afforded by redundant binary techniques, while avoiding the relatively slow conversion back to 2's-complement numbers until a final 2's-complement result is necessary. Inherently, the CASP device provides intermediate register storage for redundant binary, as well as 2's-complement numbers. A new high-performance inner-product processor using redundant binary number representation is presented in this dissertation.

When the Booth coding technique is used, our proposed RB inner-product processor can significantly reduce the number of partial product to 25%. Also, it can be dynamically reconfigured/controlled to perform real, complex and redundant binary number computations such as parallel multiplications and inner-product computations. The extended computational capabilities of the RB IP processor are developed for real, complex, and redundant binary number or mixed computations. In Chapter 2, the structure of $A_0B_0 + A_1B_1$ for IP computation is studied. Two possible implementations , the inline and the cross partial product methods, are compared, with our inline method provides several advantages in speed and flexibility.

Complex number representations and arithmetic are also studied. Different complex radices such as radix-$(2j)$, radix-$(j$-1) and radix-$(\sqrt{2}j)$ are investigated and compared. It is found that the complex radices have no advantage in hardware implementations. The traditional redundant binary number representation is used to implement complex-number multiplication and inner-product processing. The new RB inline inner-product processor can be reconfigured/controlled to perform complex-number computations. The structures for $A_0B_0 + A_1B_1$ and $A_0B_0 - A_1B_1$ is developed and compared. The implementation of $A_0B_0 + A_1B_1$ can be easily controlled to perform the computation of $A_0B_0 - A_1B_1$. The complex number inner-product processor is investigated based upon this unified structure for $A_0B_0 \pm A_1B_1$. The implementation using the RB IP processor is compared with the TMS320C6XXX processor. This comparison shows there is some speed improvement for the RB IP core. Next, a unified signed/unsigned multiplier without and with Booth encoding is presented. Based upon the unified multiplier, the RB IP processor is further extended to realize a redundant binary multiplier that can accept both 2's-complement or RB inputs. The ability to accept RB inputs is essential for iterative calculations such as real and complex number division.

In Chapter 3, different division methods are reviewed. Two function iteration division methods, Newton-Raphson and Goldschimdt, are compared in detail. The theoretical equivalence of these two methods is shown. Further studies show that the Goldschmidt method is preferred over the Newton-Raphson method for efficient hardware implementation. Extension to the RB IP core are provided for performing

Goldschmidt division. The division implementation structure for both real and complex numbers is discussed using the same IP processor.

In Chapter 4, together with the basic inner-product operations, the computational capabilities afforded can be implemented using control signals or circuit reconfiguration if configurable hardware is used. These extended operations provide a rich set of computational capabilities targeted for implementation in a complex arithmetic signal processor (CASP). Various extensions such as IP computations, parallel multiplication of real, complex and redundant binary numbers are studied. Possible pipeline implementations of the RB IP core are investigated. A two-stage and three-stage pipeline structures are presented and the time delay model of these stages is studied. An improved RB to 2's-complement number converter is investigated in Chapter 5. This converter shows improvement in speed with a small increase in area.

Several areas of research are suggested. Further development of the IP core is required for the extended calculation capabilities, primarily dealing with the segmented accumulator and the requirements for flag setting based on arithmetic results for both saturation and overflow arithmetic. In addition, the IP processor can be developed to provide computational capabilities for square root, CORDIC, and other iterative functions.

Since the IP processor developed here serves as a core DSP computing capability, the overall architecture of the Complex Arithmetic Signal Processor (CASP) device requires extensive research to provide a dual numeric representation, i.e., 2's-complement and redundant binary. The CASP device should have a rich instruction set architecture that leverages the IP core for performing calculations for signed/unsigned, real/complex binary numbers, as well as intermediate calculations on redundant binary numbers. The CASP device will significantly enhance future applications requiring high-performance inner-product calculations.

APPENDIX
VHDL HARDWARE IMPLEMENTATION

For the information regarding VHDL hardware implementations, please contact the Office of Technology Development, University of Oklahoma.

