UNIVERSITY OF CALIFORNIA

Los Angeles

Complex Number On-line Arithmetic for Reconfigurable Hardware: Algorithms, Implementations, and Applications

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Robert Dean McIlhenny

2002

The dissertation of Robert Dean McIlhenny is approved.

Nathaniel Grossman

Jason Cong

David Rennels

Miloš D. Ercegovac, Committee Chair

University of California, Los Angeles

2002

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

x	floating point number
x^*	complex conjugate of x
X	significand of x
e_x	exponent of x
$X[k]$	value of X after kth iteration
$\overline{X[k]}$	low-precision estimate of $X[k]$
x_k	kth most-significant digit of X in on-line form
$\overline{x_k}$	signed digit with value $-x_k$
X_R	real component of X
X_I	imaginary component of X
j	imaginary unit representing $\sqrt{-1}$
m	precision of a computation
r	radix of a computation
K	redundancy factor of a number system
L_d	lower bound for selection of value d
U_d	upper bound for selection of value d
δ	on-line delay
ρ	largest value of a symmetric digit set

LIST OF ACRONYMS

ASIC	Application-Specific Integrated Circuit
CLB	Configurable Logic Block
FPGA	Field Programmable Gate Array
LUT	Lookup Table
MMP	Minus-minus-plus Full Adder
MSDF	Most Significant Digit First
PPM	Plus-plus-minus Full Adder
RCNS	Redundant Complex Number System
$RCNS_{rj,\rho}$	Redundant Complex Number System of radix rj and largest digit ρ
VHDL	Very High Speed Integrated Circuit Hardware Description Language

ACKNOWLEDGEMENTS

I am grateful to my advisor, Professor Miloš D. Ercegovac, whose guidance and support encouraged me to see "the big picture" and complete this research. Special thanks also go to the members of my committee, Professors David Rennels, Jason Cong, and Nathaniel Grossman, for their constructive comments and participation.

Many people have helped me throughout the course of my study and my stay at UCLA. I would especially like to thank my office mates over the years, namely John Harding, Marianne Louie, Alex Tenca, George Mustafa, Chuck Fabian, John Pipan, and Zhijun Huang, who shared their space and time to help me. Special thanks go to Verra Morgan, who went the extra mile many times to help me get all the right forms filled out.

I am eternally grateful for my coworkers in Univesity Bible Fellowship, especially Pastor Paul Chin who helped me to have vision and hope to pursue a Ph.D., as well as my parents, my wife Sarah, and my daughter Esther, for their many prayers and encouragement that strengthened me in many ways.

Above all, I would like to thank my Heavenly Father, for His grace and mercy upon me to be an instrument for His glory.

So whether you eat or drink or whatever you do, do it all for the glory of God. – 1 Corinthians 10:31 (NIV)

VITA

September 25, 1970	Born, Redondo Beach, California
1993	B.S. Information and Computer Science University of California, Irvine Irvine, California
1996	M.S. Computer Science University of California, Los Angeles Los Angeles, California

PUBLICATIONS

R. McIlhenny and M.D. Ercegovac, *On Using 1-out-of-n Codes for (p,q) Counter Implementations*, 30th Asilomar Conference on Signals, Systems, and Computers, Nov. 1996.

R. McIlhenny and M.D. Ercegovac, *On the Implementation of a Three-operand Multiplier*, 31st Asilomar Conference on Signals, Systems, and Computers, Nov. 1997.

R. McIlhenny and M.D. Ercegovac, *On-line Algorithms for Complex Number Arithmetic*, 32nd Asilomar Conference on Signals, Systems, and Computers, Nov. 1998.

R. McIlhenny and M.D. Ercegovac, *On the Design of an On-line FFT Network for FPGA's*, 33rd Asilomar Conference on Signals, Systems, and Computers, Nov. 1999.

R. McIlhenny, Z. Huang, K. Wong, A. Schneider, and M.D. Ercegovac, *BigSky–a Tool for Mapping Numerically Intensive Computations onto Reconfigurable Hardware*, 34th Asilomar Conference on Signals, Systems, and Computers, Nov. 2000.

R. McIlhenny and M.D. Ercegovac, *On the Design of an On-line Complex Givens Rotation*, 35th Asilomar Conference on Signals, Systems, and Computers, Nov. 2001.

ABSTRACT OF THE DISSERTATION

Complex Number On-line Arithmetic for Reconfigurable Hardware:

Algorithms, Implementations, and Applications

by

Robert Dean McIlhenny

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2002

Professor Miloš D. Ercegovac, Chair

Complex number arithmetic plays an important role in various signal processing tasks, such as correlations, convolutions, and digital filtering. There is a need in such applications to exploit parallelism in implementing sequences of arithmetic expressions, as well as to reduce the bandwidth of the data path. Reconfigurable architectures are emerging as a viable technology for mapping numeric-intensive computations onto hardware. In order to provide efficient implementations that can accommodate changes with a minimal amount of redesign, the methodology for deriving algorithms and corresponding implementations must be flexible toward change.

In this dissertation, we present an efficient representation which treats the real and imaginary components as a unified number. We propose algorithms for various complex number on-line floating-point arithmetic operations. The algorithms

are translated into actual implementations mapped onto reconfigurable hardware. The implementations are applied toward computing the complex singular value decomposition of a matrix, with a significant reduction in cost compared to networks of real number on-line and parallel arithmetic approaches.

CHAPTER 1

Introduction

1.1 Motivation

Complex number arithmetic is encountered in various signal processing and scientific tasks such as complex orthogonal transformations, convolutions, correlations, and filtering [BMP88], which serve as time-critical components for radar, satellite, and digital modulation applications [WDC95]. Such applications require efficient representation and manipulation of complex numbers. In the usual representation, the real and imaginary components of a complex number are treated separately. Implementing a complex number algorithm is not efficient in general, since four operands have to be accessed with address computations, and complex number multiplication, for instance, requires the realization of four real multiplications. Traditionally, commercially available signal processors do not have special hardware capability for complex number arithmetic. Rather, what is proposed is a software decomposition of complex number arithmetic by elementary real operations. The resulting overhead due to control operations in the decomposed solution decreases the throughput of computations. Therefore, there is a need for efficient hardware implementations of elementary complex number operations.

Several papers have proposed efficient hardware approaches for complex number multipliers. Wei, et. al. [WDC95] describes the design of a complex number multiplier using radix 4 digits and a transformation from the common 4-multiply/2-add approach to a 3-multiply/5-add approach. Oklobdzija, et. al. [OVS94] presents an integrated parallel multiplier for complex numbers, in which a modified Wallace tree using 4:2 counters is shared among the real and imaginary parts for the summation of partial products. Shin, et. al. [SSB98] designed a 200-MHz complex number multiplier using redundant binary arithmetic, transforming a complex number multiplication into two redundant binary multiplications for the real and imaginary terms of the final product. Chang and Parhi [CP00] present a high-performance digit-serial complex multiplier based on a *real-imaginary-alternate* scheme. Distributed arithmetic has been applied by various authors for the design of a pipelined bit-serial complex number multiplier [HT95] and a complex number tree multiplier [BOT97].

Each of these approaches presents efficient complex number multiplier implementations. However, the methods are not extendible to other complex number operations, such as division and square root. These are not trivial operations, but rather become quite complicated to perform on machines with only real number operation capability. Also, each of the above designs are not well-suited for numeric-intensive computations containing division and/or square root operations, since division and square root require the presence of the most significant digit of the dividend (or radicand) before operation, but each of the above approaches

generates the most significant digit of the product last. A more suitable method for complex number operations is needed. We propose to use on-line arithmetic as the basis of our algorithm. The method we propose is suitable for mapping either to reconfigurable designs or application-specific integrated circuits (ASICs) However, the main target of the research is toward reconfigurable hardware, due to the flexibility of reconfigurable designs to support changes in specification.

1.2 On-line arithmetic

On-line arithmetic is a class of arithmetic operations in which all operations are performed digit serially, in a most significant digit first (MSDF) manner. The inherent digit-pipeline characteristic provides the means for efficient hardware implementations. Extensive research has been done in the area of real number on-line arithmetic over the past several decades. Algorithms have been developed for all basic operations [TE77] [Erc78] [Wat81] [Erc84b], arithmetic units have been designed [Irw77] [GE81] [Tu86] [GHM89], a simulator has been coded [RE81], and applications have been explored [Tu90] [Fer93]. The algorithms, implementations, and summariess of cost and critical delay for basic radix-2 floating-point on-line arithmetic operations are given in Appendix A.

The general assumption has been a real number system of the form 2^k. This leaves open the investigation into on-line arithmetic using a complex number system. This dissertation presents algorithms and efficient implementations for complex number on-line arithmetic with a complex radix $2j$.

One of the key parameters of on-line arithmetic is the *on-line delay*, defined as the number of digits of the operand(s) necessary in order to generate the first digit of the result. Each successive digit of the result is generated one per cycle. This is illustrated in Figure 1.1, with on-line delay $\delta = 4$.

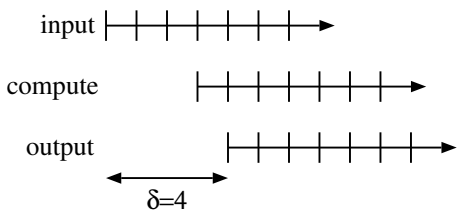


Figure 1.1: On-line delay of a function

Given a real integer radix r, the *on-line representation* of a fractional number x, up to the kth step, where δ is the on-line delay, is defined as:

$$X_k = X_{k-1} + x_{k+\delta-1} r^{-k-\delta+1}, \qquad X_0 = \sum_{i=1}^{\delta-1} x_i r^{-i} \tag{1.1}$$

Each digit x_i belongs to a symmetric redundant signed digit set consisting of digits in the range $\{-\rho, \ldots, -1, 0, 1, \ldots, \rho\}$, where ρ determines the amount of redundancy. Although such a digit set is redundant as long as $\rho \geq r/2$, it is

minimally-redundant if $\rho = r/2$

maximally-redundant if $\rho = r - 1$

over-redundant if $\rho > r - 1$

On-line arithmetic has several advantages including:

- *Ability to overlap dependent operations*, since on-line algorithms produce the output serially, most-significant digit first, enabling successive operations to begin before previous operations have completed. This allows pipelining at the digit level, and is well suited toward division and square root which by nature generate their respective outputs digit serially, most-significant digit first.

- *Low-bandwidth communication*, since intermediate results pass to and from modules digit-serially, so connections need only be one digit wide. As technology moves deeper into the submicron level, interconnection plays a more important role in determining chip area.

- *Support for variable precision*, since once a desired precision is obtained, successive outputs can be ignored. This is especially attractive for fixed point multiplication, where only the most significant half is retained. In on-line arithmetic, the least significant half does not need to be produced then discarded.

However, on-line arithmetic also has several drawbacks including:

- *Linear latency dependent on precision*, since one digit per cycle is generated, after the initial on-line delay δ. The total latency for m-digit precision is $m + \delta$ cycles. For single operations, conventional parallel implementations have typically much lower latency.

- *High cost to pipeline units*, which is true for serial arithmetic in general, since on-line units must be replicated in order to support pipelining at the operation level. A single on-line unit is occupied for a single operation and cannot begin another operation until the previous result is generated for a desired precision.

1.3 Reconfigurable hardware

Reconfigurable hardware modules containing a large number of switching-circuit components (logic and memory) have emerged as a viable technology for custom design. These modules, known as field programmable gate arrays (FPGAs), include an array of simple logic cells, some storage, and interconnection paths in each cell that are configured to perform specific functions by loading predetermined bit patterns. Once these configuration bit patterns are developed, the actual customization is very fast compared to the standard design process of producing custom designs. This introduces enormous flexibility and a variety of hardware structures available to the user. It also allows direct mapping of an application to hardware and allows optimization for maximum performance. Although the algorithms and corresponding implementations presented in this dissertation, and on-line arithmetic in general, are suited for ASIC technologies, FPGAs have recently gained notice as an attractive option for designs using on-line arithmetic. Girau and Tisserand [GT96] implemented multilayer perceptron back-propagation algorithms. Tisserand and Dimmler [TD97] designed real-time digital controllers.

Tenca and Ercegovac [TE98] designed a variable long-precision arithmetic unit. Tisserand, Marchal, and Piguet [TMP99] developed an on-line arithmetic based FPGA architecture for low power custom computing. Mosanya and Sanchez [MS99] implemented a generalized profile search. Lau, et. al. [LSE99] implemented signal processing algorithms for Fast Fourier Transform and Discrete Cosine Transform.

The key component of an FPGA is a configurable logic block (CLB). In the Xilinx Virtex family of FPGAs [Xil01], each CLB consists of two slices. Each CLB slice implements two 4-input Look-Up-Tables (LUTs) denoted F and G, two D-type flip-flops (DFFs) denoted X and Y, and some carry/control logic, as shown in Figure 1.2. The cost of a design mapped onto a Xilinx FPGA can be measured in terms of the number of CLB slices utilized. The timing characteristics of a Virtex CLB slice with speed grade -6 are summarized in Table 1.1.

All designs were realized by VHDL code corresponding to the functionality of the modules. When appropriate, the precision of a module was left as an unset generic variable, which could be set statically either by another higher-level module specifying the precision, or could be set individually for testing for a specific precison. The VHDL code was compiled, tested for behavioral correctness, and synthesized onto a Virtex FPGA to verify CLB slice cost and critical delay using the Xilinx Foundation 3.1i design tool [Xil01]. The VHDL source code for all modules is given in Appendix B.

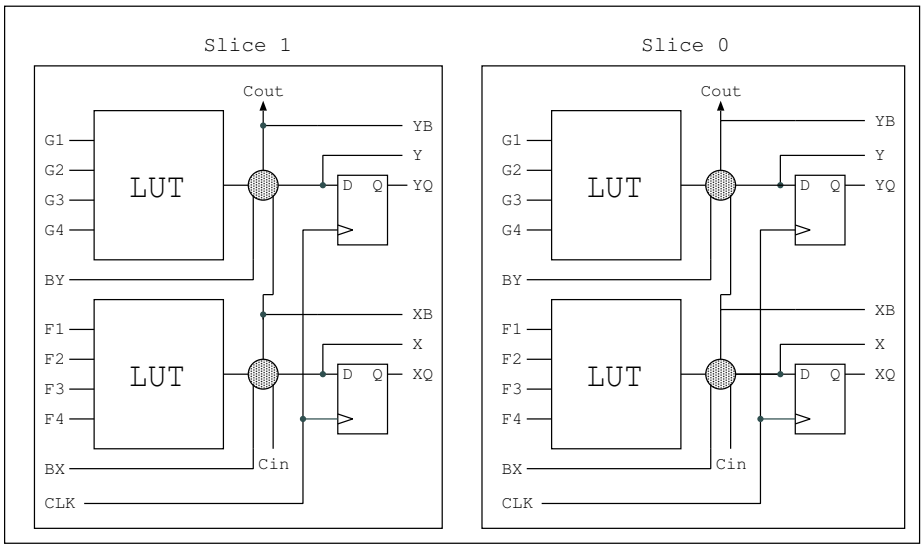


Figure 1.2: Virtex CLB

Description	Symbol	Max delay (ns)
Combinational Delays		
F/G inputs to X/Y outputs	T_{ILO}	0.6
F/G inputs to XB/YB outputs	T_{OPB}	1.3
BX/BY input to YX/YB outputs	T_{BB}	0.5
Cin input to X/Y outputs	T_{CIN}	0.5
Cin input to XB/YB outputs	T_{CINB}	0.1
Cin input to Cout output	T_{CC}	0.1
Sequential Delay		
Flip-flop clock to XQ/YQ outputs	T_{CKO}	1.1
Setup time before clock CLK		
F/G inputs	T_{ICK}	1.0
BX/BY inputs	T_{BICK}	1.6
Interconnect delay	T_{NET}	varies

Table 1.1: Virtex CLB slice timing characteristics (speed grade -6)

1.4 Related work

Related work in complex number on-line arithmetic and the derivation of algorithms for complex number arithemetic has been investigated by several authors. Frougny [Fro96] presented proofs on the computability of on-line algorithms for addition in complex number systems. Nielsen [Nie96] sketched the design of a complex number on-line multiplier using composite radix 2 digits consisting of a real and imaginary component. However, no actual implementations are produced.

1.5 Objectives

The goal of this research is to present the following:

- Presentation and utilization of a Redundant Complex Number System for the specification and design of primitive and basic on-line arithmetic modules.
- Derivation of algorithms for complex number on-line floating-point operations based on algorithms for known real number on-line floating-point operations.
- Implementation of the algorithms onto reconfigurable hardware, and comparisons with equivalent networks of real number on-line floating-point arithmetic operatorsin terms of cost and delay.
- Application of the proposed operators toward the design of an arithmetic unit to perform the complex singular value decomposition of a matrix.

CHAPTER 2

Redundant Complex Number System

In order to handle complex numbers efficiently, a suitable representation is essential. Various authors have proposed complex number systems based on residue arithmetic. These include Polynomial Residue Number System (PRNS) [SSS89], Quadratic Residue Number System (QRNS) [Leu81], Quadratic-Like Residue Number System (QLRNS) [SP84], and Modified Quadratic Residue Number System (MQRNS) [KJM85]. These are suitable for addition, subtraction and multiplication, but not for division and square root, since residue arithmetic is not closed under such operations. Other representations such as Complex Logarithmic Number System (CLNS) [Lew99] and Polygonal Number System (PNS) [DHK91] do not provide efficient methods for conversion to and from a conventional binary representation.

What is essential is a number system that is applicable to all basic arithmetic operations, namely addition, subtraction, multiplication, division, and square root. Also, there should be a simple method for converting to and from a conventional binary representation. For these reasons, Redundant Complex Number System (RCNS) [AOH95] is adopted for efficient complex number representation throughout this dissertation. Section 2.1 presents Redundant Complex Number System

(RCNS). Section 2.2 presents a suitable representation for complex floating-point numbers. Section 2.3 presents a binary-level borrow-save encoding representation. Section 2.4 presents recoding methods for RCNS. Section 2.5 presents conversion to and from radix-2 borrow-save encoding. Section 2.6 presents the on-line methodology for RCNS. Section 2.7 presents primitive on-line operations for RCNS. Section 2.8 summarizes the chapter.

2.1 Definition

Redundant Complex Number System (RCNS) is a radix rj system, in which digits are in the set $\{-\rho, \ldots, 0, \ldots, \rho\}$, where $r \geq 2$ and $\lceil r^2/2 \rceil \leq \rho \leq r^2 - 1$. Such a number system can be denoted $RCNS_{rj,\rho}$. The redundancy factor K for $RCNS_{rj,\rho}$ is

$$K = \frac{\rho}{r^2 - 1} \tag{2.1}$$

Redundant Complex Number System with $r = 2$, $\rho = 3$, and hence $K = 1$, denoted $RCNS_{2j,3}$, allows ease of the definition of primitive on-line arithmetic modules, as well as ease of conversion to and from other representations. Conversion to and from radix 2 borrow-save encoding is shown in Section 2.5. This number system was introduced as Quarter-imaginary Number System in [Knu73].

The inherent interleaving of real and imaginary digits allows the sharing of modules for implementation. Modules utilized for the generation of imaginary output digits in odd-numbered cycles are utilized for the generation of real output

digits in even-numbered cycles. This reduces the cost in comparison to approaches that utilize dedicated hardware for the generation of real and imaginary output digits, with the tradeoff of increased on-line delay due to alternating the arrival of real and imaginary digits.