660 Parrington Oval
Evans Hall, Room 201
Norman, Oklahoma 73019
Tel: (405) 325-3800
Fax: (405) 325-7162

REFERENCES

[1] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi, “A high-speed multiplier using a redundant binary adder tree,” *IEEE Journal of Solid-State Circuits*, vol. 22, no. 1, pp. 28-33, Feb. 1987.

[2] X. Huang, W. Liu, and B. W. Y. Wei, “A high-performance CMOS redundant binary multiplication-and-accumulation (MAC),” *IEEE Transactions on Circuits and Systems I.* vol. 41, no. 1, pp. 33-39, Jan. 1994.

[3] H. Makino, Y. Nakase, H. Suzuki, H. Morinaka, H. Shinohara, and K. Mashiko, “An 8.8-ns 54x54-bit multiplier with high speed redundant binary architecture, " *IEEE Journal of Solid-state Circuits*, vol. 31, no. 6, pp. 773-783, June 1996.

[4] N. Takagi, H. Yasuura, and S. Yajima, “High-speed VLSI multiplication algorithm with a redundant binary addition tree,” *IEEE Transactions on Computers*, vol. C-34, no. 9, pp. 789-796, Sept., 1985.

[5] G. A. Ruiz, “4bit CLA-based conversion from redundant to binary representation for CMOS simple and multi-output implementations,” *Electronics Letters*, vol. 35, no. 4, pp. 281-283, Feb. 1999.

[6] A. Herrfeld and Hentschke S., “Conversion of redundant binary into two's complement representations,” *Electronics Letters*, vol. 31, no. 14, pp. 1132-1133, July 1995.

[7] S. Yen, C. Laih, C. Chen, and J. Lee, “An efficient redundant-binary number to binary number converter, ” *IEEE Journal of Solid-State Circuits,* vol. 27, no. 1, pp. 109-112, Jan. 1992.

[8] M. O. Ahmad and D. V. Poornalah, “Design of an efficient VLSI inner-product processor for real-time DSP applications,” *IEEE Transactions on Circuits and Systems,* vol. 36, no. 2, pp. 324-329, Feb. 1989.

[9] E. Abdel-Raheem, A. Tawfik, M. Fahmi, and F. El-Guibaly, “New inner-product processor for FIR filter implementation,” in *Proceedings of IEEE Pacific RIM Conference Communication, Computer and Signal Processing,* Victoria, May 17-19, 1995, pp. 395-398.

[10] D. J. Soudris, V. Paliouras, T. Stouraitis, and C. E. Goutis, “A VLSI design methodology for RNS full adder-based inner product architectures,” *IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing,* vol. 44, no. 4, pp. 315-318, April 1997.

[11] M. N. Fahmi, F. El-Guibaly, S. Sunder, and D. J. Shpak, "Design of novel serial-parallel inner-product processors," in *1994 IEEE International Symposium on Circuits and Systems,* London, UK, May 30-June 2, 1994, pp. 55-58.

[12] S. Haynal and B. Parhami, "Arithmetic structures for inner-product and other computations based on a latency-free bit-serial multiplier design," in *33rd Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, Nov. 3-6, 1996, pp. 197-201.

[13] W. P. Burleson and L. L. Scharf, "VLSI design of inner-product computers using distributed arithmetic," in *1989 IEEE International Symposium on Circuits and Systems*, Portland, OR, May 8-11, 1989, pp. 158-161.

[14] A. S. Vega, P. S. R. Diniz, and A. C. Mesquita, "A modular distributed-arithmetic implementation of the inner product and its application to digital filters," *Journal of VLSI Signal Processing,* vol. 10, pp. 93-106, 1995.

[15] N. Kazakova, R. Sung, N. Durdle, M. Margala, and J. Lamoureux, "Fast and low-power inner product processor," in *The 2001 IEEE International Symposium on Circuits and Systems*, Sydney, Australia, May 6-9, 2001, vol. 4, pp. 646-649.

[16] R. Lin and S. Olariu, "A new buses scheme for fast inner-product computation," in *28th Asilomar Conference on Signals, Systems and Computers*, Monterey, CA, Oct. 30-Nov. 2, 1994, pp. 1402-1406.

[17] J. A. Starzyk and C. Chen, "A VLSI inner-product processor for real-time DSP applications," in *Proceedings of the 26th Southeastern Symposium on System Theory*, Athens, OH, March 20-22, 1994, pp. 219-223.