2.2 Complex number floating-point arithmetic

A real floating-point number $x = X \cdot r^{e_x}$ is *normalized* if

$$r^{-1} \leq |X| < 1 \tag{2.2}$$

For a conventional non-redundant representation, this holds true if the most-significant digit of X is non-zero. For a redundant representation, however, this does not hold true. For example, the fraction $(0.1\overline{1111})_2 = (0.00001)_2$ is not normalized, even though the most-significant digit is non-zero. From the definitions in [Wat81], x is

normalized if $r^{-1} \leq |X| < 1$

quasi-normalized if $r^{-2} \leq |X| < 1$

pseudo-normalized if $r^{-k} \leq |X| < 1 \qquad (3 \leq k \leq m)$

Using $RCNS_{2j,3}$, a complex number $x = (X_R + jX_I) \cdot (2j)^{e_x}$ can be normalized with regard either to the real component X_R or the imaginary component X_I, depending on which has larger absolute value. The exponent e_x is shared between the real and imaginary component. Exponent overflow/underflow can be handled

by setting an exception flag, and allowing processing of results (although erroneous) to continue. However, this is not considered in the design of the arithmetic modules presented in this dissertation. A $RCNS_{2j,3}$ fraction x is considered

normalized if $2^{-1} \leq \max(|X_R|, |X_I|) < 1$

quasi-normalized if $2^{-2} \leq \max(|X_R|, |X_I|) < 1$

pseudo-normalized if $2^{-k} \leq \max(|X_R|, |X_I|) < 1 \qquad (3 \leq k \leq m)$

The output of a complex number operation can be unnormalized for several reasons:

1. The range of an output determined by the on-line algorithm allows it to be unnormalized.

2. Digit cancellation resulting from the addition/subtraction of numbers with the same exponent value.

In this dissertation, we assume operands of an $RCNS_{2j,3}$ on-line algorithm have non-zero most significant digits and are quasi-normalized. When the result Z exceeds the range of a quasi-normalized fraction (i.e. $\max(|Z_R, Z_I|) \geq 1$) then the exponent is incremented. When the result is below the range of a quasi-normalized fraction (i.e. $\max(|Z_R, Z_I| < \frac{1}{4})$, then the exponent is decremented and leading zeros are discarded. The normalization algorithm which takes as input the generated output digit z_k, the output exponent e_z and the on-line delay for

the arithmetic operation δ is shown below. This is similar to the normalization algorithm presented in [EL88b] for radix-2 on-line rotation.

NORM(z_k, e_z, δ)

/* Initialization */
  $k = 1$

/* Computation */
  if $k = \delta - 1$ and $z_k \neq 0$ then $\{Overnormalized\}$
    $e_z = e_z + 1$
  else if $k \geq \delta$ and $z_k = 0$ then $\{Undernormalized\}$
    $e_z = e_z - 1$
  end if
  $k = k + 1$

The cost for a normalization unit, assuming e-bit exponent precision is $2e$ CLB slices. The critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

2.3 Borrow-save encoding

For implementation, we need a bit-level representation for $RCNS_{2j,3}$ digits. Since the set of digit values in $RCNS_{2j,3}$ is $\{-3, \ldots, 3\}$, a minimum of 3 bits is required to represent each digit. However, a more efficient design due to increased redundancy in representation can be realized with 4 bits, by means of extending from the idea of radix 2 borrow-save encoding [GHM89]. Also, since Xilinx CLBs operate on the basis of 4-input LUTs, a 3-bit representation presents no advantage.

In radix-2 borrow-save encoding, digits belonging to the digit set $\{\overline{1}, 0, 1\}$ are coded with binary variables. Each digit x_k is represented as (x_k^+, x_k^-), such that $x_k = x_k^+ - x_k^-$ ($\overline{1}$ is represented as (0,1), 1 is represented as (1,0), and 0 is represented

either as (0,0) or (1,1)).

Using borrow-save encoding, for $RCNS_{2j,3}$, each digit x_k is represented as 4 bits $(x^+_{k,1}, x^-_{k,1}, x^+_{k,0}, x^-_{k,0})$, such that $x_k = 2x^+_{k,1} - 2x^-_{k,1} + x^+_{k,0} - x^-_{k,0}$. Valid representations of digits are shown in Table 2.1.

Digit	Valid representation(s)
$\overline{3}$	0101
$\overline{2}$	0100, 0111
$\overline{1}$	0001, 0110, 1101
0	0000, 0011, 1100, 1111
1	0010, 1001, 1110
2	1000, 1011
3	1010

Table 2.1: Valid representations for $RCNS_{2j,3}$ digits

2.4 Recoding algorithms

Although $RCNS_{2j,3}$ allows flexibility in representation, there are also several drawbacks:

- Handling digits 3 and -3 requires producing significand multiples $3X$ and $-3X$ which requires an extra addition step.
- A significand X with fractional real and imaginary components X_R and X_I can have integer digits, such as $(11.32\overline{12})_{2j} = \frac{3}{8} + \frac{3}{8}j$, which can complicate ensuring complex significands within the range $\max(|X_R|, |X_I|) < 1$.

To handle these cases, several recoding modules are presented: (i) digit-set recoding; and (ii) most-significant-digit recoding.

2.4.1 Digit-set recoding

In order to reduce the complexity introduced by handling digits -3 and 3, digit-set recoding initially recodes a $RCNS_{2j,3}$ digit $x_k \in \{-3, \ldots, 3\}$ into a pair of digits (t_{k-2}, w_k), in which $t_{k-2} \in \{-1, 0, 1\}$ and $w_k \in \{-2, \ldots, 2\}$ such that $x_k = -4t_{k-2} + w_k$. Then a $RCNS_{2j,2}$ digit χ_k is computed as $\chi_k = t_k + w_k$. In order to restrict $\chi_k \in \{-2, \ldots, 2\}$, two cases of pairs of values must be prevented: (i) $t_k = 1$, $w_k = 2$, (ii) $t_k = -1$, $w_k = -2$. To do so, x_{k+2} is examined. If $x_{k+2} \leq -2$ and $x_k = 2$, which could allow the first case, x_k is recoded as $(\overline{1}, \overline{2})$, otherwise as $(0, 2)$. In the same way, if $x_{k+2} \geq 2$ and $x_k = -2$, which could allow the second case, x_k is recoded as $(1, 2)$, otherwise as $(0, \overline{2})$ Then it is assured that $\chi_k \in \{-2, \ldots, 2\}$. The digit-set recoding algorithm DSREC is shown below.

DSREC(x_k, x_{k+2})

$$(t_{k-2}, w_k) = \begin{cases} (1,1) & \text{if } x_k = \overline{3} \\ (1,2) & \text{if } x_k = \overline{2} \text{ and } x_{k+2} \geq 2 \\ (0,\overline{2}) & \text{if } x_k = \overline{2} \text{ and } x_{k+2} < 2 \\ (0,\overline{1}) & \text{if } x_k = \overline{1} \\ (0,0) & \text{if } x_k = 0 \\ (0,1) & \text{if } x_k = 1 \\ (0,2) & \text{if } x_k = 2 \text{ and } x_{k+2} > -2 \\ (\overline{1},\overline{2}) & \text{if } x_k = 2 \text{ and } x_{k+2} \leq -2 \\ (\overline{1},\overline{1}) & \text{if } x_k = 3 \end{cases}$$

$\chi_k = t_k + w_k$

An example of recoding $(.013\overline{2}231\overline{1})_{2j}$ into $(.\overline{1}2\overline{1}12\overline{1}1\overline{1})_{2j}$ is shown in Table 2.2. The cost is 14 CLB slices and the critical delay is $T_{CKO} + T_{ICK} + T_{ILO} + T_{NET} = (2.7 + T_{NET})$ns.

k	x_k	t_k	w_k	χ_k
1	0	$\overline{1}$	0	$\overline{1}$
2	1	1	1	2
3	3	0	$\overline{1}$	$\overline{1}$
4	$\overline{2}$	$\overline{1}$	2	1
5	2	0	2	2
6	3	0	$\overline{1}$	$\overline{1}$
7	1	0	1	1
8	$\overline{1}$	0	$\overline{1}$	$\overline{1}$

Table 2.2: Digit-set recoding example

2.4.2 Most-significant-digit recoding

In order to handle carries produced when performing operations on significands consisting of $RCNS_{2j,3}$ digits, most-significant-digit recoding recodes most-significant residual digits $w_{-1}, w_0 \in \{-1, 0, 1\}$ of respective weights $(2j)^1 = 2j$ and $(2j)^0 = 1$, and digits $w_1, w_2 \in \{-3, \ldots, 3\}$, of respective weights $(2j)^{-1}$ and $(2j)^{-2}$, into digits $\omega_1, \omega_2 \in \{-3, \ldots, 3\}$ of respective weights $(2j)^{-1}$ and $(2j)^{-2}$. The algorithm MSREC for recoding general digits w_{k-2} and w_k into digit ω_k is shown below. The cost is 13 CLB slices and the critical delay is $4T_{ILO}+5T_{NET} = (2.4+5T_{NET})$ns.

MSREC(w_{k-2}, w_k)

$$\omega_k = \begin{cases} \overline{3} & \text{if } (w_{k-2} = 0 \text{ and } w_k = \overline{3}) \text{ or } (w_{k-2} = 1 \text{ and } w_k = 1) \\ \overline{2} & \text{if } (w_{k-2} = 0 \text{ and } w_k = \overline{2}) \text{ or } (w_{k-2} = 1 \text{ and } w_k = 2) \\ \overline{1} & \text{if } (w_{k-2} = 0 \text{ and } w_k = \overline{1}) \text{ or } (w_{k-2} = 1 \text{ and } w_k = 3) \\ 0 & \text{if } w_{k-2} = 0 \text{ and } w_k = 0 \\ 1 & \text{if } (w_{k-2} = 0 \text{ and } w_k = 1) \text{ or } (w_{k-2} = \overline{1} \text{ and } w_k = \overline{3}) \\ 2 & \text{if } (w_{k-2} = 0 \text{ and } w_k = 2) \text{ or } (w_{k-2} = \overline{1} \text{ and } w_k = \overline{2}) \\ 3 & \text{if } (w_{k-2} = 0 \text{ and } w_k = 3) \text{ or } (w_{k-2} = \overline{1} \text{ and } w_k = \overline{1}) \end{cases}$$

2.5 Comparison with radix-2 borrow-save representation

In order to compare approaches with similar throughput, we compare implementations of complex number operators using $RCNS_{2j,3}$, which interleaves the real and imaginary components, with equivalent networks of radix-2 on-line arithmetic units, denoted as *real networks*. A $RCNS_{2j,3}$ on-line arithmetic unit outputs after the initial on-line delay, one $RCNS_{2j,3}$ digit $z_k = (z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0})$ per cycle, equivalent to two bits of either the real or imaginary component of the result in a conventional binary representation. Alternatively, a complex number on-line arithmetic unit implemented as a network of radix-2 arithmetic units, outputs after the initial on-line delay, one real radix-2 borrow-save digit $z_{R,k} = (z^+_{R,k}, z^-_{R,k})$ and one imaginary radix-2 borrow-save digit $z_{I,k} = (z^+_{I,k}, z^-_{I,k})$ per cycle, equivalent to one bit of the real and one bit of the imaginary component in a conventional binary representation. In both approaches, an equivalent m-bit real and m-bit imaginary result is produced after $m + \delta$ cycles. Therefore, ignoring potentially different on-line delays, the throughput of the two approaches is equivalent, as shown in Figure 2.1. The main difference is that the $RCNS_{2j,3}$ approach shares modules to produce real and imaginary digits in alternate cycles, whereas the radix-2 approach uses dedicated hardware to produce real and imaginary digits within the same cycle.

The effect sharing has on cost and peformance is that the $RCNS_{2j,3}$ approach generally achieves lower cost, as shown in the later chapters, than the equivalent

radix-2 network appoach in the implementation of a complex number arithmetic operator. The digit set recoding step presented in Section 2.4.1. eliminates the need to produce multiples $-3X$ and $3X$ when performing vector-digit multiplication, which is one of the common operations of an on-line arithmetic recurrence. Producing multiples $-3X$ and $3X$ is handled by adding $\pm 2X$ and $\pm X$, either at the time of performing vector-digit multiplication, or is postponed as a later operation. Eliminating this step results in a significant reduction in cost, a reduction which cannot be accomplished in the equivalent radix 2 network approach. However, alternating cycles results in increasing the on-line delay, which can result in overall higher latency in comparison to the equivalent radix-2 network approach.

Figure 2.1: Throughput of radix-2 and $RCNS_{2j,3}$ on-line approaches

Since the throughput of the two approaches is the same, one format can be easily converted to the other by means of rearranging and negating bits based on the index of the digit to convert between $(2)^{e_x}$ and $(2j)^{e_x}$. Section 2.3.1 describes radix-2 to $RCNS_{2j,3}$ borrow-save encoding conversion, and Section 2.3.2. describes $RCNS_{2j,3}$ to radix-2 borrow-save encoding conversion.

2.5.1 Radix-2 to $RCNS_{2j,3}$ borrow-save conversion

Conversion from individual radix-2 real and imaginary borrow-save digits $z_{R,k} = (z^+_{R,k}, z^-_{R,k})$ and $z_{I,k} = (z^+_{I,k}, z^-_{I,k})$, respectively, to the equivalent $RCNS_{2j,3}$ digit $z_k = (z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0})$ is given below. An example, assuming real and imaginary inputs: $z_R = (0.\overline{1}00100)_2$ and $z_I = (0.\overline{11}0110)_2$ and output $z = (0.12\overline{2}1\overline{3}0)_{2j}$, is shown in Table 2.3, with a constant cost of 3 CLBs and a critical delay of $T_{CKO} + T_{ILO} + 2T_{NET} = (2.1 + 2_{NET})$ns.

Radix-2 to $RCNS_{2j,3}$ Borrow-save Conversion

if k mod 4=0 then
$(z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0}) = (z^+_{R,k-2}, z^-_{R,k-2}, z^+_{R,k}, z^-_{R,k})$
else if k mod 4=1 then
$(z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0}) = (\overline{z^+_{I,k-2}}, \overline{z^-_{I,k-2}}, \overline{z^+_{I,k}}, \overline{z^-_{I,k}})$
else if k mod 4=2 then
$(z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0}) = (\overline{z^+_{R,k-2}}, \overline{z^-_{R,k-2}}, \overline{z^+_{R,k}}, \overline{z^-_{R,k}})$
else if k mod 4=3 then
$(z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0}) = (z^+_{I,k-2}, z^-_{I,k-2}, z^+_{I,k}, z^-_{I,k})$

k	$z_{R,k}$	$z_{I,k}$	z_k
1	$\overline{1}$	$\overline{1}$	1
2	0	$\overline{1}$	2
3	0	0	$\overline{2}$
4	1	1	1
5	0	1	$\overline{3}$
6	0	0	0

Table 2.3: Radix-2 to $RCNS_{2j,3}$ borrow-save conversion example

2.5.2 $RCNS_{2j,3}$ to radix-2 borrow-save conversion

Conversion from $RCNS_{2j,3}$ borrow-save digit $z_k = (z^+_{k,1}, z^-_{k,1}, z^+_{k,0}, z^-_{k,0})$ to equivalent individual real and imaginary radix-2 borrow-save digits $z_{R,k} = (z^+_{R,k}, z^-_{R,k})$ and $z_{I,k} = (z^+_{I,k}, z^-_{I,k})$, respectively, is shown below. An example, assuming input $z = (0.12\overline{2}1\overline{3}0)_{2j}$ and real and imaginary outputs $z_R = (0.\overline{1}00100)_2$ and $z_I = (0.\overline{11}0110)_2$ is shown in Table 2.4, with a cost of 3 CLBs and a critical delay of $T_{CKO} + T_{ILO} + 2T_{NET} = (2.1 + 2_{NET})$ns.

$RCNS_{2j,3}$ to Radix-2 Borrow-save Conversion

if k mod 4=0 then
$(z^+_{R,k}, z^-_{R,k}) = (z^+_{k,1}, z^-_{k,1})$
$(z^+_{I,k}, z^-_{I,k}) = (z^+_{k-2,0}, z^-_{k-2,0})$
else if k mod 4=1 then
$(z^+_{R,k}, z^-_{R,k}) = (z^+_{k-2,0}, z^-_{k-2,0})$
$(z^+_{I,k}, z^-_{I,k}) = (\overline{z^+_{k,1}}, \overline{z^-_{k,1}})$
else if k mod 4=2 then
$(z^+_{R,k}, z^-_{R,k}) = (\overline{z^+_{k,1}}, \overline{z^-_{k,1}})$
$(z^+_{I,k}, z^-_{I,k}) = (\overline{z^+_{k-2,0}}, \overline{z^-_{k-2,0}})$
else if k mod 4=3 then
$(z^+_{R,k}, z^-_{R,k}) = (\overline{z^+_{k-2,0}}, \overline{z^-_{k-2,0}})$
$(z^+_{I,k}, z^-_{I,k}) = (z^+_{k,1}, z^-_{k,1})$

k	z_k	$z_{R,k}$	$z_{I,k}$
1	1	$\overline{1}$	$\overline{1}$
2	2	0	$\overline{1}$
3	$\overline{2}$	0	0
4	1	1	1
5	$\overline{3}$	0	1
6	0	0	0

Table 2.4: $RCNS_{2j,3}$ to radix-2 borrow-save conversion example

2.6 Design methodology for complex number on-line arithmetic

The systematic method for developing on-line algorithms for real number arithmetic was presented in [EL88a]. For complex number arithmetic operations, we initially treat the real and imaginary results separately using a real radix $-r^2$, then interleave the two recurrences by scaling the imaginary recurrence by rj, producing a unified expression with a complex radix rj. The complex number on-line methodology consists of the following nine steps:

Step 1–Description

Describe the complex number on-line operation by the bound on the error after k digits have been computed. For a complex number binary operation f, with operands $X = X_R + jX_I$ and $Y = Y_R + jY_I$ and result $Z = Z_R + jZ_I$, f can be separated into two functions, f_R which computes the real component, and f_I which computes the imaginary component. Assuming a real radix $-r^2$, the bounds have the form

$$\begin{aligned} |f_R(X_R[k], X_I[k], Y_R[k], Y_I[k]) - Z_R[k]| &< |(-r^2)^{-k}| \\ |f_I(X_R[k], X_I[k], Y_R[k], Y_I[k]) - Z_I[k]| &< |(-r^2)^{-k}| \end{aligned} \tag{2.3}$$

Step 2–Transform

Define a transform G, such that $G(f_R)$ and $G(f_I)$ are recurrences using only vector append, radix multiplication, vector-digit multiplication, and vector addition

operations. This results in expressions of the form

$$\begin{array}{l} G(-|(-r^2)^{-k}|) < G(f_R(X_R[k], X_I[k], Y_R[k], Y_I[k]) - Z_R[k]) < G(|(-r^2)^{-k}|) \\ G(-|(-r^2)^{-k}|) < G(f_I(X_R[k], X_I[k], Y_R[k], Y_I[k]) - Z_I[k]) < G(|(-r^2)^{-k}|) \end{array} \tag{2.4}$$

Step 3–Residual

Define scaled residuals as

$$\begin{array}{l} W_R[k] = (-r^2)^k (G(f_R(X_R[k], X_I[k], Y_R[k], Y_I[k]) - Z_R[k])) \\ W_R[0] = G(f_R(X_R[0], X_I[0], Y_R[0], Y_I[0]) - Z_R[0]) \\ W_I[k] = (-r^2)^k (G(f_I(X_R[k], X_I[k], Y_R[k], Y_I[k]) - Z_I[k])) \\ W_I[0] = G(f_I(X_R[0], X_I[0], Y_R[0], Y_I[0]) - Z_I[0]) \end{array} \tag{2.5}$$

with bounds

$$\begin{array}{l} |(-r^2)^k| G(-|(-r^2)^{-k}|) \leq -\omega_R < W_R[k] < \omega_R \leq |(-r^2)^k| G(|(-r^2)^{-k}|) \\ |(-r^2)^k| G(-|(-r^2)^{-k}|) \leq -\omega_I < W_I[k] < \omega_I \leq |(-r^2)^k| G(|(-r^2)^{-k}|) \end{array} \tag{2.6}$$

Step 4–Recurrence

Since $\sqrt{-r^2} = rj$, interleaving the two residuals such that the imaginary residual is scaled by rj, determine a recurrence of the form

$$\begin{array}{rl} W[k] = & (rj)W[k-1] + (rj)(G(f(X[k], Y[k]) - Z[k]) \\ & -G(f(X[k-1], Y[k-1]) - Z[k-1])) \end{array} \tag{2.7}$$

where

$$\begin{array}{l} W[k] = W_R[k] + rjW_I[k] \\ X[k] = X_R[k] + rjX_I[k] \\ Y[k] = Y_R[k] + rjY_I[k] \\ Z[k] = Z_R[k] + rjZ_I[k] \end{array} \tag{2.8}$$

Step 5–Decompose

Decompose the recurrence to make $H_1[k]$ independent of the output digit at step $k-1$, namely z_{k-1}

$$W[k] = (rj)(W[k-1] + H_2[k]z_{k-1}) + H_1[k] \tag{2.9}$$

Step 6–Limits

Determine the limits L_d and U_d of the selection intervals of the even-indexed component of $W[k-1]$, denoted $W_E[k-1]$, such that if $L_d \leq W_E[k] < U_d$, then selecting output digit $z_k = d$ is valid. This alternates the selection of real and imaginary digits. The limits are defined based on the error ω as

$$\begin{aligned} L_d &= -dH_2[k] - \omega \\ U_d &= -dH_2[k] + \omega \end{aligned} \tag{2.10}$$

Step 7–Containment

Determine the error ω using the containment condition

$$\begin{aligned} U_\rho &\geq \max(r(W_E[k-1] + z_{k-1}H_2[k]) + H_{1,E}) = r\omega + \max(H_{1,E}) \\ L_\rho &\leq \min(r(W_E[k-1] + z_{k-1}H_2[k]) + H_{1,E}) = -r\omega + \min(H_{1,E}) \end{aligned} \tag{2.11}$$

where $H_{1,E}[k]$ is the even-indexed component of $H_1[k]$.

Step 8–Value of on-line delay δ

Determine the minimum overlap between adjacent intervals to use an estimate of $W_E[k]$. Compute the bound on the on-line delay δ, with error

$$|W_E[k] - \overline{W_E[k]}| < \omega \tag{2.12}$$

After assimilating t bits of $W_E[k]$, then for the selection interval $[m_d, M_d]$ for d,

$$\begin{array}{c} m_d = M_{d-1} + 2^{-t} \\ M_{d-1} + \omega \leq U_{d-1} \\ m_d - \omega \geq L_d \end{array} \tag{2.13}$$

This results in a minimum overlap of

$$\Delta(d-1, d) = U_{d-1} - L_d \geq 2\omega - 2^{-t} \tag{2.14}$$

The worst case value of the expression results in a relation between δ and t.

Step 9–Selection

Define the selection function as

$$z_k = \begin{cases} -\rho & \widehat{W}_E[k] < m_{-\rho} \\ d & m_d \leq \widehat{W}_E[k] < m_{d+1} \text{ where } (-\rho < d < \rho) \\ \rho & \widehat{W}_E[k] \geq m_\rho \end{cases} \tag{2.15}$$

2.7 Modular design of complex number on-line arithmetic operators

On-line arithmetic units are implemented as a series of digit slice modules, each performing operations on a pair of digits. For a $RCNS_{2j,2}$ on-line arithmetic unit, connections are made between "nearest two modules". The general case is shown in Figure 2.2. Each module consists of a fixed set of primitive modules. These include (i) borrow-save digit multipliers, (ii) borrow-save digit adders, and (iii) various latches and flip-flops (both bit-wide and digit-wide) for "appending" digits to vectors and performing radix multiplication (left shift). Each operation will be discussed at both a high-level description and at the binary level. Parameters including CLB cost and critical delay are also given.

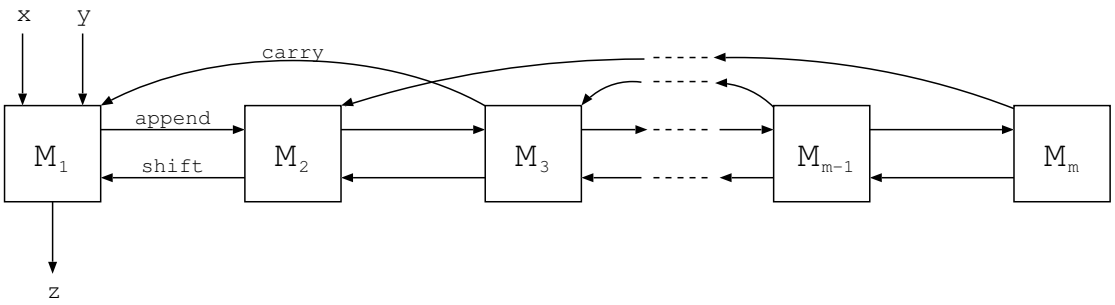


Figure 2.2: Modular structure of a complex number on-line arithmetic unit

2.7.1 Vector append

Appending a digit x_k to a vector $X[k-1]$ consists of replacing the default value of 0 at index k with x_k. An append register of length m consists of m digit-wide latches (label L_1 to L_m). When START is set to 1, L_1 is enabled and loaded with x_1. On the next clock, START is set to 0, L_1 is disabled, and L_2 is enabled and loaded with x_2. The process continues until all m input digits are loaded. This is illustrated in Figure 2.3. Each digit slice consists of a bit-wide register of D flip-flops and a digit-wide latch is mapped to 5 DFFs. Assuming m-digit precision, the cost is $\lceil 5m/2 \rceil$ CLB slices. The critical delay $T_{CKO}+T_{ICK}+T_{NET} = (2.1+T_{NET})$ns.

2.7.2 Radix multiplication

Performing the operation $Z = (2j)X[k]$, where $X[k] = (x_1 x_2, \ldots, x_k)$, consists of left shifting the digits, such that the bits of each digit $z_i = (z_{i,1}^+, z_{i,1}^-, z_{i,0}^+, z_{i,0}^-)$ are

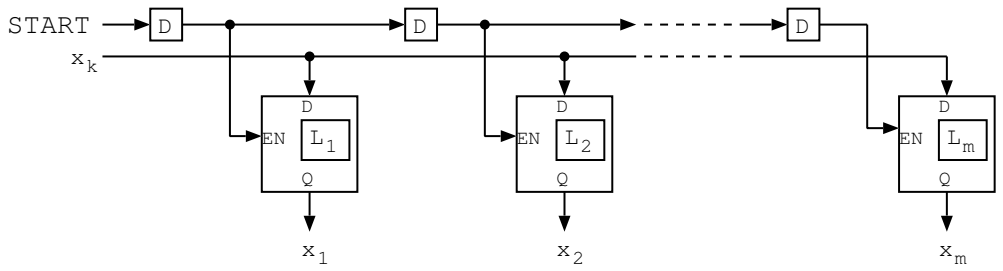


Figure 2.3: Append register

set as

$$\begin{array}{ll} z^+_{i,1} = x^+_{i+1,1} & z^-_{i,1} = x^-_{i+1,1} \\ z^+_{i,0} = x^+_{i+1,0} & z^-_{i,0} = x^-_{i+1,0} \end{array} \tag{2.16}$$

It can be implemented using a digit-wide register of D flip-flops, with a cost, assuming m-digit precision, of $2m$ CLB slices and a critical delay of $T_{CKO} + T_{ICK}$=2.1ns.

2.7.3 Vector-digit multiplication

Vector-digit multiplication for $RCNS_{2j,2}$ is defined such that given a digit vector $X[k] = (x_1, \ldots, x_k)$, with recoded digits $x_i \in \{-2, \ldots, 2\}$ and recoded digit $y_k \in \{-2, \ldots, 2\}$, we want to compute $Z[k] = X[k]y_k$, where $Z[k] = (z_1, \ldots, z_k)$ and each $z_i \in \{-3, \ldots, 3\}$.

For each output digit z_i ($i = 1$ to k), we compute an intermediate carry digit c_{i-2} and product digit p_i, where $x_i y_k = -4c_{i-2} + p_i$ such that

$$(c_{i-2}, p_i) = \begin{cases} (\overline{1}, x_i y_k - 4) & \text{if } x_i y_k > 2 \\ (1, x_i y_k + 4) & \text{if } x_i y_k < -2 \\ (0, x_i y_k) & \text{otherwise} \end{cases} \tag{2.17}$$

$x_i y_k$	$\overline{4}$	$\overline{2}$	$\overline{1}$	0	1	2	4
c_{i-2}	1	0	0	0	0	0	$\overline{1}$
p_i	0	$\overline{2}$	$\overline{1}$	0	1	2	0

Table 2.5: Range of $RCNS_{2j,3}$ vector-digit multiplication values

This is illustrated in Table 2.5. Then $z_i = c_i + p_i$. Since $p_i \in \{-2, \ldots, 2\}$ and $c_i \in \{-1, 0, 1\}$, then $z_i \in \{-3, \ldots, 3\}$, and each digit z_i can be computed in parallel. An example of $RCNS_{2j,2}$ vector-digit multiplication is shown below.

i	-1	0	1	2	3	4	5	6
x_i			.1	$\overline{1}$	2	1	3	$\overline{2}$
y_k		$\times$						$\overline{2}$
p_i			$\overline{2}$	2	0	$\overline{2}$	$\overline{2}$	0
c_i	0	0	1	0	1	$\overline{1}$		
z_i			$.\overline{1}$	2	1	$\overline{3}$	$\overline{2}$	0

A $RCNS_{2j,3}$ vector-digit multiplier can be implemented as a series of $RCNS_{2j,3}$ borrow-save digit multipliers. Each $RCNS_{2j,3}$ borrow-save digit multiplier takes as input two recoded $RCNS_{2j,3}$ digits, namely $x_i = (x^+_{i,1}, x^-_{i,1}, x^+_{i,0}, x^-_{i,0})$ and $y_k = (y^+_{k,1}, y^-_{k,1}, y^+_{k,0}, y^-_{k,0})$, and the two most significant bits of digit x_{i+2}, namely $x_{i+2,1} = (x^+_{i+2,1}, x^-_{i+2,1})$, and produces digit $z_i = (z^+_{i,1}, z^-_{i,1}, z^+_{i,0}, z^-_{i,0})$, as shown in Figure 2.4. The cost for a m-digit borrow-save vector-digit multiplier is $4m$ CLB slices and the critical delay is $2T_{ILO} + 2T_{NET} = (1.2 + 2T_{NET})$ns.

2.7.4 Vector addition

Addition of vectors is used to update the residual based on new input digits. For the general addition of $RCNS_{2j,3}$ vectors: $Z[k] = X[k] + Y[k]$, this can be

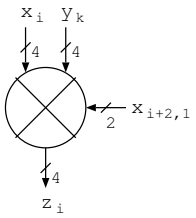


Figure 2.4: $RCNS_{2j,3}$ borrow-save digit multiplier

realized by computing intermediate carry digits c_{i-2} and sum digits s_i of individual elements x_i and y_i, where $x_i + y_i = -4c_{i-2} + s_i$, such that

$$(c_{i-2}, s_i) = \begin{cases} (\overline{1}, x_i + y_i - 4) & \text{if } x_i + y_i > 2 \\ (1, x_i + y_i + 4) & \text{if } x_i + y_i < -2 \\ (0, x_i + y_i) & \text{otherwise} \end{cases} \tag{2.18}$$

This is illustrated in Table 2.6.

$x_i \pm y_i$	$\overline{6}$	$\overline{5}$	$\overline{4}$	$\overline{3}$	$\overline{2}$	$\overline{1}$	0	1	2	3	4	5	6
c_{i-2}	1	1	1	1	0	0	0	0	0	$\overline{1}$	$\overline{1}$	$\overline{1}$	$\overline{1}$
s_i	$\overline{2}$	$\overline{1}$	0	1	$\overline{2}$	$\overline{1}$	0	1	2	$\overline{1}$	0	1	2

Table 2.6: Range of $RCNS_{2j,3}$ vector addition values

Then $z_i = c_i + s_i$. Since $s_i \in \{-2, \ldots, 2\}$ and $c_i \in \{-1, 0, 1\}$, then each output digit $z_i \in \{-3, \ldots, 3\}$ can be computed in parallel. An example of $RCNS_{2j,3}$ addition is shown below.

i	-1	0	1	2	3	4	5	6
x_i			.1	$\overline{1}$	2	3	1	$\overline{2}$
y_i		+	.1	$\overline{1}$	2	$\overline{1}$	2	$\overline{3}$
s_i			2	$\overline{2}$	0	2	$\overline{1}$	$\overline{1}$
c_i	0	0	$\overline{1}$	0	$\overline{1}$	1		
z_i			.1	$\overline{2}$	$\overline{1}$	3	$\overline{1}$	$\overline{1}$

For implementation, borrow-save addition consists of a network of PPM (plus-plus-minus) and MMP (minus-minus-plus) adder cells [GHM89]. These are basic full-adders in which certain inputs/outputs are negated (labeled $-$), or not negated (labeled $+$). The two cells are equivalent in functionality, differing only in the inversion of various inputs and outputs. The cells are illustrated in Figure 2.5. Each PPM/MMP adder cell has a cost of 1 CLB slice and delay $T_{ILO} + T_{NET} = (0.6 + T_{NET})$ns. A $RCNS_{2j,3}$ vector adder can be implemented as a series of $RCNS_{2j,3}$ borrow-save digit adders. The two commonly used types of digit adders, two-input (denoted 2:1) and three-input (denoted 3:1) are described below.

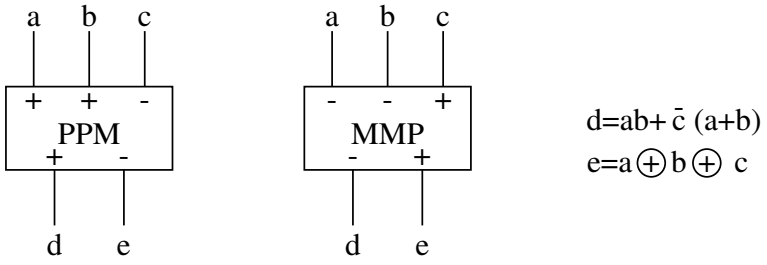
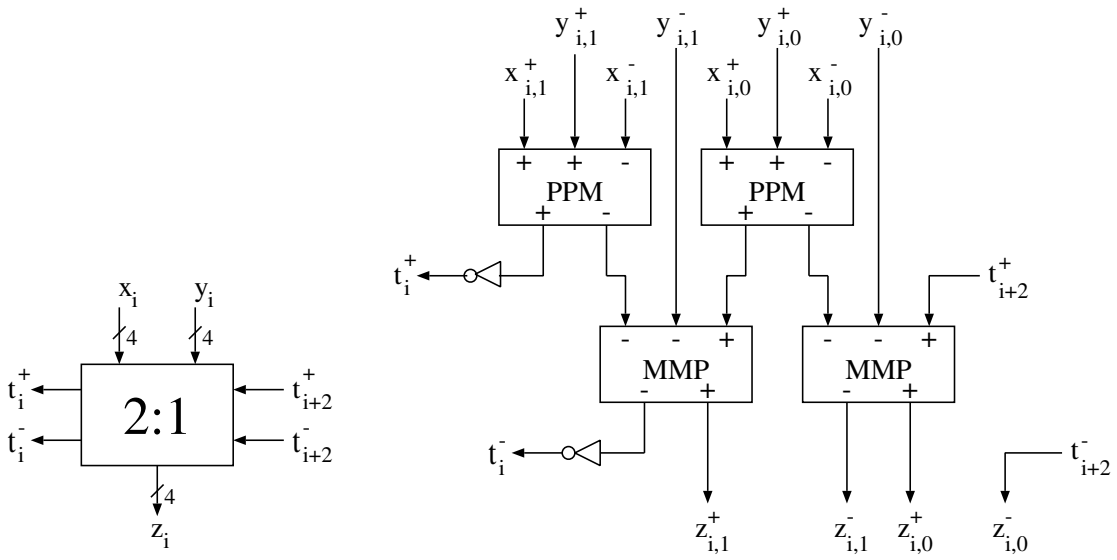


Figure 2.5: PPM and MMP adder cells

2.7.4.1 2:1 borrow-save digit adder

A $RCNS_{2j,3}$ 2:1 borrow-save digit adder, as shown in Figure 2.6, takes two $RCNS_{2j,3}$ borrow-save digits x_i and y_i, carry-in bits t_{i+2}^{+} and t_{i+2}^{-}, and produces sum digit z_i and carry-out bits t_i^{+} and t_i^{-}. The cost is 4 CLB slices and the delay is $2T_{ILO} + 2T_{NET} = (1.2 + 2T_{NET})$ns.



Symbol Schematic

Figure 2.6: $RCNS_{2j,3}$ 2:1 borrow-save digit adder

2.7.4.2 3:1 borrow-save digit adder

A $RCNS_{2j,3}$ 3:1 borrow-save digit adder, as shown in Figure 2.7 takes three $RCNS_{2j,3}$ borrow-save digits w_i, x_i and y_i, carry-in bits $t^+_{i+2,1}$, $t^-_{i+2,1}$, $t^+_{i+2,0}$, and $t^-_{i+2,0}$, and produces sum digit z_i and carry-out bits $t^+_{i,1}$, $t^-_{i,0}$, $t^+_{i,0}$, and $t^-_{i,0}$. The cost is 8 CLB slices and the delay is $3T_{ILO} + 4T_{NET} = (1.8 + 4T_{NET})$ns.

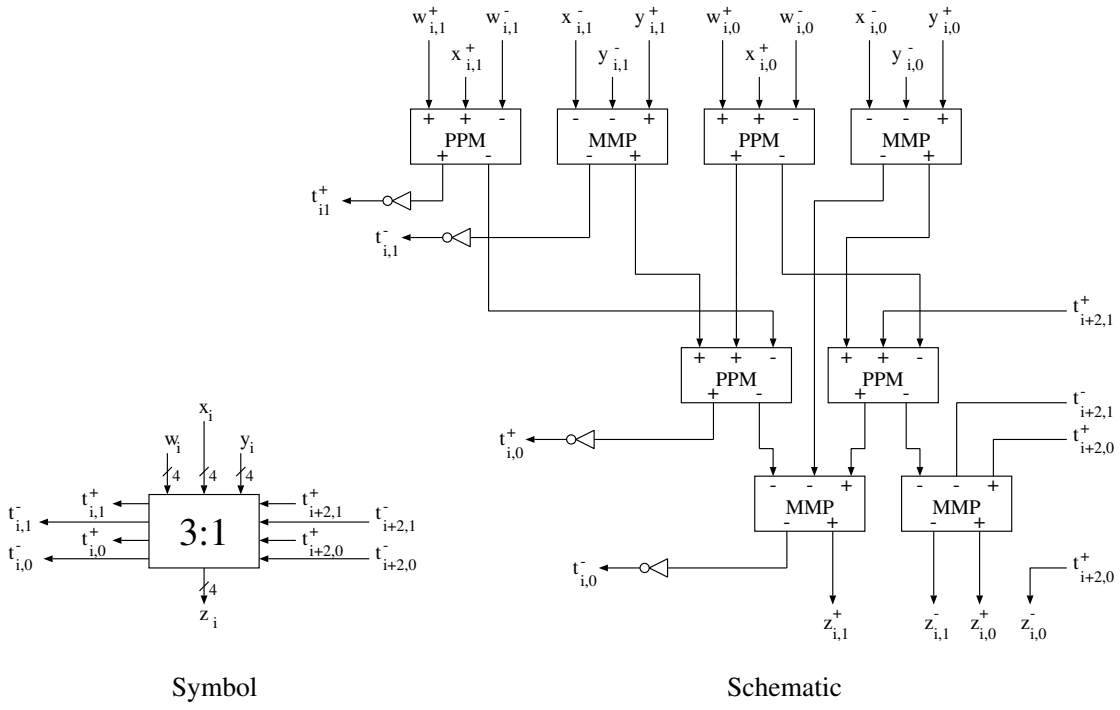


Figure 2.7: $RCNS_{2j,3}$ 3:1 borrow-save digit adder

2.8 Concluding Remarks

From the presentation of redundant complex number system (RCNS), the algorithms for conversion and recoding, and the design methodology for complex number on-line arithmetic, we make the following general observations.

1. The $RCNS_{2j,3}$ representation simplifies the derivation of complex number on-line arithmetic algorithms by unifying the real and imaginary component expressions.

2. Conversion between radix-2 and $RCNS_{2j,3}$ borrow-save encoding, which have the same throughput, can be performed at a cost independent of precision and at the digit level.

3. The $RCNS_{2j,3}$ representation reuses modules by alternating the generation of real and imaginary output digts, which reduces cost due to the digit-set recoding step, with the tradeoff of an increased latency.

4. Sharing the exponent between the real and imaginary components eliminates the need for normalization of both components. However, this results in either the real or imaginary component being left undernormalized, leading to potential loss of significance.

CHAPTER 3

Complex Number On-line Floating-point Addition

Complex number floating-point addition ($z = x + y$) is defined such that given inputs $x = (X_R + jX_I) \cdot (2j)^{e_x}$ and $y = (Y_R + jY_I) \cdot (2j)^{e_y}$, the output $z = (Z_R + jZ_I) \cdot (2j)^{e_z}$ is produced such that

$$\begin{aligned} Z_R &= X_R + Y_R \\ Z_I &= X_I + Y_I \\ e_z &= \max(e_x, e_y) \end{aligned} \tag{3.1}$$

For example, if $x = (-\frac{21}{32} + j\frac{3}{32}) \cdot (2j)^2 = \frac{21}{32} - j\frac{3}{8}$ and $y = (\frac{9}{32} - j\frac{3}{16}) \cdot (2j)^{-1} = -\frac{3}{32} - j\frac{9}{64}$ then $z = (-\frac{81}{128} + j\frac{33}{256}) \cdot (2j)^2 = \frac{81}{32} - j\frac{33}{64}$.