[18] C. C. Wang, C. J. Huang, and Y. P. Chen, "Design of an inner-product processor for hardware realization of multi-valued exponential bidirectional associative memory," *IEEE Transactions on Circuits and Systems II: Analog Digital Signal Processing,* vol. 47, no. 11, pp. 1271-1278, Nov. 2000.

[19] C. C. Wang, P. M. Lee and C. J. Huang, "Three alternative architectures of digital ratioed compressor design with application to inner-product processing," *IEE Proceedings on Computers and Digital Techniques,* vol. 147, no. 2, pp. 65-74, March 2000.

[20] D. C. M. Bilsby, R. L. Walke, and R. W. M. Smith, "Comparison of a programmable DSP and a FPGA for real-time multiscale convolution," in *Proceeding of the 1998 IEE Colloquium on High Performance Architectures for Real-Time Image Processing*, London, UK, Feb. 12, 1998, pp. 4/1-4/6.

[21] L. Breveglieri and L. Dadda, "A VLSI inner product macrocell," *IEEE Transactions on VLSI Systems*, vol. 6, no. 2, pp. 292-298, June 1998.

[22] E. Dujardin and O. Gay-bellile, "Software implementation of ADSL application with a convolution coprocessor," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP,* Seattle, WA, May 12-15, 1998, pp. 3053-3056.

[23] O. Gay-Bellile and E. Dujardin, "Architecture of a programmable FIR filter co-processor," in *1998 IEEE International Symposium on Circuits and Systems,* Pacific Grove, CA, May 31-June 3, 1998, vol. 5, pp. 433-436.

[24] R. Lin, "Reconfigurable parallel inner product processor architectures," *IEEE Transactions on VLSI Systems,* vol. 9, no. 2, pp. 261-272, April, 2001.

[25] R. Lin, A.S., Botha, K.E. Kerr, and G.A., Brown, "An inner product processor design using novel parallel counter circuits", in *The First IEEE Asia Pacific Conference on ASICs*, Seoul, Korea, Aug. 23-25, 1999, pp. 99-102.

[26] W. K. Luk and J. E. Vuillemin, "Recursive implementation of optimal time VLSI integer multiplier," in *Proceedings of VLSI 1983,* Amsterdam, Netherlands, 1983, pp. 155-168.

[27] R. Managuli, G. York, and Y. Kim, "An efficient convolution algorithm for VLIW mediaprocessors," in *Proceeding of SPIE—the International Society for Optical Engineering*, Jan. 1999, pp. 65-74.

[28] S. H. Baik, K.N. Han, and E. Yoon, "A 230MHz 8 tap programmable FIR filter using redundant binary number system," in *IEEE International Symposium on Circuits and Systems*, Orlando, FL, May 30-June 2, 1999, pp. 415-417.

[29] M. A. Sacristan, V. Rodellar, A. Diaz, V. Garcia, and P. Gomez, "A reusable inner product unit for DSP applications," in *25th Proceedings of EUROMICRO Conference*, 1999, pp. 209-213.

[30] G. Wang and M. Tull, "The implementation of an efficient and high-speed inner-product processor," in 35th *Asilomar Conference on Signals, Systems, and Computers*, Nov. 4-7, 2001, vol. 2, pp. 1362-1366.

[31] T. Aoki, Ohi, Y., and Higuchi, T., "Redundant complex number arithmetic for high-speed signal processing," in *IEEE Workshop on VLSI Signal Processing*, Sakai, Japan, Oct. 16-18, 1995, pp. 523-532.

[32] J. Buhler, M.A. Shokrollahi, and V. Stemann, "Fast and precise Fourier transforms," *IEEE Transactions on Information Theory*, vol. 46, no. 1, pp. 213-228, Jan 2000.

[33] D. Fu and A. N. Willson Jr., "A high-speed processor for rectangular-to-polar conversion with applications in digital communications," in 1999 *Global*

Telecommunications Conference, GLOBECOM '99, Janeireo, Brazil, Dec. 5-9, 1999, vol. 4, pp. 2172-2176.