Section 3.1 presents the derivation of the algorithm for complex number on-line floating-point addition, including key characteristics such as on-line delay and the output digit selection function. Section 3.2 presents the modular design of a complex number on-line floating-point adder. The design is mapped to a Virtex FPGA, and design parameters including cost and delay are determined. Section 3.3 compares the design with a conventional network of real number on-line arithmetic operators in terms of cost and delay. Section 3.4 summarizes the chapter and reviews the main contributions.

3.1 Derivation of the algorithm

Assuming a real number negative radix $-r^2$, complex number on-line addition can be derived from the bound

$$\begin{array}{l} |(X_R[k] + Y_R[k]) - Z_R[k]| < |(-r^2)^{-k}| \\ |(X_I[k] + Y_I[k]) - Z_I[k]| < |(-r^2)^{-k}| \end{array} \tag{3.2}$$

The residuals are defined as

$$\begin{array}{l} W_R[k] = (-r^2)^k((X_R[k] + Y_R[k]) - Z_R[k]) \\ W_I[k] = (-r^2)^k((X_I[k] + Y_I[k]) - Z_I[k]) \end{array} \tag{3.3}$$

The recurrences are

$$\begin{array}{l} W_R[k] = -r^2(W_R[k-1] - z_{R,k-1}) + (-r^2)^{-\delta+1}(x_{R,k+\delta-1} + y_{R,k+\delta-1}) \\ W_I[k] = -r^2(W_I[k-1] - z_{I,k-1}) + (-r^2)^{-\delta+1}(x_{I,k+\delta-1} + y_{I,k+\delta-1}) \end{array} \tag{3.4}$$

with initial conditions

$$\begin{array}{l} W_R[0] = X_R[0] + Y_R[0] \\ W_I[0] = X_I[0] + Y_I[0] \end{array} \tag{3.5}$$

Defining

$$W[k] = W_R[k] + rjW_I[k] \tag{3.6}$$

the two recurrences can be combined as

$$\begin{array}{rl} W[k] = & -r^2((W_R[k-1] + rjW_I[k-1]) - (z_{R,k-1} + rjz_{I,k-1})) \\ & +(-r^2)^{-\delta+1}((x_{R,k+\delta-1} + rjx_{I,k+\delta-1}) + (y_{R,k+\delta-1} + rjy_{I,k+\delta-1})) \end{array} \tag{3.7}$$

Since $\sqrt{-r^2} = rj$, alternating the real and imaginary digits yields recurrence equation

$$W[k] = (rj)(W[k-1] - z_{k-1}) + (rj)^{-\delta+1}(x_{k+\delta-1} + y_{k+\delta-1}) \tag{3.8}$$

where

$$\begin{aligned} x_{k+\delta-1} &= \begin{cases} x_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjx_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ y_{k+\delta-1} &= \begin{cases} y_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjy_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ z_{k-1} &= \begin{cases} z_{R,k-1} & \text{if } k-1 \text{ is even} \\ rjz_{I,k-1} & \text{if } k-1 \text{ is odd} \end{cases} \end{aligned} \tag{3.9}$$

This can be decomposed into

$$W[k] = (rj)(W[k-1] + H_2[k]z_{k-1}) + H_1[k] \tag{3.10}$$

where

$$\begin{aligned} H_1[k] &= (rj)^{-\delta+1}(x_{k+\delta-1} + y_{k+\delta-1}) \\ H_2[k] &= -1 \end{aligned} \tag{3.11}$$

The limits can be determined as

$$\begin{aligned} U_d &\leq \omega - H_2[k]d = d + \omega \\ L_d &\geq -\omega - H_2[k]d = d - \omega \end{aligned} \tag{3.12}$$

Under the containment condition

$$\omega = K(1 - 2r^{-\delta+1}) \tag{3.13}$$

The minimum limit on the on-line delay is

$$\delta = 2\left(-\log_{r^2}\left(\frac{1}{2} - \frac{1}{4K}\right)\right) + 1 \tag{3.14}$$

Using $RCNS_{2j,3}$ borrow-save representation, $r = 2$ and K=1. Then the on-line delay δ, the error ω, and the limits U_d, L_d are

$$\begin{aligned} &\delta = 3 \\ &\omega = \tfrac{1}{2} \\ &U_d = \tfrac{1}{2} + d \\ &L_d = -\tfrac{1}{2} + d \end{aligned} \tag{3.15}$$

The $RCNS_{2j,3}$ on-line floating-point addition algorithm is shown below. The selection points m_d ($d = -3$ to 3) for the output digit selection function $z = SEL_{add}(\overline{W_E[k]})$ are shown in Table 3.1, where $\overline{W_E[k]}$ is the low-precision estimate of the even-indexed component of $W[k]$.

$RCNS_{2j,3}$ On-line Floating-Point Addition

/* Initialization */

$e_d = e_x - e_y$

e_z=max(e_x, e_y)

$W[-\delta + 1] = 0$

$z_0 = 0$

for $k = -\delta + 2$ to 0 do

$$(x'_{k+\delta-1}, y'_{k+\delta-1}) = \begin{cases} (0, y_{k+\delta-1}) & \text{if } e_d < 0 \\ (x_{k+\delta-1}, 0) & \text{if } e_d > 0 \\ (x_{k+\delta-1}, y_{k+\delta-1}) & \text{if } e_d = 0 \end{cases}$$

$$W[k] = 2j(W[k-1]) + (2j)^{-\delta+1}(x'_{k+\delta-1} + y'_{k+\delta-1})$$

end for

/* Recurrence */

for $k = 1$ to m do

$$(x'_{k+\delta-1}, y'_{k+\delta-1}) = \begin{cases} (0, y_{k+\delta-1}) & \text{if } k \le |e_d| \text{ and } e_d < 0 \\ (x_{k+\delta-1}, 0) & \text{if } k \le |e_d| \text{ and } e_d \ge 0 \\ (x_{k+\delta-1-|e_d|}, y_{k+\delta-1}) & \text{if } k > |e_d| \text{ and } e_d < 0 \\ (x_{k+\delta-1}, y_{k+\delta-1-|e_d|}) & \text{if } k > |e_d| \text{ and } e_d \ge 0 \end{cases}$$

$$W[k] = 2j(W[k-1] - z_{k-1}) + (2j)^{-\delta+1}(x'_{k+\delta-1} + y'_{k+\delta-1})$$

$z_k = SEL_{add}(\overline{W_E[k]})$

$e_z = NORM(z_k, e_z, \delta)$

end for

	-3	-2	-1	0	1	2	3
L_d	$-7/2$	$-5/2$	$-3/2$	$-1/2$	$1/2$	$3/2$	$5/2$
U_d	$-5/2$	$-3/2$	$-1/2$	$1/2$	$3/2$	$5/2$	$7/2$
m_d	$-7/2$	$-5/2$	$-3/2$	$-1/2$	$1/2$	$3/2$	$5/2$

Table 3.1: Selection points for $RCNS_{2j,3}$ on-line floating-point addition

3.2 FPGA implementation

A $RCNS_{2j,3}$ on-line floating adder to compute $z = x + y$ can be designed as a $RCNS_{2j,3}$ 2:1 borrow-save digit adder with appropriate insertion of flip-flops and alignment of operands, as shown in Figure 3.1. Alignment is accomplished by: (i) computing the difference of the exponents ($e_d = e_x - e_y$), (ii) swapping x with y if the difference is negative, such that

$$\begin{array}{llll} x' = x, & y' = y & \text{if } e_d \geq 0 \\ x' = y, & y' = x & \text{if } e_d < 0 \end{array} \tag{3.16}$$

and (iii) delaying the operand y' by inserting e_d flip-flops. The cost is summarized in Table 3.2, assuming e-bit exponent and m-digit significand precision. The SUBE unit computes the difference of the exponents, not considering exponent overflow/underflow. The ALIGN unit performs alignment of operand y'. The SWAP unit exchanges the operands if necessary. The NORM unit normalizes the result by updating the output exponent e_z.

Module	CLB slices
SUBE	e
ALIGN	$3m$
SWAP	e
PPM/MMP	4
NORM	$2e$
Total cost	$3m + 4e + 4$

Table 3.2: Cost of $RCNS_{2j,3}$ on-line floating-point adder

The total cost is $3m + 4e + 4$ CLB slices. The critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

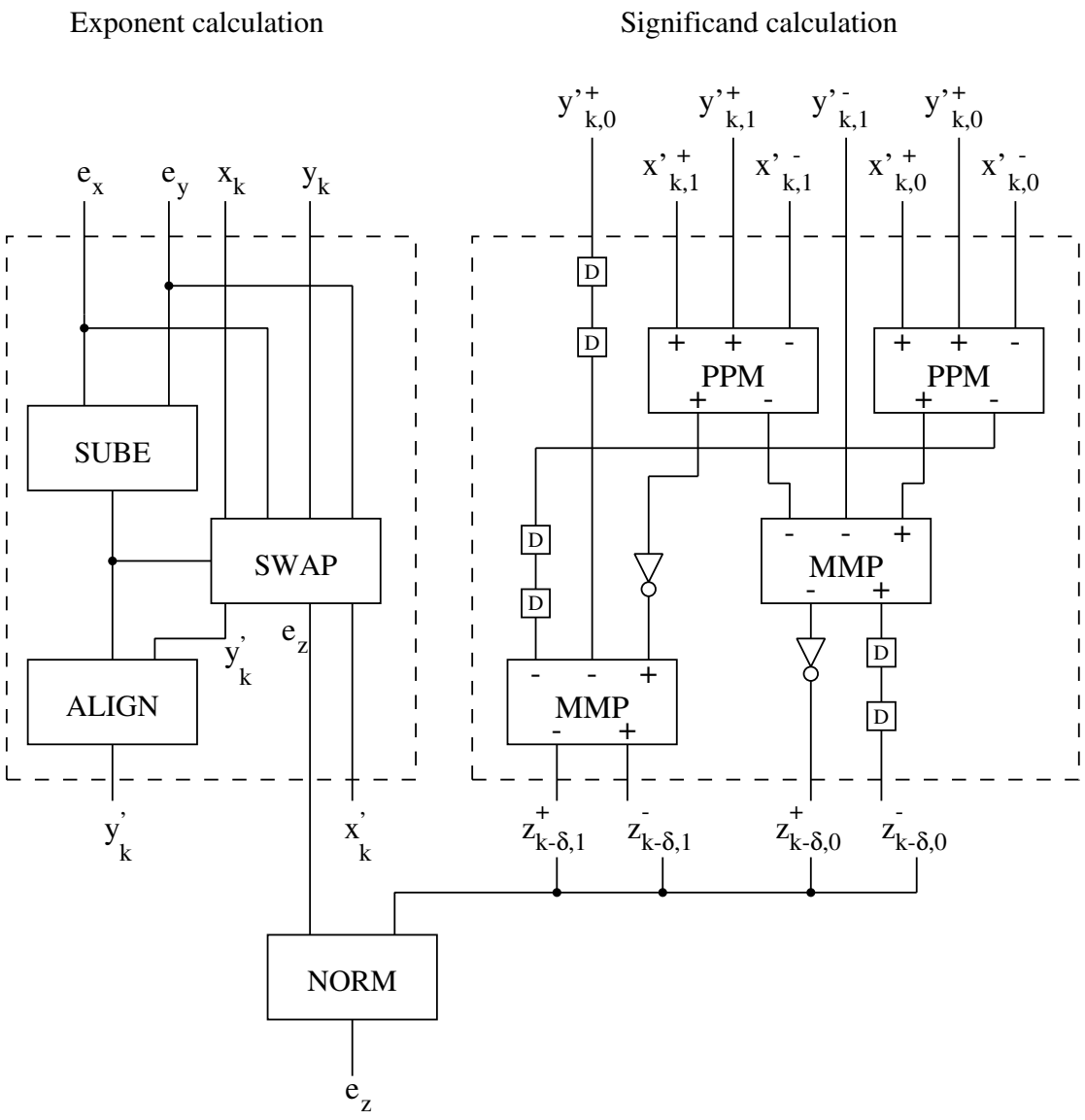


Figure 3.1: $RCNS_{2j,3}$ on-line floating-point adder

3.3 Comparison with real network

A complex number on-line floating point adder can be implemented according to the individual equations for the real and imaginary components, as shown in Equation (3.1) consisting of two radix-2 on-line floating-point adders. A comparison of the $RCNS_{2j,3}$ approach and the real network approach for the implementation of a m-digit significand and e-bit exponent complex number on-line floating point adder is shown in Table 3.3. The value δ is the minimum on-line delay, assuming no cancellation of leading digits in the result. In both cases, the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

For $m = 24$ and $e = 8$, the $RCNS_{2j,3}$ approach has a cost of 108 CLB slices, whereas the equivalent radix-2 network has a cost of 140 CLB slices.

Approach	CLB slices	δ
$RCNS_{2j,3}$	$3m + 4e + 4$	3
radix-2 network	$3m + 8e + 4$	3

Table 3.3: Comparison of approaches

Since the $RCNS_{2j,3}$ approach shares the exponent between the real and imaginary component, there is potential loss of precision for either the real or imaginary output significand, which does not occur in the radix-2 approach. For example, if $x = \frac{1}{256} + j\frac{1}{2}$ and $y = \frac{1}{512} + j\frac{1}{8}$, assuming 8-digit precision, then the $RCNS_{2j,3}$ approach will yield an inaccurate result

$$z = (0.\overline{1}0100001) \cdot (2j)^0 = \frac{1}{256} + j\frac{5}{8} \tag{3.17}$$

whereas the radix-2 approach will yield the accurate result

$$z = (0.11000000) \cdot 2^{-7} + j(0.10100000) \cdot 2^{0} = \frac{3}{512} + j\frac{5}{8} \tag{3.18}$$

The bounds on the loss of precision for either the real or imaginary component can be as great as the difference of the exponents of the individual real and imaginary components when both are normalized. Therefore, for an individual floating-point addition operation, sharing the exponent in complex number addition is not desirable.

However, for a network of complex number arithmetic operations, involving multiplication, division, or square root, in which the computed exponents for the individual real and imaginary components are based on the maximum of the individual real and imaginary exponents of the inputs, the loss of precision of one of the components is inevitable regardless of the appraoch. This is due to the fact that real networks for multiplication, division, and square root all require an addition/subtraction operation requiring alignment and hence potential loss of significance for fixed size representation.

3.4 Concluding Remarks

From the derivation of the algorithm and the implementation for complex number on-line floating-point addition, we make the following general observations.

1. The result can be unnormalized due to either being overnormalized or undernormalized, requiring special a normalization step. Overnormalized results

are handled by incrementing the exponent. Undernormalized results are handled by decrementing the result and discarding output digits of value zero. For an undernormalized result, normalization can result in an on-line delay as large as the precision of the operation. Exponent overflow and underflow are not considered.

2. The difference in cost between the $RCNS_{2j,3}$ and the radix-2 approach is due to the exponent precision, rather than the significand precision. Therefore, for a fixed exponent precision, an increase in significand precision yields negligible difference in cost between the two approaches.

3. The $RCNS_{2j,3}$ approach can produce loss of significance for either the real or imaginary component due to sharing the exponent, which may become negligible when the addition operation is part of a network.

CHAPTER 4

Complex Number On-line Floating-point Multiplication

Complex number floating-point multiplication ($z = xy$) is defined such that given inputs $x = (X_R + jX_I) \cdot (2j)^{e_x}$ and $y = (Y_R + jY_I) \cdot (2j)^{e_y}$, the output $z = (Z_R + jZ_I) \cdot (2j)^{e_z}$ is produced such that

$$\begin{aligned} Z_R &= X_R Y_R - X_I Y_I \\ Z_I &= X_R Y_I + X_I Y_R \\ e_z &= e_x + e_y \end{aligned} \tag{4.1}$$

For example, given $x = (\frac{9}{32} + j\frac{13}{32}) \cdot (2j)^2 = (-\frac{9}{8} - j\frac{13}{8})$ and $y = (\frac{11}{16} - j\frac{5}{32}) \cdot (2j)^{-1} = (\frac{5}{64} + j\frac{11}{32})$, then $z = (\frac{263}{1024} + j\frac{241}{1024}) \cdot (2j)^1 = (-\frac{241}{512} + j\frac{263}{512})$.

Section 4.1 presents the derivation of the algorithm for complex number on-line floating-point multiplication, including key characteristics such as on-line delay and the output digit selection function. Section 4.2 presents the modular design of a $RCNS_{2j,3}$ on-line floating-point multiplier. The design is mapped to a Virtex FPGA, and design parameters including cost and delay are determined. Section 4.3 compares the design with conventional networks of real number on-line arithmetic operators in terms of cost and delay. Section 4.4 considers various optimizations for complex number on-line floating-point multiplication for special types of operands and the effects on design parameters. Section 4.5 summarizes the chapter and reviews the main contributions.

4.1 Derivation of the algorithm

Assuming a real number negative radix $-r^2$, complex number on-line multiplication can be derived from the bound

$$\begin{array}{l} |(X_R[k]Y_R[k] - X_I[k]Y_I[k]) - Z_R[k]| < |(-r^2)^{-k}| \\ |(X_R[k]Y_I[k] + X_I[k]Y_R[k]) - Z_I[k]| < |(-r^2)^{-k}| \end{array} \tag{4.2}$$

The residuals are defined as

$$\begin{array}{l} W_R[k] = (-r^2)^k((X_R[k]Y_R[k] - X_I[k]Y_I[k]) - Z_R[k]) \\ W_I[k] = (-r^2)^k((X_R[k]Y_I[k] + X_I[k]Y_R[k]) - Z_I[k]) \end{array} \tag{4.3}$$

The recurrences are

$$\begin{array}{rl} W_R[k] = & -r^2(W_{R,k-1} - z_{R,k-1}) \\ & +(-r^2)^{-\delta+1}(X_R[k]y_{R,k+\delta-1} + Y_R[k-1]x_{R,k+\delta-1}) \\ & -(X_I[k]y_{I,k+\delta-1} + Y_I[k-1]x_{I,k+\delta-1})) \\ W_I[k] = & -r^2(W_{I,k-1} - z_{I,k-1}) \\ & +(-r^2)^{-\delta+1}(X_R[k]y_{I,k+\delta-1} + Y_I[k-1]x_{R,k+\delta-1}) \\ & +(X_I[k]y_{R,k+\delta-1} + Y_R[k-1]x_{I,k+\delta-1})) \end{array} \tag{4.4}$$

with initial conditions

$$\begin{array}{l} W_R[0] = X_R[0]Y_R[0] - X_I[0]Y_I[0] \\ W_I[0] = X_R[0]Y_I[0] + X_I[0]Y_R[0] \end{array} \tag{4.5}$$

Defining

$$W[k] = W_R[k] + rjW_I[k] \tag{4.6}$$

the two recurrences can be combined as

$$\begin{array}{rl} W[k] = & -r^2((W_R[k-1] + rjW_I[k-1]) - (z_{R,k-1} + rjz_{I,k-1})) + (-r^2)^{-\delta+1} \\ & ((X_R[k] + rjX_I[k])y_{R,k+\delta-1} + (Y_R[k-1] + rjY_I[k-1])x_{R,k+\delta-1} \\ & +(-X_I[k] + rjX_R[k])y_{I,k+\delta-1} + (-Y_I[k-1] + rjY_R[k-1])x_{I,k+\delta-1}) \end{array} \tag{4.7}$$

Defining

$$\begin{array}{l} X[k] = X_R[k] + rjX_I[k] \\ Y[k-1] = Y_R[k-1] + rjY_I[k-1] \\ rjX[k] = -X_I[k] + rjX_R[k] \\ rjY[k-1] = -Y_I[k-1] + rjY_R[k-1] \end{array} \tag{4.8}$$

the recurrence can be rewritten as

$$\begin{aligned} W[k] = \quad & (-r^2)(W[k-1] - (z_{R,k-1} + rjz_{I,k-1}) \\ & +(-r^2)^{-\delta+1}(X[k](y_{R,k+\delta-1} + rjy_{I,k+\delta-1}) \\ & +Y[k-1](x_{R,k+\delta-1} + rjx_{I,k+\delta-1}) \end{aligned} \tag{4.9}$$

Since $\sqrt{-r^2} = rj$, alternating the real and imaginary components yields recurrence equation

$$W[k] = (rj)(W[k-1] - z_{k-1}) + (rj)^{-\delta+1}(X[k]y_{k+\delta-1} + Y[k-1]x_{k+\delta-1}) \tag{4.10}$$

where

$$\begin{aligned} x_{k+\delta-1} &= \begin{cases} x_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjx_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ y_{k+\delta-1} &= \begin{cases} y_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjy_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ z_{k-1} &= \begin{cases} z_{R,k-1} & \text{if } k-1 \text{ is even} \\ rjz_{I,k-1} & \text{if } k-1 \text{ is odd} \end{cases} \end{aligned} \tag{4.11}$$

This can be decomposed into

$$W[k] = (rj)(W[k-1] + H_2[k]z_{k-1}) + H_1[k] \tag{4.12}$$

where

$$\begin{aligned} H_1[k] &= (rj)^{-\delta+1}(X[k]y_{k+\delta-1} + Y[k-1]x_{k+\delta-1}) \\ H_2[k] &= -1 \end{aligned} \tag{4.13}$$

The limits can be determined as

$$\begin{aligned} U_d &\leq \omega - H_2[k]d = d + \omega \\ L_d &\geq -\omega - H_2[k]d = d - \omega \end{aligned} \tag{4.14}$$

Under the containment condition

$$\omega = K(1 - 2r^{-\delta+1}) \tag{4.15}$$

The minimal on-line delay is

$$\delta = 2\left(-\log_{r^2}\left(\frac{(2K-1) - 2^{-t}}{4K}\right)\right) + 1 \tag{4.16}$$

where t is the number of fractional bits of the residual for the selection function. Using $RCNS_{2j,3}$ borrow-save representation, $r = 2$, $K = 1$, and $t = 3$. Then the on-line delay δ, the error ω, and the limits U_d, L_d are

$$\begin{aligned} \delta &= 5 \\ \omega &= \tfrac{7}{8} \\ U_d &= \tfrac{7}{8} + d \\ L_d &= -\tfrac{7}{8} + d \end{aligned} \tag{4.17}$$

The $RCNS_{2j,3}$ on-line floating-point multiplication algorithm is shown below. The selection points m_d ($d = -3$ to 3) for the output digit selection function $z = SEL_{mul}(\overline{W_E[k]})$ are shown in Table 4.1, where $\overline{W_E[k]}$ is the low-precision estimate of the even-indexed component of $W[k]$.