[34] Y.P. Lee, L.G., M.J. Chen, and C.W. Ku, "A new design and implementation of 8×8 2-D DCT/IDCT," in *Workshop on VLSI Signal Processing*, San Francisco, CA, Oct. 30-Nov. 1, 1996, pp. 408-417.

[35] K. Sasayama, M. Okuno, and K. Habara, "Coherent optical transversal filter using silica-based single-mode waveguides," *Electronics Letters*, vol. 25, no. 22, pp. 1508-1509, Oct. 1989.

[36] K. Sasayama, M. Okuno, and K. Habara, "Coherent optical transversal filter using silica-based waveguides for high-speed signal processing," *Journal of Lightwave Technology,* vol. 9, no. 10, pp. 1225-1230, Oct 1991.

[37] M.A. Soderstrand, D.H. Chu, W. Chan, M. Lazkani, H.H. Loomis Jr., "Multi-rate bandpass filter bank implemented in QRNS complex arithmetic using parallel multiple DSP chips or ASICs," in *27th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, Nov. 1-3, 1993, pp. 801-806.

[38] S. Toledo, "On the communication complexity of the discrete Fourier transform," *IEEE Signal Processing Letters,* vol. 3, no. 6, pp. 171-172, June 1996.

[39] M. Tull, G. Wang, and M. Ozaydin, "High-speed complex number multiplier and inner-product processor," in *45th Midwest Symposium on Circuits and Systems*, August 4-7, 2002, vol. 3, pp. 640-643.

[40] A. Berkeman, V. Owall, and M. Torkelson, "A low logic depth complex multiplier using distributed arithmetic," *IEEE Journal of Solid-State Circuits,* vol. 35, no. 4, pp. 656-659, April 2000.

[41] S. He and M. Torkelson, "A complex array multiplier using distributed arithmetic," in *Proceedings of the IEEE 1996 Custom Integrated Circuits Conference*, San Diego, CA, May 5-8, 1996, pp. 71-74.

[42] M. Karlsson, M. Vesterbacka, and L. Wanhammar, "Design and implementation of a complex multiplier using distributed arithmetic, " in *1997 IEEE Workshop on Signal Processing*, Leicester, UK, Nov. 3-5, 1997, pp. 222-231.

[43] V. G. Oklobdzija, "An integrated multiplier for complex numbers," *Journal of VLSI Signal Processing,* vol. 7, pp. 213-222, 1994.

[44] V. G. Oklobdzija, D. Villeger, and T. Soulas, "Considerations for design of a complex multiplier," in *26th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, Oct. 26-28, 1992, pp. 366-370.

[45] A. P. Pascual, J. Valls, and M. M. Peiro, "Efficient complex number multipliers mapped on FPGA," in *6th IEEE International Conference on Electronics, Circuits and Systems*, Pafos, Cyprus, Sept. 5-8, 1999, pp. 1123-1126.

[46] A. P. Pascual, T. Sansaloni, and J. Valls, "FPGA based on-line complex-number multipliers," in *8th IEEE International Conference on Electronics, Circuits and Systems,* Malta, Sept. 2-5, 2001, vol. 3, pp. 1481-1481.

[47] K. Z. Pekmestzi, "Complex number multipliers," *IEE Proceedings,* vol. 136, no. 1, pp. 70-75, Jan. 1989.

[48] T. J. Sansalon, J. Valls, and K. K. Parhi, "FPGA-based digit-serial complex number multiplier accumulator," *2000 IEEE International Symposium on Circuits and Systems*, Geneva, Switzerland, May 28-31, 2000, pp. 585-588.

[49] K. W. Shin and H. W. Jeon, "High-speed complex-number multiplications based on redundant binary representation of partial products," *International Journal of Electronics,* vol. 87, no. 6, pp. 683-702, June 2000.

[50] K. W. Shin, B. S. Song, and K. Bacrania, "A 200-Mhz complex number multiplier using redundant binary arithmetic," *IEEE Journal of Solid-State Circuits,* vol. 33, no. 6, pp. 904-909, June 1998.

[51] A. Skavantzos and T. Stouraitis, "Decomposition of complex multipliers using polynomial encoding," *IEEE Transactions on Computers,* vol. 41, no. 10, pp. 1331-1333, Oct. 1992.