$RCNS_{2j,3}$ On-line Floating-Point Multiplication

/* Initialization */
$e_z = e_x + e_y$
$W[-\delta + 1] = 0$
$X[-\delta + 1] = 0$
$Y[-\delta + 1] = 0$
$z_0 = 0$

for $k = -\delta + 2$ to 0 do
$X[k] = X[k-1] + x_{k+\delta-1}(2j)^{-k-\delta+1}$
$W[k] = (2j)(W[k-1]) + (2j)^{-\delta+1}(X[k]y_{k+\delta-1} + Y[k-1]x_{k+\delta-1}$
$Y[k] = Y[k-1] + y_{k+\delta-1}(2j)^{-k-\delta+1}$
end for

/* Recurrence */
for $k = 1$ to m do
$X[k] = X[k-1] + x_{k+\delta-1}(2j)^{-k-\delta+1}$
$W[k] = (2j)(W[k-1] - z_{k-1}) + (2j)^{-\delta+1}(X[k]y_{k+\delta-1} + Y[k-1]x_{k+\delta-1})$
$z_k = SEL_{mul}(\overline{W_E[k]})$
$Y[k] = Y[k-1] + y_{k+\delta-1}(2j)^{-k-\delta+1}$
$e_z = NORM(z_k, e_z, \delta)$
end for

	-3	-2	-1	0	1	2	3
L_d	$-31/8$	$-23/8$	$-15/8$	$-7/8$	$1/8$	$9/8$	$17/8$
U_d	$-17/8$	$-9/8$	$-1/8$	$7/8$	$15/8$	$23/8$	$31/8$
m_d	$-7/2$	$-5/2$	$-3/2$	$-1/2$	$1/2$	$3/2$	$5/2$

Table 4.1: Selection points for $RCNS_{2j,3}$ on-line floating-point multiplication

4.2 FPGA implementation

A $RCNS_{2j,3}$ on-line floating-point multiplier can be designed as a series of modular slices, where each slice consists of two borrow-save digit multipliers, a 3:1 borrow-save digit adder, a pair of digit-wide latches, a D flip-flop, and a digit-wide register of D flip-flops. The operands x_k and y_k are recoded into digit set $\{-2, \ldots, 2\}$ using two DSREC units. The two most significant digits of the recurrence are determined using two MSREC units which perfom output digit selection as well as handle potential most significant carry-out bits from the adders. The ADDE unit adds the two input exponents to produce the exponent of the output, not considering exponent overflow/underflow. The NORM unit normalizes the result by updating the output exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point multiplier is shown in Figure 4.1. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table 4.2.

The total cost is $16m + 3e + 44$ CLB slices and the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

Module	Count	CLB slices
ADDE	1	e
DSREC	2	24
BSD mult. ($\otimes$)	$2m$	$8m$
BSD adder (3:1)	m	$8m$
MSREC	2	20
NORM	1	$2e$
Total cost		$16m + 3e + 44$

Table 4.2: Cost of $RCNS_{2j,3}$ on-line floating-point multiplier

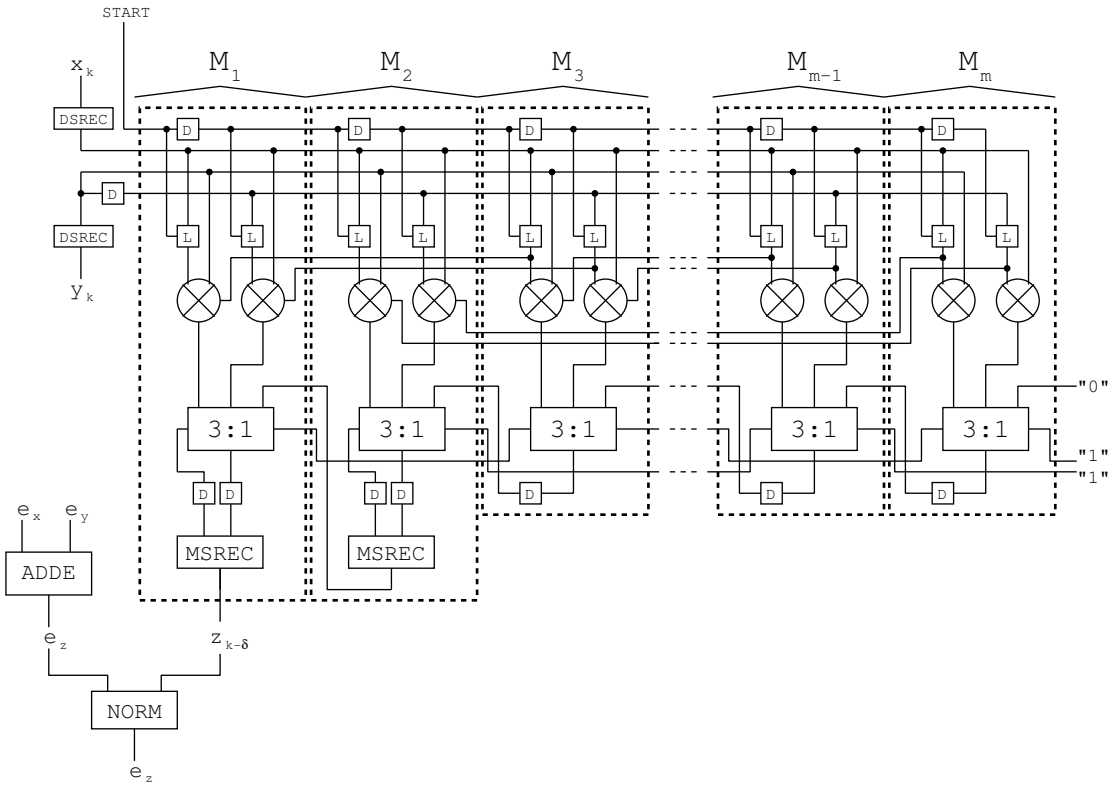


Figure 4.1: $RCNS_{2j,3}$ on-line floating-point multiplier

4.3 Comparison with real network

A complex number on-line floating-point multiplier can be implemented according to the individual equations for the real and imaginary components, as shown in Equation (4.1) consisting of four radix-2 on-line floating-point multipliers and two radix-2 floating-point adders. Another approach to save one multiplication, with an implementation cost of three radix-2 on-line floating-point multipliers and five radix-2 on-line floating-point adders, is to evaluate Z as

$$\begin{aligned} M &= Y_I(X_R - X_I) \\ Z_R &= M + X_R(Y_R - Y_I) \\ Z_I &= M + X_I(Y_R + Y_I) \end{aligned} \tag{4.18}$$

A comparison of the $RCNS_{2j,3}$ approach and the real network approaches for the implementation of a m-digit significand and e-bit exponent complex number on-line floating-point multiplier is shown in Table 4.3. In all approaches, the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

Approach	CLB slices	δ
$RCNS_{2j,3}$	$16m + 3e + 44$	5
Radix-2 network (1st approach)	$27m + 20e + 60$	7
Radix-2 network (2nd approach)	$\lceil 25.5m + 29e + 52 \rceil$	10

Table 4.3: Comparison of approaches

For $m = 24$ and $e = 8$, the $RCNS_{2j,3}$ approach has a cost of 452 CLB slices, whereas the 1st radix-2 network approach has a cost of 868 CLB slices, and the 2nd radix-2 network approach has a cost of 896 CLB slices.

4.4 Optimizations for special operands

The algorithm and implementation for complex number on-line floating-point multiplication can be optimized when the operands are of a special form. These include: (i) when the operands are the same (denoted *complex number on-line floating-point square*); and (ii) when the operands are complex conjugates of each other (denoted *complex number on-line floating-point conjugate multiplication*). The effects on the recurrence equation as well as the implementation, and comparisons with real number on-line arithmetic operators networks, are considered for each case.

4.4.1 Complex number on-line floating-point square

When the operands are known in advance to be the same, the operation can be denoted as complex-number on-line floating-point square ($z = x^2$), where the output z is produced such that

$$\begin{aligned} Z_R &= X_R^2 - X_I^2 \\ Z_I &= 2X_R X_I \\ e_z &= 2e_x \end{aligned} \tag{4.19}$$

Then the recurrence can be determined as

$$W[k] = 2j(W[k-1] - z[k-1]) + (2j)^{-\delta+1}((2X[k-1] + x_{k+\delta-1}(2j)^{-k-\delta+1})x_{k+\delta-1}) \tag{4.20}$$

A $RCNS_{2j,3}$ floating-point square unit can be designed as a series of modular slices, where each slice consists of a borrow-save digit multiplier, a 2:1 borrow-save

digit adder, a digit-wide latch, a D flip-flop, a digit-wide register of D flip-flops, a $TWICE$ unit for computing $2X[k-1]$, and a 2-to-1 MUX for appropriately appending digit $x_{k+\delta-1}$ to vector $2X[k-1]$. The digit x_k is recoded into digit set $\{-2,\ldots,2\}$ using one DSREC unit. The two most significant digits of the recurrence are determined using two MSREC units, which perform output digit selection as well as handle most significant carry-out bits produced from the adders. The MULT2E unit multiplies the operand exponent by two (left shift one position) to produce the exponent of the output, not considering exponent overflow/underflow. The NORM unit normalizes the result by updating the output exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point square unit is shown in Figure 4.2. The number of individual module type utilized, the cost per module type, and the total overall cost are summarized in Table 4.4.

Module	Count	CLB slices
MULT2E	1	e
DSREC	1	12
2-to-1 MUX	m	$2m$
BSD mult. ($\otimes$)	m	$4m$
BSD adder (2:1)	m	$4m$
MSREC	2	20
NORM	1	$2e$
Total cost		$10m+e+32$

Table 4.4: Cost of $RCNS_{2j,3}$ on-line floating-point square unit

The total cost is $10m+3e+32$ CLB slices and the critical delay is $T_{CKO}+T_{ICK}+4T_{ILO}+5T_{NET}=(4.5+5T_{NET})$ns.

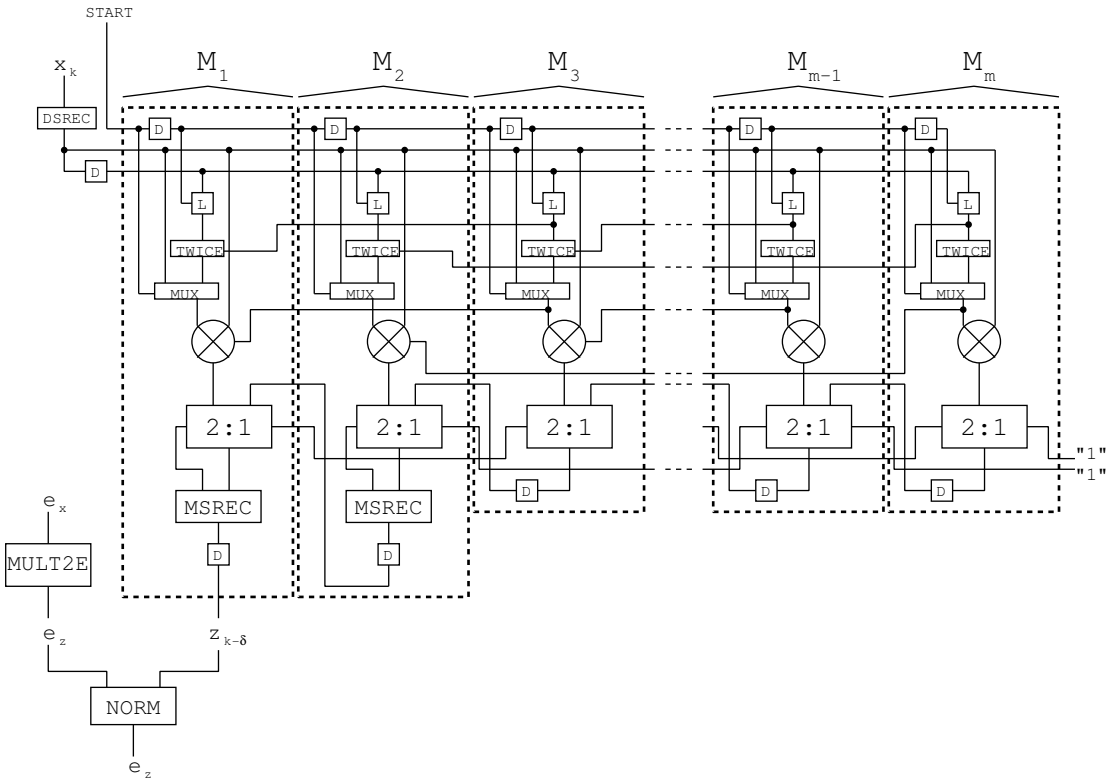


Figure 4.2: $RCNS_{2j,3}$ on-line square unit

Alternatively, a complex number on-line floating-point square unit can be implemented according to the individual equations for the real and imaginary components, as shown in Equation (4.19) consisting of 2 real on-line square units, 1 real on-line multiplier unit, and 1 real on-line adder. A comparison of the $RCNS_{2j,3}$ approach and the radix-2 network approach is shown in Table 4.5. In both approaches, the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

Approach	CLB slices	δ
$RCNS_{2j,3}$	$10m + 3e + 32$	5
radix-2 network	$\lceil 15.5m + 13e + 44 \rceil$	5

Table 4.5: Comparison of approaches

For $m = 24$ and $e = 8$, the $RCNS_{2j,3}$ approach has a cost of 296 CLB slices, whereas the radix-2 network approach has a cost of 520 CLB slices.

4.4.2 Complex number on-line floating-point conjugate multiplication

When the operands are complex conjugates of each other, denoting the complex conjugate of x as $x^* = (X_R - jX_I) \cdot (2j)^{e_x}$, the output z is produced such that

$$\begin{aligned} Z_R &= X_R^2 + X_I^2 \\ Z_I &= 0 \\ e_z &= 2e_x \end{aligned} \tag{4.21}$$

Since

$$\begin{aligned} x^*_{k+\delta-1} &= \begin{cases} x_{k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ -x_{k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ z_{k-1} &= \begin{cases} z_{R,k-1} & \text{if } k \text{ is even} \\ 0 & \text{if } k \text{ is odd} \end{cases} \end{aligned} \tag{4.22}$$

the recurrence can be determined as

$$W[k] = \begin{cases} (2j)(W[k-1]) + (2j)^{-\delta+1}((2X_R[k-1] \\ +x_{k+\delta-1}(2j)^{-k-\delta+1})x_{k+\delta-1}) & \text{if } k+\delta-1 \text{ is even} \\ (2j)(W[k-1] - z_{k-1}) + (2j)^{-\delta+1}(-2X_I[k-1] \\ -x_{k+\delta-1}(2j)^{-k-\delta+1})x_{k+\delta-1}) & \text{if } k+\delta-1 \text{ is odd} \end{cases} \tag{4.23}$$

For implementation, two types of modular slices are required. An odd-indexed slice M_{2k-1} ($k = 1$ to $m/2$) consists of one borrow-save multiplier, a 2:1 borrow-save digit adder, a digit-wide latch, a D flip-flop, a digit-wide register of D flip-flops, a TWICE unit for computing $2X[k-1]$, a 3-to-1 MUX for appropriately appending digit x_k to vector $2X[k-1]$, and a 2-to-1 MUX for appropriately shifting the residual. An even-indexed slice M_{2k} ($k = 1$ to $m/2$) consists of a digit-wide latch, a bit-wide D flip-flop, and a 2X unit for computing $2X[k-1]$. The CONJ unit generates digits of x^*. The digit x_k is recoded into digit set $\{-2, \ldots, 2\}$ using one DSREC unit. The most-significant digit of the recurrence is determined using one MSREC unit, which performs output digit selection as well as handles most significant carry-out bits of the adder. The MULT2E unit multiplies the operand exponent by two to produce the output exponent e_z, not considering exponent overflow/underflow. The NORM unit normalizes the result by updating the output exponent e_z. The design of a m-digit significand and e-bit exponent $RCNS_{2j,3}$ on-line floating-point conjugate multiplier unit is shown in Figure 4.3. The number of individual modules types utilized, the cost per module type, and the total overall cost are summarized in Table 4.6. The total cost is $\lceil 7.25m + 3e + 26 \rceil$ CLB slices and the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

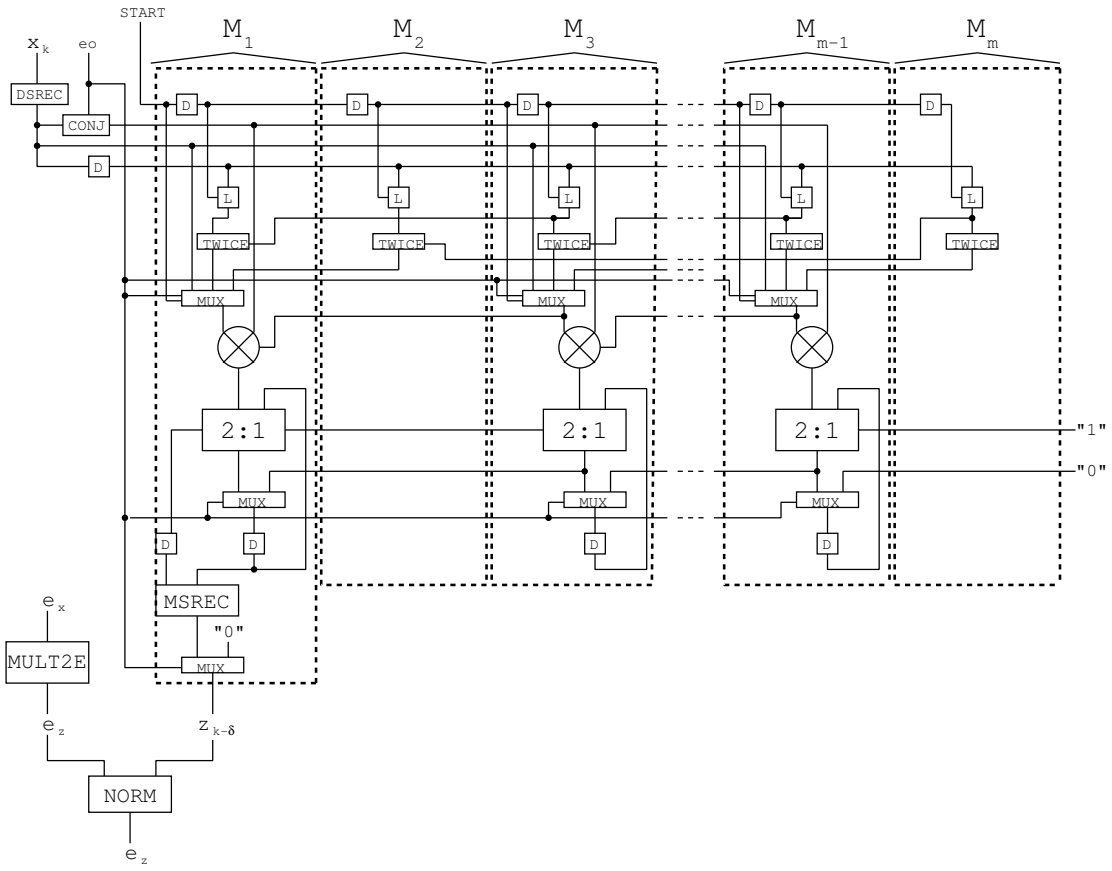


Figure 4.3: $RCNS_{2j,3}$ on-line floating-point conjugate multiplier

Module	Count	CLB slices
MULT2E	1	e
CONJ	1	4
DSREC	1	12
3-to-1 MUX	$\frac{m}{2}$	$2.25m$
2-to-1 MUX	$\frac{m}{2}$	m
BSD mult. ($\otimes$)	$\frac{m}{2}$	$2m$
BSD adder (2:1)	$\frac{m}{2}$	$2m$
MSREC	1	10
NORM	1	$2e$
Total cost		$\lceil 7.25m + 3e + 26 \rceil$

Table 4.6: Cost of $RCNS_{2j,3}$ on-line floating-point conjugate multiplier

Alternatively, a complex-number on-line floating-point conjugate multiplier unit can be implemented according to the equation for the real component, as shown in Equation (4.21), consisting of two radix-2 on-line floating-point square units and one radix-2 on-line floating-point adder. A comparison of the $RCNS_{2j,3}$ approach and the real network approach is shown in Figure 4.7. In both approaches, the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

Approach	CLB slices	δ
$RCNS_{2j,3}$	$\lceil 7.25m + 3e + 26 \rceil$	5
radix-2 network	$\lceil 9.5m + 10e + 30 \rceil$	7

Table 4.7: Comparison of approaches

For $m = 24$ and $e = 8$, the $RCNS_{2j,3}$ approach has a cost of 224 CLB slices, wheareas the radix-2 network approach has a cost of 338 CLB slices.

4.5 Concluding Remarks

From the derivation of the algorithm and the implemenation for complex number on-line floating-point multiplication, we make the following general observations.

1. The result can be unnormalized due to either being overnormalized or undernormalized, requiring special a normalization step. Overnormalized results are handled by incrementing the exponent. Undernormalized results are handled by decrementing the result and discarding output digits of value zero. For complex number on-line floating-point multiplication, since the result is in the range $\frac{1}{16} \leq \max(|Z_R|, |Z_I|) < 2$, at most two steps for normalization are necessary to make the result quasi-normalized. Exponent overflow and underflow are not considered.

2. The $RCNS_{2j,3}$ approach for complex number on-line floating-point multiplication achieves approximately a 40% decrease in cost compared to the first radix-2 network approach, and a 37% decrease in cost compared to the second radix-2 network approach as the precision m increases, for fixed exponent precision.

3. The $RCNS_{2j,3}$ approach for complex number on-line floating-point square achieves approximately a 35% decrease in cost compared to the radix-2 network approach as the precision m increases, for fixed exponent precision.

4. The $RCNS_{2j,3}$ approach for complex number on-line floating-point conjugate multiplication achieves approximately a 23% decrease in cost compared to the radix-2 network approach as the precision m increases, for fixed exponent precision.

5. The $RCNS_{2j,3}$ approach for general complex number on-line floating-point multiplication can lead to loss of precision for either the real or imaginary component of the output due to sharing the exponent, when one of the components is undernormalized, whereas in the radix-2 network approach the real and imaginary components of the output are normalized independently. The tradeoff is that there is no need for a digit cancallation step in the $RCNS_{2j,3}$ approach, whereas the radix-2 network approach may require digit cancellation due to internal floating-point real addition/subtraction operations, resulting in an on-line delay as large as the precision of the result.

CHAPTER 5

Complex Number On-line Floating-point Division

Complex number floating-point division ($z = x/y$) is defined such that given inputs $x = (X_R + jX_I) \cdot (2j)^{e_x}$ and $y = (Y_R + jY_I) \cdot (2j)^{e_y}$, the output $z = (Z_R + jZ_I) \cdot (2j)^{e_z}$ satisfies

$$\begin{aligned} Z_R &= \tfrac{X_R Y_R + X_I Y_I}{Y_R^2 + Y_I^2} \\ Z_I &= \tfrac{X_I Y_R - X_R Y_I}{Y_R^2 + Y_I^2} \\ e_z &= e_x - e_y \end{aligned} \tag{5.1}$$

For example, given inputs $x = (-\frac{223}{1024} - j\frac{97}{512}) \cdot (2j)^1 = \frac{97}{256} + j\frac{223}{512}$ and $y = (-\frac{1}{2} + j\frac{13}{4}) \cdot (2j)^{-2} = \frac{1}{8} - j\frac{13}{16}$ then $z = (-\frac{3}{64} + j\frac{19}{256}) \cdot (2j)^3 = \frac{19}{32} + j\frac{3}{8}$.

Section 5.1 presents the derivation of the algorithm for complex number on-line floating-point division, including key characteristics such as on-line delay and the output digit selection function. Section 5.2 presents the modular design of a $RCNS_{2j,3}$ on-line floating-point divider. The design is mapped to a Virtex FPGA, and design parameters including cost and delay are determined. Section 5.3 compares the design with conventional networks of real number on-line arithmetic operators in terms of cost and delay. Section 5.4 summarizes the chapter and reviews the main contributions.

5.1 Derivation of the algorithm

Assuming a real number negative radix $-r^2$, and transforming the functions, complex number on-line division can be derived from the bound

$$\begin{array}{l} |(X_R[k]Y_R[k] + X_I[k]Y_I[k]) - Z_R[k](Y_R[k]Y_R[k] + Y_I[k]Y_I[k])| < |(-r^2)^{-k}| \\ |(X_I[k]Y_R[k] - X_R[k]Y_I[k]) - Z_I[k](Y_R[k]Y_R[k] + Y_I[k]Y_I[k])| < |(-r^2)^{-k}| \end{array} \tag{5.2}$$

Since

$$\begin{array}{l} Y_R[k]Y_R[k] + Y_I[k]Y_I[k] = Y[k]Y^*[k] \\ X_R[k]Y_R[k] + X_I[k]Y_I[k] = Re(X[k]Y^*[k]) \\ X_I[k]Y_R[k] - X_R[k]Y_I[k] = Im(X[k]Y^*[k]) \end{array} \tag{5.3}$$

where $Y^*[k]$ is the complex conjugate of $Y[k]$, then factoring out $Y^*[k]$ yields

$$\begin{array}{l} |X_R[k] - (Z_R[k]Y_R[k] - Z_I[k]Y_I[k])| < |(-r^2)^{-k}| \\ |X_I[k] - (Z_R[k]Y_I[k] + Z_I[k]Y_R[k])| < |(-r^2)^{-k}| \end{array} \tag{5.4}$$

The residuals are defined as

$$\begin{array}{l} W_R[k] = (-r^2)^k(X_R[k] - (Z_R[k]Y_R[k] - Z_I[k]Y_I[k])) \\ W_I[k] = (-r^2)^k(X_I[k] - (Z_R[k]Y_I[k] - Z_I[k]Y_R[k])) \end{array} \tag{5.5}$$

The recurrences are

$$\begin{array}{ll} W_R[k] = & (-r^2)(W_R[k-1] - Y_R[k]z_{R,k} + Y_I[k]z_{I,k}) \\ & +(-r^2)^{-\delta+1}(x_{R,k+\delta-1} - Z_R[k-1]y_{R,k+\delta-1} + Z_I[k-1]y_{I,k+\delta-1}) \\ W_I[k] = & (-r^2)(W_R[k-1] - Y_I[k]z_{R,k} - Y_R[k]z_I[k]) \\ & +(-r^2)^{-\delta+1}(x_{I,k+\delta-1} - Z_I[k-1]y_{R,k+\delta-1} - Z_R[k-1]y_{I,k+\delta-1}) \end{array} \tag{5.6}$$

with initial conditions

$$\begin{array}{l} W_R[0] = X_R[0] \\ W_I[0] = X_I[0] \end{array} \tag{5.7}$$

Defining

$$W[k] = W_R[k] + rjW_I[k] \tag{5.8}$$

the two recurrences can be combined as

$$\begin{aligned} W[k] = \; & (-r^2)((W_R[k-1] + rjW_I[k-1]) \\ & -(Y_R[k] + rjY_I[k])z_{R,k} - (-Y_I[k] + rjY_R[k])z_{I,k}) \\ & +(-r^2)^{-\delta+1}(x_{R,k+\delta-1} + rjx_{I,k+\delta-1} - (Z_R[k-1] + rjZ_I[k-1])y_{R,k+\delta-1} \\ & -(rjZ_{R,k-1} - Z_{I,k-1})y_{I,k+\delta-1}) \end{aligned} \tag{5.9}$$

Defining

$$\begin{aligned} & Z[k-1] = Z_R[k-1] + rjZ_I[k-1] \\ & Y[k] = Y_R[k] + rjY_I[k] \\ & rjZ[k-1] = -Z_I[k-1] + rjZ_R[k-1] = rj(Z_R[k-1] + rjZ_I[k-1]) \\ & rjY[k] = -Y_I[k] + rjY_R[k] = rj(Y_R[k] + rjY_I[k]) \end{aligned} \tag{5.10}$$

the recurrence can be rewritten as

$$\begin{aligned} W[k] = \; & (-r^2)(W[k-1] - Y[k]z_k) \\ & +(-r^2)^{-\delta+1}((x_{R,k+\delta-1} + rjx_{I,k+\delta-1}) - Z[k-1](y_{R,k+\delta-1} + rjy_{I,k+\delta-1})) \end{aligned} \tag{5.11}$$

Since $\sqrt{-r^2} = rj$, alternating the real and imaginary components yields recurrence equation

$$W[k] = (rj)(W[k-1] - Y[k]z_{k-1}) + (rj)^{-\delta+1}(x_{k+\delta-1} - Z[k-1]y_{k+\delta-1}) \tag{5.12}$$

where

$$\begin{aligned} x_{k+\delta-1} &= \begin{cases} x_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjx_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ y_{k+\delta-1} &= \begin{cases} y_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjy_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ z_{k-1} &= \begin{cases} z_{R,k-1} & \text{if } k-1 \text{ is even} \\ rjz_{I,k-1} & \text{if } k-1 \text{ is odd} \end{cases} \end{aligned} \tag{5.13}$$

This can be decomposed into

$$W[k] = (rj)(W[k-1] + H_2[k]z_{k-1}) + H_1[k] \tag{5.14}$$

where

$$\begin{aligned} & H_1[k] = r^{-\delta+1}(x_{k+\delta-1} - Z[k-1]y_{k+\delta-1}) \\ & H_2[k] = -Y[k] \end{aligned} \tag{5.15}$$

The limits can be determined as

$$\begin{aligned} U_d &\leq \omega - |H_2[k]|d = |Y[k]|d + \omega \\ L_d &\geq -\omega - |H_2[k]|d = |Y[k]|d - \omega \end{aligned} \tag{5.16}$$

where $|Y_[k]|$ is the absolute value of $Y[k]$ defined as

$$|Y[k]| = \sqrt{Y_R[k]^2 + Y_I[k]^2} \tag{5.17}$$

Under the containment condition, the residual bound is

$$\omega = K(|Y[k]| - 2r^{-\delta+1}) \tag{5.18}$$

The minimal on-line delay is

$$\delta = 2\left(-\log_{r^2}\left(\frac{(2K-1)|Y[k]| - 2^{-t}}{4K}\right)\right) + 1 \tag{5.19}$$

where t is the number of fractional bits of the residual for the selection function. Using $RCNS_{2j,3}$ borrow-save representation, $r = 2$ and $K = 1$, and $t = 3$. Then the on-line delay δ, the error ω, and the limits U_d, L_d are

$$\begin{aligned} &\delta = 5 \\ &\omega = |Y[k]| - \tfrac{1}{8} \\ &U_d = |Y[k]|(d+1) - \tfrac{1}{8} \\ &L_d = |Y[k]|(d-1) + \tfrac{1}{8} \end{aligned} \tag{5.20}$$

The $RCNS_{2j,3}$ on-line floating-point division algorithm is shown below. The selection points m_d ($d = -3$ to 3) for the output digit selection function $z = SEL_{div}(\overline{W_E[k]}, \overline{|Y[k]|})$ are shown in Table 5.1, where $\overline{W_E[k]}$ is the low-precision estimate of the even-indexed component of $W[k]$, and $\overline{|Y[k]|}$ is the low-precision estimate of the complex absolute value of $Y[k]$ defined in Equation 5.17.

$RCNS_{2j,3}$ On-line Floating-Point Division

/* Initialization */

$e_z = e_x - e_y$

$W[-\delta+1] = Y[-\delta+1] = 0$

$Z[0] = 0$

$z_0 = 0$

for $k = -\delta+2$ to 0 do

$W[k] = (2j)W[k-1] + (2j)^{-\delta+1} x_{k+\delta-1}$

$Y[k] = Y[k-1] + y_{k+\delta-1}(2j)^{-k-\delta+1}$

end for

/* Recurrence */

for $k = 1$ to m do

$W[k] = 2j(W[k-1] - Y[k-1]z_{k-1}) + (2j)^{-\delta+1}(x_{k+\delta-1} - Z[k-1]y_{k+\delta-1})$

$Y[k] = Y[k-1] + y_{k+\delta-1}(2j)^{-k-\delta+1}$

$z_k = SEL_{div}(\overline{W_E[k]}, \overline{|Y[k]|})$

$Z[k] = Z[k-1] + z_k(2j)^{-k}$

$e_z = NORM(e_z, z_k, \delta)$

end for

$\overline{\lvert Y[k]\rvert}$	L_{-3}, U_{-3} m_{-3}	L_{-2}, U_{-2} m_{-2}	L_{-1}, U_{-1} m_{-1}	L_0, U_0 m_0	L_1, U_1 m_1	L_2, U_2 m_2	L_3, U_3 m_3
$\frac{4}{8}$	$-\frac{15}{8}, -\frac{9}{8}$ $-\frac{7}{4}$	$-\frac{11}{8}, -\frac{5}{8}$ $-\frac{5}{4}$	$-\frac{7}{8}, -\frac{1}{8}$ $-\frac{3}{4}$	$-\frac{3}{8}, \frac{3}{8}$ $-\frac{1}{4}$	$\frac{1}{8}, \frac{7}{8}$ $\frac{1}{4}$	$\frac{5}{8}, \frac{11}{8}$ $\frac{3}{4}$	$\frac{9}{8}, \frac{15}{8}$ $\frac{5}{4}$
$\frac{5}{8}$	$-\frac{19}{8}, -\frac{11}{8}$ -2	$-\frac{14}{8}, -\frac{6}{8}$ $-\frac{3}{2}$	$-\frac{9}{8}, -\frac{1}{8}$ -1	$-\frac{4}{8}, \frac{4}{8}$ $-\frac{1}{2}$	$\frac{1}{8}, \frac{9}{8}$ $\frac{1}{2}$	$\frac{6}{8}, \frac{14}{8}$ 1	$\frac{11}{8}, \frac{19}{8}$ $\frac{3}{2}$
$\frac{6}{8}$	$-\frac{23}{8}, -\frac{13}{8}$ $-\frac{5}{2}$	$-\frac{17}{8}, -\frac{7}{8}$ -2	$-\frac{11}{8}, -\frac{1}{8}$ -1	$-\frac{5}{8}, \frac{5}{8}$ $-\frac{1}{2}$	$\frac{1}{8}, \frac{11}{8}$ $\frac{1}{2}$	$\frac{7}{8}, \frac{17}{8}$ 1	$\frac{13}{8}, \frac{23}{8}$ 2
$\frac{7}{8}$	$-\frac{27}{8}, -\frac{15}{8}$ -3	$-\frac{20}{8}, -\frac{8}{8}$ -2	$-\frac{13}{8}, -\frac{1}{8}$ -1	$-\frac{6}{8}, \frac{6}{8}$ $-\frac{1}{2}$	$\frac{1}{8}, \frac{13}{8}$ $\frac{1}{2}$	$\frac{8}{8}, \frac{20}{8}$ 1	$\frac{15}{8}, \frac{27}{8}$ 2

Table 5.1: Selection points for $RCNS_{2j,3}$ on-line floating-point division

5.2 FPGA implementation

A $RCNS_{2j,3}$ on-line floating-point divider can be designed as a series of modular slices, where each slice consists of two borrow-save digit multipliers, a 3:1 borrow-save digit adder, a pair of digit-wide latches, a D flip-flop, and a digit-wide register of D flip-flops. The operand digit y_k and the output digit z_k are recoded into digit set $\{-2, \ldots, 2\}$ using two DSREC units. The most significant digits of the recurrence are determined using two MSREC units which handle potential most significant carry-out bits from the adders. The SELDIV unit, selects the output digit z_k. The SUBE unit subtracts the two exponents to produce the exponent of the output, not considering exponent overflow/underflow. The digit $x_{k+\delta-1}$ is appended to the most significant end of the vector product $Z[k-1]y_{k+\delta-1}$. The NORM unit normalizes the result by updating the exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point divider is shown in Figure 5.1. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table 5.2.

The total cost is $16m + 3e + 137$ CLB slices and the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

5.3 Comparison with real network

A complex number on-line floating-point divider can be implemented according to the individual equations for the real and imaginary components, as shown in

Module	Count	CLB slices
SUBE	1	e
DSREC	2	24
BSD mult. ($\otimes$)	$2m$	$8m$
BSD adder (3:1)	m	$8m$
MSREC	2	20
SELDIV	1	93
NORM	1	$2e$
Total cost		$16m + 3e + 137$

Table 5.2: Cost of $RCNS_{2j,3}$ on-line floating-point divider

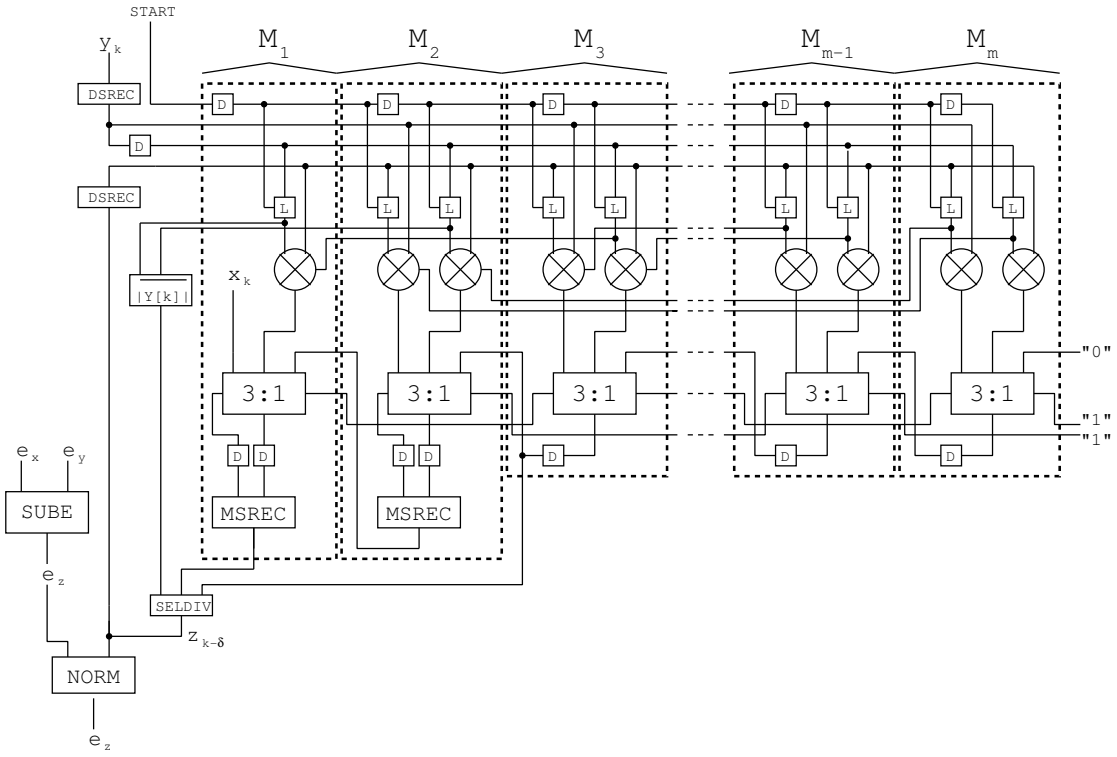


Figure 5.1: $RCNS_{2j,3}$ on-line floating-point divider

Equation (5.1) consisting of four radix-2 on-line floating-point multipliers, two radix-2 on-line floating-point dividers, three radix-2 on-line floating-point adders, and two radix-2 on-line floating-point square operators. Another approach, to save one multiplication, with an implementation cost of three radix-2 on-line floating-point multipliers, two radix-2 on-line floating-point dividers, six radix-2 on-line floating-point adders, and 2 radix-2 on-line floating-point square operators, is to evaluate z as

$$\begin{aligned} M &= Y_I(X_I - X_R) \\ Z_R &= \frac{M + X_R(Y_R + Y_I)}{Y_R^2 + Y_I^2} \\ Z_I &= \frac{M + X_I(Y_R - Y_I)}{Y_R^2 + Y_I^2} \end{aligned} \tag{5.21}$$

A comparison of the $RCNS_{2j,3}$ approach and the real network approaches for the implementation of a m-digit significand and e-bit exponent complex number on-line floating-point divider, is shown in Table 5.3. In all approaches, the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

Approach	CLB slices	δ
$RCNS_{2j,3}$	$16m + 3e + 137$	5
Radix-2 network (1st approach)	$\lceil 48.5m + 36e + 50 \rceil$	12
Radix-2 network (2nd approach)	$47m + 45e + 54$	15

Table 5.3: Comparison of approaches

For $m = 24$ and $e = 8$, the $RCNS_{2j,3}$ approach has a cost of 545 CLB slices, wheareas the 1st radix-2 network approach has a cost of 1502 CLB slices, and the 2nd radix-2 network approach has a cost of 1542 CLB slices.