[52] T. Soulas, D. Villeger, and V. G. Oklobdzija, "An ASIC macro cell multiplier for complex numbers," in *4th European Conference on Design Automation with the European Event in ASIC Design*, 1993, pp. 589-593.

[53] T. Stouraitis and A. Skavantzos, "Multiplication of complex numbers encoded as polyniamials," *Journal of VLSI Signal Processing,* vol. 3, no. 4, pp. 319-328, April 1991.

[54] T. Stouraitis and A. Skavantzos, "Parallel decomposition of complex multipliers," in *22nd Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 1988, pp. 379-383.

[55] D. C. Blest and Jamil, T., "Efficient division in the binary representation of complex numbers," in *Southeastcon 2001 Proceedings IEEE,* Clemson, SC, March 30-April 1, 2001, pp. 188-195.

[56] Y. N. Chang and K. K. Parhi, "High-performance digit-serial complex-number multiplier-accumulator", in *Proceedings of the 1998 IEEE International Conference on Computer Design*, Austin, TX, Oct. 5-7, 1998, pp. 211-213.

[57] Y. N. Chang and K. K. Parhi, "High-performance digit-serial complex multiplier," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing II*, vol. 47, no. 6, pp. 570-572, June 2000.

[58] T. T. Dao, "Knuth's complex arithmetic with quaternary hardware," in *12th International Symposium on Multiple-Value Logic*, May 1982, pp. 94-98.

[59] J. Duprat, Y. Herreros, and S. Kla, "New redundant representations of complex number," *IEEE Transactions on Computers,* vol. 42, no. 7, pp. 817-823, July 1993.

[60] C. Fougny, "Parallel and on-line addition in negative base and some complex number systems," *Euro-Par '96 Parallel Processing*, pp. 175-182, 1996.

[61] W. J. Gilbert, "Complex numbers with three radix expansions," *Canadian Journal of Mathematics,* vol. 24, no. 6, pp.1335-1348, June 1982.

[62] T. Jamil, N. Holmes, and D. Blest, "Towards implementation of a binary number system for complex numbers," in *Proceedings of the IEEE Southeastcon 2000*, Nasville, TN, April 7-9, 2000, pp. 268-274.

[63] D. E. Knuth, "An imaginary number system," *Communications of the ACM,* vol. 3, no. 4, pp. 245-247, April 1960.

[64] D. E. Knuth, *The Art of Computer Programming, Vol. 2*: Addison-Wesley Publishing Company, 1969.

[65] I. Koren and Maliniak, Y., "On classes of positive, negative and imaginary radix number systems", *IEEE Transactions on Computers* vol. C-30, no. 5, pp. 312-317, May 1981.

[66] W. Penney, "A binary system for complex numbers," *Journal of the Associated for Computing Machinery,* vol. 12, no. 2, pp. 247-248, April, 1965.

[67] A. Slekys, "Design of complex number digital arithmetic units based on a modified bi-imaginary number system," Ph.D. dissertation, University of California at Los Angels, Los Angeles, CA, 1976.

[68] V. N. Stepanenko, "Computer arithmetic of complex numbers," *Cybernetics and Systems Analysis*, vol. 32, no. 4, pp. 585-591, July 1996.

[69] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 389-400, Sept. 1961.

[70] G. M. Blair, "The Equivalence of two's-complement addition and the conversion of redundant binary of twos-complement numbers," *IEEE Transactions on Circuits and*

Systems I: Fundamental Theory and Applications, vol. 45, no. 6, pp. 669-671, June 1998.

[71] T. N. Rajashekhara and A.S. Nale, "Conversion from signed-digit to radix complement representation," *International Journal of Electronics*, vol. 69, no. 6, pp. 717-721, Dec. 1990.

[72] M. D. Ercegovac and T. Lang, "On-the-fly conversion of redundant into conventional representations", *IEEE Transactions on Computers,* vol. C-36, no. 7, pp. 895-897, July 1987.

[73] M.P. Tull, G. Wang, and M. Ozaydin, "Method and apparatus for converting redundant binary numbers to two's-complement binary numbers", Disclosure 03NOR005, University of Oklahoma, July 2002.

[74] I. Choo and R. G. Deshmukh, "A novel conversion scheme from a redundant Binary to two's complement binary number for parallel architectures," in *SoutheastCon 2001 Proceedings of IEEE, 2001,* Clemson, SC, March 30-April 1, 2001, pp. 196 – 201.

[75] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Transactions on Computers,* vol. 46, no. 2, pp. 154-161, Feb. 1997.

[76] G. Wang, M. Ozaydin, and M. Tull, "High-performance divider using redundant binary representation," in *45th Midwest Symposium on Circuits and Systems*, August 4-7, 2002, vol. 3, pp. 640-643.

[77] T. Aoki, H. Tokoyo, and T. Higuchi, "High-radix parallel dividers for VLSI signal processing," in 4th *Workshop on VLSI Signal Processing,* Pacific Grove, CA, Oct. 30-Nov. 1, 1996, pp. 83-92.

[78] D. E. Atkins, "Higher-radix division using estimates of the divisor and partial remainder," *IEEE Transactions on Computers,* vol. C-17, no. 10, pp. 925-934, Oct. 1968.

[79] M. D. Ercegovac and T. Lang, *Division and Square Root: Digit-Recurrence Algorithms and Implementations*: Kluwer Academic, 1994.

[80] M. Flynn, "On division by functional iteration," *IEEE Transaction on Computers,* vol. 19, no. 8, Aug. 1970.

[81] C. V. Freiman, "Statistical analysis of certain binary division algorithms," *IRE Proceedings*, vol. 49, pp. 91-103, 1961.

[82] R. E. Goldschmidt, "Application of division by convergence," M.S. thesis, MIT, Cambridge, MA, June 1964.

[83] S. Kuninobu, T. Nishiyama, H. Edamatsu, T. Taniguchi, and N. Takagi, "Design of high speed MOS multiplier and divider using redundant binary representation," in *Proceedings of 8th Symposium on Computer Arithmetic*, 1987, pp. 80-86.

[84] S. F. Oberman and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on Computers,* vol. 46, no. 8, pp. 833-854, Aug. 1997.

[85] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*: Oxford University Press, 2000.

[86] J. E. Robertson, "A new class of digital division methods," *IRE Transactions on Electronic Computers,* vol. 7, pp. 218-222, Sep. 1958.

[87] E. M. Schwarz and M. J. Flynn, "Hardware starting approximation method and its application to the square root operation," *IEEE Transactions on Computers,* vol. 45, no. 12, pp. 1356-1369, Dec. 1996.

[88] H. R. Srinivas, "High speed computer arithmetic architectures," Ph.D. dissertation, University of Minnesota, Twin Cities, MN, 1994.

[89] K. G. Tan, "The theory and implementation of high-radix division," in *Proceeding of 4th IEEE Symposium of Computer Arithmetic,* 1978, pp. 154-163.

[90] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *Quarterly Journal of Mechanic Applied Mathematics,* vol. 11, part 3, pp. 386-383, 1958.

[91] Intel Architecture Tutorial, Intel Company, http://www.intel.com.

[92] "TMS320C6000 programmer's guide," http://dspvillage.ti.com/docs/catalog/resources/techdocs.jhtml?navSection=user_guides&familyId=132.

[93] S. D. Pezaris, "A 40-ns 17-bit by 17-bit array multipliers," *IEEE Transactions on Computers,* vol. 20, pp. 442-447, April 1971.

[94] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transaction on Electronic Computer,* vol. EC-13, pp. 14-17, Feb. 1964.

[95] IEEE Standard VHDL Language Reference Manual IEEE Std 1076-2002, http://ieeexplore.ieee.org/Xplore/DynWel.jsp

[96] A.D. Booth, "A Signed binary multiplication technique," *Quarterly Journal Mechanics and Applied Mathematics,* vol. 4, pt. 2, pp. 236-240, June 1951.

[97] T. Aoki, Amada, H., and T. Higuchi, "Real/Complex reconfigurable arithmetic using redundant complex number systems," in *13th IEEE Symposium on Computer Arithmetic—Arith'97,* Pacific Grove, CA, July 6-9, 1997, pp. 200-207.