5.4 Concluding Remarks

From the derivation of the algorithm and the implementation for complex number on-line floating-point division, we make the following general observations.

1. The result can be unnormalized due to either being overnormalized or undernormalized, requiring a special normalization step. Overnormalized results are handled by incrementing the exponent. Undernormalized results are handled by decrementing the result and discarding output digits of value zero. For complex number on-line floating-point division, since the result is in the range $\frac{1}{8} \leq \max(|Z_R|, |Z_I|) < 4$, at most one step for normlization is necessary to make the result quasi-normalized. Exponent overflow and underflow are not considered.

2. The $RCNS_{2j,3}$ approach for complex number on-line floating-point division achieves approximately a 67% decrease in cost compared to the first radix-2 network approach, and a 66% decrease in cost compared to the second radix-2 network approach as the precision m increases, for fixed exponent precision.

3. The $RCNS_{2j,3}$ approach for general complex number on-line floating-point division can lead to loss of precision for either the real or imaginary component of the output due to sharing the exponent, when one of the components is undernormalized, whereas in the radix-2 network approach the real and

imaginary components of the output are normalized independently. The tradeoff is that there is no need for a digit cancellation step in the $RCNS_{2j,3}$ approach, whereas the radix-2 network approach may require digit cancellation due to internal floating-point real addition/subtraction operations, resulting in an on-line delay as large as the precision of the result.

CHAPTER 6

Complex Number On-line Floating-point Square Root

Complex number floating-point square root ($z = \sqrt{x}$) is defined such that given input $x = (X_R + jX_I) \cdot (2j)^{e_x}$, the output $z = (Z_R + jZ_I) \cdot (2j)^{e_z}$ satisfies

$$\begin{aligned} Z_R &= \sqrt{\frac{X_R + \sqrt{X_R^2 + X_I^2}}{2}} \\ Z_I &= sign(X_I)\sqrt{\frac{-X_R + \sqrt{X_R^2 + X_I^2}}{2}} \\ e_z &= \lceil \frac{e_x}{2} \rceil \end{aligned} \tag{6.1}$$

To ensure e_z is an integer, if e_x is odd, then the mantissa X is initially scaled by $(2j)^{-1}$ and e_x is incremented by 1 to make e_x even. For example, given input $x = (\frac{117}{256} + j\frac{11}{64}) \cdot (2j)^{-3} = \frac{11}{128} - j\frac{117}{512}$, since the exponent e_x is odd, x is scaled as $x = (\frac{11}{32} - j\frac{117}{128}) \cdot (2j)^{-2}$. Then $z = (\frac{13}{16} + j\frac{9}{16}) \cdot (2j)^{-1} = -\frac{9}{32} - j\frac{13}{32}$.

Section 6.1 presents the derivation of the algorithm for complex number on-line floating-point square root, including key characteristics such as on-line delay and the output digit selection function. Section 6.2 presents the modular design of a $RCNS_{2j,3}$ on-line floating-point square root unit. The design is mapped to a Virtex FPGA, and design parameters including cost and delay are determined. Section 6.3 compares the design with a conventional network of real number on-line arithmetic operators in terms of cost and delay. Section 6.4 summarizes the chapter and reviews the main contributions.

6.1 Derivation of the algorithm

Assuming a real number negative radix $-r^2$, and transforming the functions, complex number on-line square root can be derived from the bound

$$\begin{array}{l} |X_R[k] - (Z_R[k]Z_R[k] - Z_I[k]Z_I[k])| < |(-r^2)^{-k}| \\ |X_I[k] - 2Z_R[k]Z_I[k]| < |(-r^2)^{-k}| \end{array} \tag{6.2}$$

The residuals are defined as

$$\begin{array}{l} W_R[k] = (-r^2)^k(X_R[k] - (Z_R[k]Z_R[k] - Z_I[k]Z_I[k])) \\ W_I[k] = (-r^2)^k(X_I[k] - 2Z_R[k]Z_I[k]) \end{array} \tag{6.3}$$

The recurrences are

$$\begin{array}{rl} W_R[k] = & (-r^2)(W_R[k-1] - (Z_R[k-1] + Z_R[k-2])z_{R,k-1} \\ & +(Z_R[k-1] + Z_I[k-2])z_{I,k-1}) + (-r^2)^{-\delta+1}x_{R,k+\delta-1} \\ W_I[k] = & (-r^2)(W_I[k-1] - 2Z_I[k-1]z_{R,k-1} \\ & -2Z_R[k-1]z_{I,k-1}) + r^{-\delta+1}x_{I,k+\delta-1} \end{array} \tag{6.4}$$

with initial conditions

$$\begin{array}{l} W_R[0] = X_R[0] \\ W_I[0] = X_I[0] \end{array} \tag{6.5}$$

Defining

$$W[k] = W_R[k] + rjW_I[k] \tag{6.6}$$

the two recurrences can be combined as

$$\begin{array}{rl} W[k] = & -r^2((W_R[k-1] + jW_I[k-1]) \\ & -(2Z_R[k-2] + z_{R,k-1}(-r^2)^{-k})z_{R,k-1} + 2rjZ_I[k-2] + z_{I,k-1}(-r^2)^{-k}) \\ & -(2Z_I[k-1] + z_{I,k-2}(-r^2)^{-k} - 2rjZ_R[k-1] + z_R[k-2](-r^2)^{-k})z_{I,k-1} \\ & +(-r^2)^{-\delta+1}(x_{R,k+\delta-1} + rjx_{I,k+\delta-1}) \end{array} \tag{6.7}$$

Since

$$\begin{array}{l} Z[k-2] = Z_R[k-2] + rjZ_I[k-2] \\ rjZ[k-2] = -Z_I[k-2] + rjZ_R[k-2] \end{array} \tag{6.8}$$

The recurrence can be rewritten as

$$\begin{aligned} W[k] = \quad & (-r^2)(W[k-1] - (2Z[k-2] \\ & +(z_{R,k-1} + rjz_{I,k-1})(-r^2)^{-k+1})(z_{R,k-1} + rjz_{I,k-1})) \\ & +r^{-\delta+1}(x_{R,k+\delta-1} + rjx_{I,k+\delta-1}) \end{aligned} \tag{6.9}$$

Since $\sqrt{-r^2} = rj$, alternating the real and imaginary components yields recurrence equation

$$W[k] = (rj)(W[k-1] - (2Z[k-2] + z_{k-1}(rj)^{-k+1})z_{k-1}) + (rj)^{-\delta+1}(x_{k+\delta-1}) \tag{6.10}$$

where

$$\begin{aligned} x_{k+\delta-1} &= \begin{cases} x_{R,k+\delta-1} & \text{if } k+\delta-1 \text{ is even} \\ rjx_{I,k+\delta-1} & \text{if } k+\delta-1 \text{ is odd} \end{cases} \\ z_{k-1} &= \begin{cases} z_{R,k-1} & \text{if } k-1 \text{ is even} \\ rjz_{I,k-1} & \text{if } k-1 \text{ is odd} \end{cases} \end{aligned} \tag{6.11}$$

This can be decomposed into

$$W[k] = (rj)(W[k-1] + H_2[k]z_{k-1}) + H_1[k] \tag{6.12}$$

where

$$\begin{aligned} H_1[k] &= r^{-\delta+1}(x_{k+\delta-1}) \\ H_2[k] &= -(2Z[k-2] + z_{k-1}(rj)^{-k+1}) \end{aligned} \tag{6.13}$$

The limits can be determined as

$$\begin{aligned} U_d &\leq \omega - |H_2[k]|d = 2|Z[k-2]|d + \omega \\ L_d &\geq -\omega - |H_2[k]|d = 2|Z[k-2]|d - \omega \end{aligned} \tag{6.14}$$

where $|Z[k-2]|$ is the absolute value of $Z[k-2]$ defined as

$$|Z[k-2]| = \sqrt{(Z_R[k-2])^2 + (Z_I[k-2])^2} \tag{6.15}$$

Under the containment condition

$$\omega = K(2|Z[k-2]| - 2r^{-\delta+1}) \tag{6.16}$$

The minimal on-line delay is

$$\delta = 2\left(-\log_{r^2}\left(\frac{(2K-1)(2|Z[k-2]|-2^{-t})}{4K}\right)\right)+1 \tag{6.17}$$

where t is the number of fractional bits of the residual for the selection function. Using $RCNS_{2j,3}$ borrow-save representation, $r = 2$, $K = 1$, $t = 3$. Then the on-line delay, the error ω, and the limits U_d, L_d are

$$\begin{array}{l} \delta = 5 \\ \omega = 2|Z[k-2]| - \frac{1}{8} \\ U_d = 2|Z[k-2]|(d+1) - \frac{1}{8} \\ L_d = 2|Z[k-2]|(d-1) + \frac{1}{8} \end{array} \tag{6.18}$$

The $RCNS_{2j,3}$ on-line floating-point square root algorithm is shown below. The selection points m_d ($d = -3$ to 3) for the output digit selection function $z = SEL_{div}(\overline{W_E[k]}, \overline{2|Z[k-2]|})$ are shown in Table 6.1, where $\overline{W_E[k]}$ is the low-precision estimate of the even-indexed component of $W[k]$, and $\overline{2|Z[k-2]|})$ is the low-precision estimate of $2Z[k-2]$.

$RCNS_{2j,3}$ **On-line Floating-Point Square Root**

/* Initialization */

$e_z = \lceil \frac{e_x}{2} \rceil$

$W[-\delta + 1] = 0$

$x_0 = 0$

$Z[0] = 0$

$z_0 = 0$

for $k = -\delta + 2$ to 0 do

$$x'_{k+\delta-1} = \begin{cases} x_{k+\delta-2} & \text{if } e_x \text{ is odd} \\ x_{k+\delta-1} & \text{if } e_x \text{ is even} \end{cases}$$

$W[k] = (2j)W[k-1] + (2j)^{-4} x'_{k+\delta-1}$

end for

/* Recurrence */

for $k = 1$ to m do

$$x'_{k+\delta-1} = \begin{cases} x_{k+\delta-2} & \text{if } e_x \text{ is odd} \\ x_{k+\delta-1} & \text{if } e_x \text{ is even} \end{cases}$$

$W[k] = 2j(W[k-1] - (2Z[k-2] + z_{k-1}(2j)^{-k+1})z_{k-1}) + (2j)^{-\delta+1} x'_{k+\delta-1}$

$z_k = SEL_{sqrt}(\overline{W_E[k]}, \overline{2|Z[k-2]|})$

$Z_k = Z[k-1] + z_k(2j)^{-k}$

$e_z = NORM(e_z, z_k, \delta)$

end for

$\overline{2\|Z[k-2]\|}$	L_{-3}, U_{-3} m_{-3}	L_{-2}, U_{-2} m_{-2}	L_{-1}, U_{-1} m_{-1}	L_0, U_0 m_0	L_1, U_1 m_1	L_2, U_2 m_2	L_3, U_3 m_3
$\frac{4}{4}$	$-\frac{15}{4}, -\frac{9}{4}$ $-\frac{7}{2}$	$-\frac{11}{4}, -\frac{5}{4}$ $-\frac{5}{2}$	$-\frac{7}{4}, -\frac{1}{4}$ $-\frac{3}{2}$	$-\frac{3}{4}, \frac{3}{4}$ $-\frac{1}{2}$	$\frac{1}{4}, \frac{7}{4}$ $\frac{1}{2}$	$\frac{5}{4}, \frac{11}{4}$ $\frac{3}{2}$	$\frac{9}{4}, \frac{15}{4}$ $\frac{5}{2}$
$\frac{5}{4}$	$-\frac{19}{4}, -\frac{11}{4}$ -4	$-\frac{14}{4}, -\frac{6}{4}$ -3	$-\frac{9}{4}, -\frac{1}{4}$ -2	$-\frac{4}{4}, \frac{4}{4}$ -1	$\frac{1}{4}, \frac{9}{4}$ 1	$\frac{6}{4}, \frac{14}{4}$ 2	$\frac{11}{4}, \frac{19}{4}$ 3
$\frac{6}{4}$	$-\frac{23}{4}, -\frac{13}{4}$ -5	$-\frac{17}{4}, -\frac{7}{4}$ -4	$-\frac{11}{4}, -\frac{1}{4}$ -2	$-\frac{5}{4}, \frac{5}{4}$ -1	$\frac{1}{4}, \frac{11}{4}$ 1	$\frac{7}{4}, \frac{17}{4}$ 2	$\frac{13}{4}, \frac{23}{4}$ 4
$\frac{7}{4}$	$-\frac{27}{4}, -\frac{15}{4}$ -6	$-\frac{20}{4}, -\frac{8}{4}$ -5	$-\frac{13}{4}, -\frac{1}{4}$ -4	$-\frac{6}{4}, \frac{6}{4}$ -3	$\frac{1}{4}, \frac{13}{4}$ 1	$\frac{8}{4}, \frac{20}{4}$ 2	$\frac{15}{4}, \frac{27}{4}$ 4

Table 6.1: Selection points for $RCNS_{2j,3}$ on-line floating-point square root

6.2 FPGA implementation

A $RCNS_{2j,3}$ on-line floating-point square root unit can be designed as a series of modular slices, where each slice consists of a borrow-save digit multiplier, a 2:1 borrow-save digit adder, a digit-wide latch, a D flip-flop, a digit-wide register of D flip-flops, a TWICE unit, and a 2-to-1 MUX for appropriately appending digit z_{k-1} to vector $2Z[k-2]$. The output digit z_k is recoded into digit set $\{-2,\ldots,2\}$ using one DSREC unit. The most sigificant digits of the recurrence are determined using two MSREC units which handle potential most significant carry-out bits from the adders. The DIV2E unit divides the operand exponent by two to produce the exponent of the output, not considering exponent overflow/underflow. A 2-to-1 MUX determines whether digit $x_{k+\delta-1}$ or $x_{k+\delta}$ is used each cycle, depending on whether the operand exponent is odd or even. The selected digit is appended to the most signficant end of the vector product $(2Z[k-2]+z_{k-1}(2j)^{-k+1})z_{k-1}$. The SELSQRT unit selects the output digit z_k. The NORM unit normalizes the result by updating the exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point square unit is shown in Figure 6.1. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table 6.2. The total cost is $10m+3e+166$ CLB slices and the critical delay is $T_{CKO}+T_{ICK}+4T_{ILO}+5T_{NET}=(4.5+5T_{NET})$ns.

Module	Count	CLB slices
DIV2E	1	e
DSREC	1	12
2-to-1 MUX	$m+1$	$2m+2$
BSD mult. ($\otimes$)	m	$4m$
BSD adder (2:1)	$m-1$	$4m-4$
BSD adder (3:1)	1	8
MSREC	2	20
SELSQRT	1	104
NORM	1	$2e$
Total cost		$10m+3e+142$

Table 6.2: Cost of $RCNS_{2j,3}$ on-line floating-point square root unit

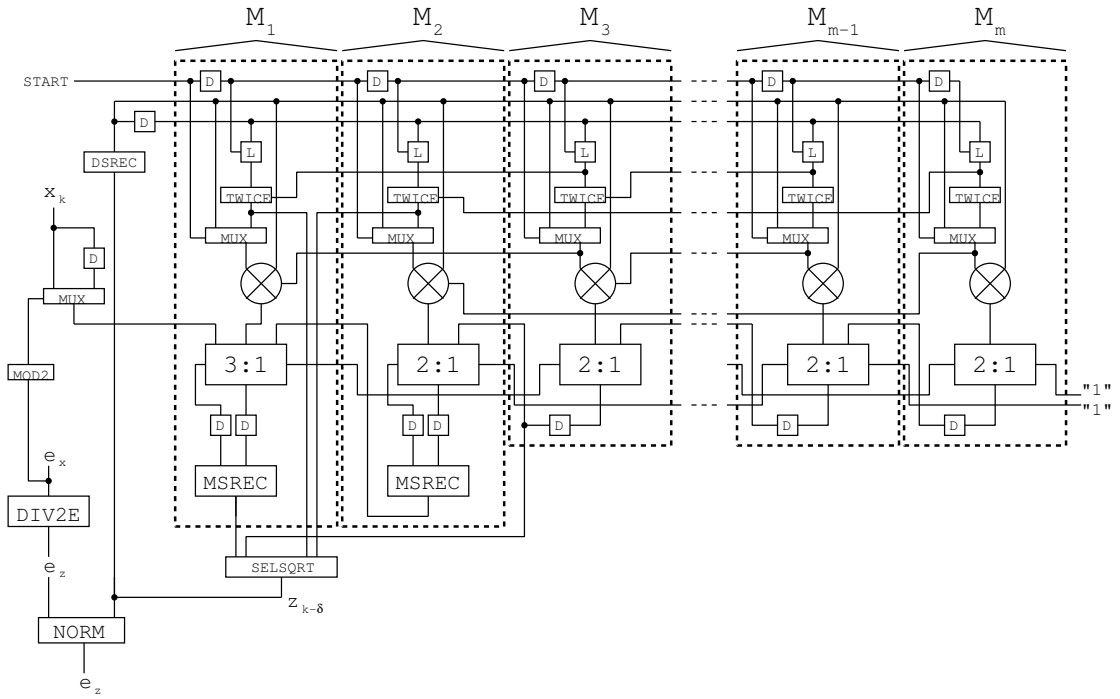


Figure 6.1: $RCNS_{2j,3}$ on-line floating-point square root unit

6.3 Comparison with real network

A complex number on-line floating-point square-root unit can be implemented according to the individual equations for the real and imaginary components as shown in Equation (6.1) consisting of three radix-2 on-line floating-point square-root units, three radix-2 on-line floating-point adders, and two radix-2 on-line floating-point square units.

A comparison of the $RCNS_{2j,3}$ approach and the real network approach for the implementation of a m-digit significand and e-bit exponent complex number on-line floating-point square root unit, is shown in Table 6.3.In both cases, the critical delay is $T_{CKO} + T_{ICK} + 3T_{ILO} + 4T_{NET} = (4.5 + 5T_{NET})$ns.