[98] T. Aoki, Hoshi, K., and Higuchi, T., "Redundant complex arithmetic and its application to complex multiplier design," in *Proceeding of 29th IEEE International Symposium on Multiple Valued Logic,* Freiburg, Germany, May 20-22, 1999, pp. 200-207.

[99] R. McIlhenny and M. D. Ercegovac, "On-line algorithms for complex number arithmetic," in *32nd Asilomar Conference on Signals, Systems, Computers*, Pacific Grove, CA, Nov. 1-4, 1998, pp. 172-176.

[100] R. McIlhenny and M.D. Ercegovac, "On the design of on-line givens rotation," in *35th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, Nov. 4-7, 2001, pp. 160-164.

[101] R. E. Blahut, *Fast Algorithms For Digital Signal Processing*: Addison-Wesley, 1987.

[102] S. Zhoar, "Negative radix conversion," *IEEE Transactions on Computers,* vol. C-19, no. 3, pp. 222-226, March 1970.

[103] D.P. Agrawal, "Negabinary carry-look-ahead adder and fast multiplier," *Electronic Letters,* vol. 10, no. 15, pp. 312-313, July 1974.

[104] N.G.P. Satish, "Negative radix arithmetic and its applications," M.S. thesis, University of Nevada, Las Vegas, NV, 1973.

[105] G. Wang, M. P. Tull, and M. Ozaydin, "Binary conversion algorithms for the implementation of complex-radix numbers," presented at Proceedings of 2nd IEEE Electro/Information Technology, Oakland, MI, June 6-8, 2001.

[106] H. Ling, "High-speed binary adder," *IBM Journal of Research and Development,* vol. 25, no. 3, pp. 156-166, March 1981.

[107] W. S. Briggs and D. W. Matula, "A 17x69 bit multiply and add unit with redundant binary feedback and single cycle latency," in *Proceeding of IEEE 11th Symposium on Computer Arithmetic,* Windsor, Canada, June 29-July 2, 1993, pp. 163-170.

[108] P. Chai, *et al.*, "A 120 MFLOPS COS floating-point processor," in *Proceeding Of IEEE 1991 Custom Integrated Circuit Conference,* San Diego, CA, May 12-15, 1991, pp. 15.1/1-15.1/4.

[109] H. M. Darley, *et al.*, "Floating-point/integer processor with divide and square root functions," U.S. Patent 4878190, Oct. 1989.

[110] S. M. Quek, L. Hu, J. P. Prabhu, and F. A. Ware, "Apparatus for determining booth recoder input control signal," U.S. Patent 5280439, Jan. 1994.

[111] Y. Harata, Y. Nakamura, H. Nagase, M. Takigawa, and N. Takagi, "A high-speed multiplier using a redundant binary adder tree," *IEEE Journal of Solid-State Circuits,* vol. 22, no. 1, pp. 28-34, Feb. 1987.

[112] N. Takagi and S. Yajima, "On a fast iterative multiplication method by recoding intermediate product," in *Proceedings Of 36th National Convention of Information Science,* Kyoto University, Aug. 1987.

[113] H.A.H. Fahmy, A.A. Liddicoat, and M.J. Flynn, "Improving the effectiveness of floating point arithmetic," in *35th Asilomar Conference on Signals, Systems, and Computers,* Pacific Grove, CA, Nov. 4-7, 2001, pp. 875-879.

[114] IA-32 Intel Architecture Software Developer's Manual volume 2: Instruction Set Reference, http://developer.intel.com/design/pentium4/manuals/.

[115] Intel Pentium Family User's Manual volume 3: Architecture and Programming Manual, Intel Corporation, http://developer.intel.com/design/pentium4/manuals/.

[116] E. Antelo, M. Boo, J. D. Bruguera, and E. L. Zapata, "A novel design of a two operand normalization circuit", *IEEE Transactions on VLSI Systems,* vol. 6, no. 1, March 1998, pp. 173-176.

[117] V. G. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis", *IEEE Transactions on VLSI Systems*, vol. 2, no. 1, pp. 124-148, March 1994.

[118] V. G. Oklobdzija, "Algorithmic design of a hierarchical and modular leading zero detector circuit," *Electronic Letters,* vol. 29, no. 3, pp. 283-284, Feb. 1993.