Approach	Cost (CLB slices)	δ
$RCNS_{2j,3}$	$10m + 3e + 142$	5
Radix-2 network	$\lceil 24.5m + 27e + 67 \rceil$	18

Table 6.3: Comparison of approaches

For $m = 24$ and $e = 8$, the $RCNS_{2j,3}$ approach has a cost of 406 CLB slices, wheareas the radix-2 network approach has a cost of 871 CLB slices.

6.4 Concluding Remarks

From the derivation of the algorithm and the implemenation for complex number on-line floating-point square root, we make the following general observations.

1. The result can be overnormalized or undernormalized, requiring special a normalization step. Overnormalized results are handled by incrementing the exponent. Undernormalized results are handled by decrementing the result and discarding output digits of value zero. For complex number on-line floating-point square root, since the result is in the range $\sqrt{\frac{1}{8}} \leq \max(|Z_R|, |Z_I|) < \sqrt{\frac{1+\sqrt{2}}{2}}$, at most one step for normalization is necessary to make the result quasi-normalized. Exponent overflow and underflow are not considered.

2. The $RCNS_{2j,3}$ approach for complex number on-line floating-point square root achieves approximately a 59% decrease in cost compared to the radix-2 network approach as the precision m increases, for fixed exponent precision.

3. The $RCNS_{2j,3}$ approach for general complex number on-line floating-point square root can lead to loss of precision for either the real or imaginary component of the output due to sharing the exponent, whereas in the radix-2 network approach the real and imaginary components of the output are normalized independently. The tradeoff is that there is no need for a digit cancallation step in the $RCNS_{2j,3}$ approach, whereas the radix-2 network approach may require digit cancellation due to internal floating-point real addition/subtraction operations, resulting in an on-line delay as large as the precision of the result.

CHAPTER 7

Complex Singular Value Decomposition

In this chapter we demonstrate the use of the $RCNS_{2j,3}$ on-line floating-point arithmetic operations for the design of an arithmetic unit that computes the singular value decomposition of an $n \times n$ matrix consisting of complex number elements. The singular value decomposition (SVD) of a real matrix $A \in R^{n \times n}$ is defined as the factorization:

$$A = U \Sigma V^T \tag{7.1}$$

where $U^T U = I$, $V^T V = I$, and Σ is a diagonal matrix with non-negative real diagonal elements. When the elements of A are complex numbers, i.e. $A \in C^{n \times n}$, this factorization is denoted as a complex singular value decomposition (CSVD), where U and V are unitary matrices, and U^T and V^T are the conjugate transpose of U and V, respectively. Since its introduction in [BG69], implementations for complex singular value decomposition have been investigated in [AFG91] [HC92] [HD96] [HLD00], and used in various applications, such as principal component analysis [EWS99], and wave analysis [SZY98]. The implementations in these papers are based on parallel arithmetic, and hence require a high latency in producing the CSVD of a matrix. We propose an approach using complex number on-line arithmetic which overlaps dependent operations and results in a smaller latency.

Section 7.1 presents the derivation of the algorithm for the CSVD, including key equations. Section 7.2 presents the design of a CSVD unit, utilizing the implementations of $RCNS_{2j,3}$ arithmetic operators from previous chapters. The design is mapped to a Virtex FPGA, and design parameters including cost and delay are determined. Section 7.3 compares the design with (i) a network of real number on-line arithmetic operators; and (ii) a network of real number parallel arithmetic operators in terms of cost and delay. Section 7.4 summarizes the chapter and reviews the main contributions.

7.1 Derivation of the algorithm

Denoting the $n \times n$ matrix as Ms, the CSVD can be computed using a mesh-connected processor array, consisting of $n/2 \times n/2$ processors, where each processor operates on a 2×2 submatrix $M_{p,q}$ $(p, q \in 1, \ldots, n/2)$ of M. This is known as the Brent, Luk, Van Loan SVD algorithm [BLV85], and is shown in Figure 7.1, where the thick-outlined 2×2 submatrices are denoted *on-diagonal* submatrices, and the other 2×2 submatrices are denoted *off-diagonal* submatrices.

The operation on each on-diagonal submatrix $M_{p,p}$ consists of two steps: (i) triangularization; and (ii) diagonalization. These steps consist of determining left and right rotation matrices $L_{p,p}$ and $R_{p,p}$ for triangularization, and $L'_{p,p}$ and $R'_{p,p}$ for diagonalization, and performing matrix multiplications $M'_{p,p} = L_{p,p} M_{p,p} R_{p,p}$ and $M'' = L'_{p,p} M'_{p,p} R'_{p,p}$. The elements of the left rotation matrices $L_{p,p}$ and $L'_{p,p}$ are broadcast in step to off-diagonal submatrices $M_{p,q}$ in the corresponding rows,

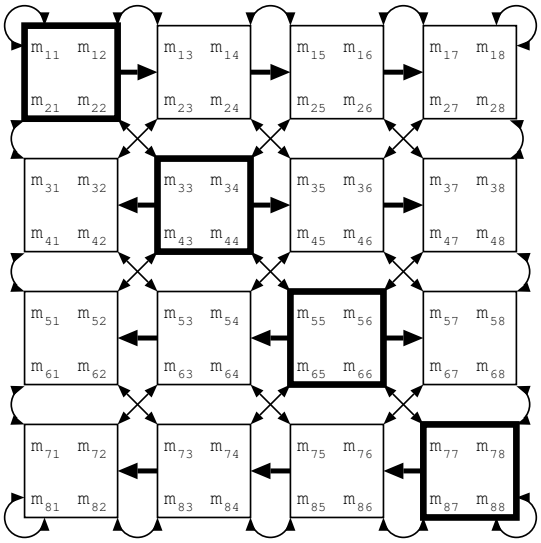


Figure 7.1: CSVD processor array

and the elements of the right rotation matrices $R_{q,q}$ and $R'_{q,q}$ are broadcast in step to off-diagonal submatrices $M_{p,q}$ in the corresponding columns. Then for each off-diagonal submatrix $M_{p,q}$, the matrix multiplications $M'_{p,q} = L_{p,p}M_{p,q}R_{q,q}$ and $M''_{p,q} = L'_{p,p}M'_{p,q}R'_{q,q}$ are performed.

After the two-step diagonalization is performed upon each diagonal submatrix, rows and columns of the matrix are exchanged between adjacent submatrices according to the parallel ordering of [BL85]. The sequence of triangularization, diagonalization, and element exchange is referred to as a *sweep*. Successive sweeps are performed until the only remaining non-zero elements of the matrix lie on the main diagonal. Each step will be described in detail next.

7.1.1 Triangularization

The general triangularization step takes as input an on-diagonal submatrix $M_{p,p}$ represented as

$$M_{p,p} = \begin{bmatrix} m_{2p-1,2p-1} & m_{2p-1,2p} \\ m_{2p,2p-1} & m_{2p,2p} \end{bmatrix} \tag{7.2}$$

and produces a complex triangular matrix $M'_{p,p}$ consisting of complex elements $m'_{2p-1,2p-1}$, $m'_{2p-1,2p}$, and real element $m'_{2p,2p}$ represented as

$$M'_{p,p} = \begin{bmatrix} m'_{2p-1,2p-1} & m'_{2p-1,2p} \\ 0 & m'_{2p,2p} \end{bmatrix} \tag{7.3}$$

For simplification of notation, setting

$$\begin{aligned} a &= m_{2p-1,2p-1} \\ b &= m_{2p-1,2p} \\ c &= m_{2p,2p-1} \\ d &= m_{2p,2p} \\ w &= m'_{2p-1,2p-1} \\ x &= m'_{2p-1,2p} \\ z &= m'_{2p,2p} \end{aligned} \tag{7.4}$$

then the transformation can be written as

$$\begin{bmatrix} e^{j\theta_\alpha}\cos\theta_\phi & -e^{j\theta_\beta}\sin\theta_\phi \\ e^{j\theta_\alpha}\sin\theta_\phi & e^{j\theta_\beta}\cos\theta_\phi \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e^{j\theta_\gamma}\cos\theta_\psi & e^{j\theta_\gamma}\sin\theta_\psi \\ -e^{j\theta_\delta}\sin\theta_\psi & e^{j\theta_\delta}\cos\theta_\psi \end{bmatrix} = \begin{bmatrix} w & x \\ 0 & z \end{bmatrix} \tag{7.5}$$

where

$$\begin{aligned} C &= \sqrt{c_R^2 + c_I^2} \\ D &= \sqrt{d_R^2 + d_I^2} \\ \theta_c &= \tan^{-1}(c_I/c_R) \\ \theta_d &= \tan^{-1}(d_I/d_R) \\ \theta_\alpha &= \theta_\beta = -(\theta_c + \theta_d)/2 \\ \theta_\gamma &= (\theta_d - \theta_c)/2 \\ \theta_\delta &= (\theta_c - \theta_d)/2 \\ \theta_\phi &= 0 \\ \theta_\psi &= \tan^{-1}(C/D) \end{aligned} \tag{7.6}$$

Defining the conjugates

$$\begin{aligned} c^* &= c_R - jc_I \\ d^* &= d_R - jd_I \end{aligned} \tag{7.7}$$

then

$$\begin{aligned} cc^* &= c_R^2 + c_I^2 \\ dd^* &= d_R^2 + d_I^2 \\ e^{j\theta_\alpha} &= e^{j\theta_\beta} = \sqrt{\sqrt{\frac{c^*d^*}{cd}}} \\ e^{j\theta_\gamma} &= \sqrt{\sqrt{\frac{c^*d}{cd^*}}} \\ e^{j\theta_\delta} &= \sqrt{\sqrt{\frac{cd^*}{c^*d}}} \\ \cos\theta_\phi &= 1 \\ \sin\theta_\phi &= 0 \\ \cos\theta_\psi &= \sqrt{\frac{dd^*}{cc^*+dd^*}} \\ \sin\theta_\psi &= \sqrt{\frac{cc^*}{cc^*+dd^*}} \end{aligned} \tag{7.8}$$

Then the rotation angles can be computed as

$$\begin{aligned} L_{2p-1,2p-1} &= e^{j\theta_\alpha} \\ L_{2p-1,2p} &= 0 \\ L_{2p,2p-1} &= 0 \\ L_{2p,2p} &= e^{j\theta_\beta} \\ R_{2p-1,2p-1} &= e^{j\theta_\gamma}\cos\theta_\psi \\ R_{2p-1,2p} &= e^{j\theta_\gamma}\sin\theta_\psi \\ R_{2p,2p-1} &= -e^{j\theta_\delta}\sin\theta_\psi \\ R_{2p,2p} &= e^{j\theta_\delta}\cos\theta_\psi \end{aligned} \tag{7.9}$$

Applying the rotation angles to the on-diagonal submatrix, since the left and right rotation angles are based on the same submatrix, then

$$\begin{aligned} (e^{j\theta_\alpha})(e^{j\theta_\gamma}) &= e^{j\theta_c} = \frac{c^*}{\sqrt{cc^*}} \\ (e^{j\theta_\alpha})(e^{j\theta_\delta}) &= e^{j\theta_d} = \frac{d^*}{\sqrt{dd^*}} \end{aligned} \tag{7.10}$$

Then the equations for w, x, and z become

$$\begin{aligned} w &= a(\tfrac{c^*}{\sqrt{cc^*}})(\sqrt{\tfrac{dd^*}{cc^*+dd^*}}) - b(\tfrac{d^*}{\sqrt{dd^*}})(\sqrt{\tfrac{cc^*}{cc^*+dd^*}}) \\ x &= a(\tfrac{c^*}{\sqrt{cc^*}})(\sqrt{\tfrac{cc^*}{cc^*+dd^*}}) + b(\tfrac{d^*}{\sqrt{dd^*}})(\sqrt{\tfrac{dd^*}{cc^*+dd^*}}) \\ z &= \sqrt{cc^* + dd^*} \end{aligned} \tag{7.11}$$

For an off-diagonal submatrix $M_{p,q}$ defined as

$$M_{p,q} = \begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \tag{7.12}$$

the left rotation angles $L_{2p-1,2p-1}$ and $L_{2p,2p}$, and the right rotation angles $R_{2q-1,2q-1}$, $R_{2q-1,2q}$, $R_{2q,2q-1}$, and $R_{2q,2q}$ are applied to produce submatrix $M'_{p,q}$ represented as

$$M'_{p,q} = \begin{bmatrix} m'_{2p-1,2q-1} & m'_{2p-1,2q} \\ m'_{2p,2q-1} & m'_{2p,2q} \end{bmatrix} \tag{7.13}$$

For simplification of notation, setting

$$\begin{aligned} a &= m_{2p-1,2q-1} \\ b &= m_{2p-1,2q} \\ c &= m_{2p,2q-1} \\ d &= m_{2p,2q} \\ w &= m'_{2p-1,2q-1} \\ x &= m'_{2p-1,2q} \\ y &= m'_{2p,2q-1} \\ z &= m'_{2p,2q} \\ L &= L_{2p-1,2p-1} = L_{2p,2p} \\ R^{11} &= R_{2q-1,2q-1} \\ R^{12} &= R_{2q-1,2q} \\ R^{21} &= R_{2q,2q-1} \\ R^{22} &= R_{2q,2q} \end{aligned} \tag{7.14}$$

then

$$\begin{aligned} w &= L(aR^{11} + bR^{21}) \\ x &= L(aR^{12} + bR^{22}) \\ y &= L(cR^{11} + dR^{21}) \\ z &= L(dR^{12} + dR^{22}) \end{aligned} \tag{7.15}$$

The operator utilization for various steps, including angle calulation and rotation for an on-diagonal submatrix and rotation of an off-diagonal submatrix are shown in Table 7.1. The computational graph for on-diagonal angle calculation and rotation is shown in Figure 7.2. The computational graph for off-diagonal rotation is shown in Figure 7.3.

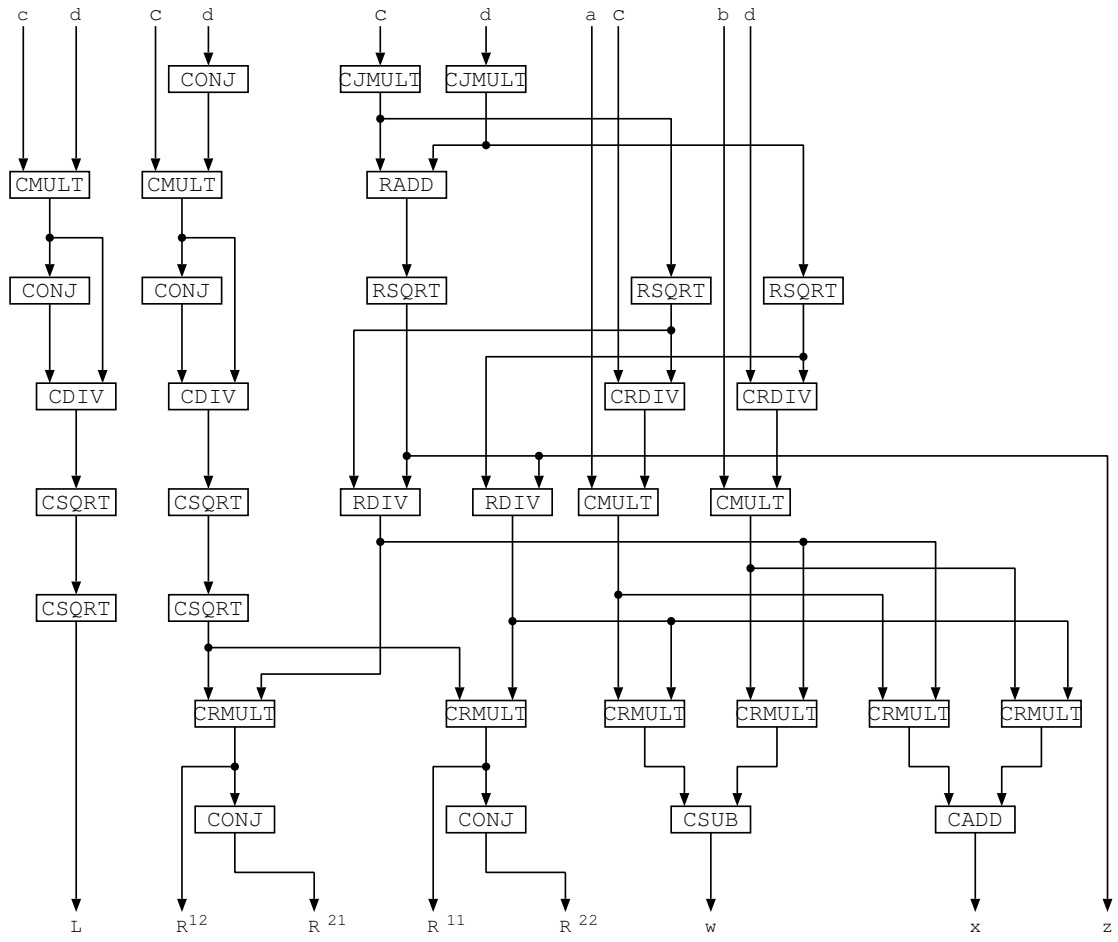


Figure 7.2: Computational graph for on-diagonal angle calculation and rotation

Operator	Left operand	Right operand	Mnemonic	On diag.	Off diag.
$+/-$	real	real	RADD/RSUB	1	—
	complex	complex	CADD/CSUB	2	4
$\times$	complex	complex	CMULT	4	12
	complex	real	CRMULT	6	—
	complex	conjugate	CJMULT	2	—
$\div$	real	real	RDIV	2	—
	complex	complex	CDIV	2	—
	complex	real	CRDIV	2	—
$\surd$	real	—	RSQRT	3	—
	complex	—	CSQRT	4	—
conjugate	complex	—	CONJ	7	—

Table 7.1: Triangularization operator utilization

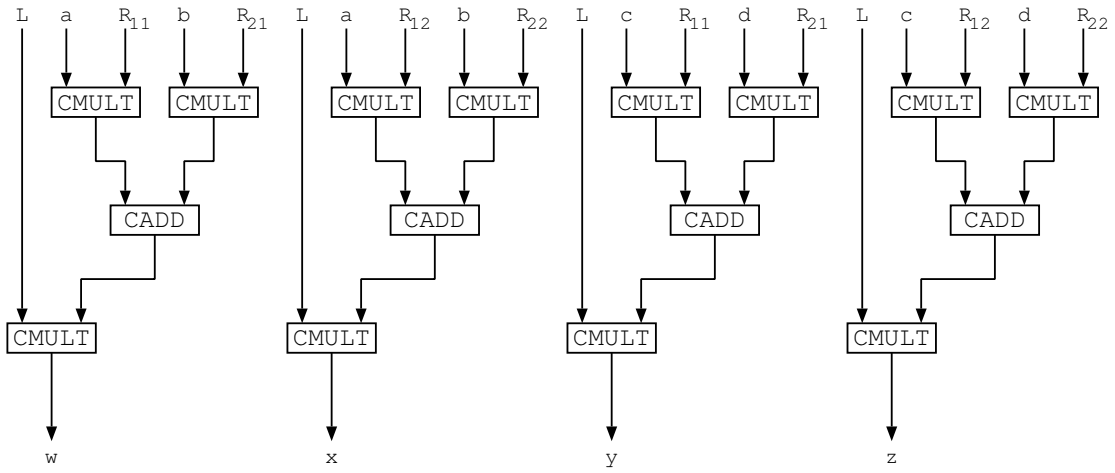


Figure 7.3: Computational graph for off-diagonal rotation

7.1.2 Diagonalization

The general diagonalization step takes as input the triangular on-diagonal submatrix $M'_{p,p}$ generated from the triangularization step, consisting of complex elements $m'_{2p-1,2p-1}$, $m'_{2p-1,2p}$, and real element $m'_{2p,2p}$, represented as

$$M'_{p,p} = \begin{bmatrix} m'_{2p-1,2p-1} & m'_{2p-1,2p} \\ 0 & m'_{2p,2p} \end{bmatrix} \tag{7.16}$$

and produces a real diagonal matrix $M''_{p,p}$ consisting of real elements $m''_{2p-1,2p-1}$ and $m''_{2p,2p}$ represented as

$$M''_{p,p} = \begin{bmatrix} m''_{2p-1,2p-1} & 0 \\ 0 & m''_{2p,2p} \end{bmatrix} \tag{7.17}$$

For simplification of notation, we set $w = m'_{2p-1,2p-1}$, $x = m'_{2p-1,2p}$, $z = m'_{2p,2p}$, $p = m''_{2p-1,2p-1}$, and $s = m''_{2p,2p}$.

Then the transformation can be written as

$$\begin{bmatrix} e^{j\theta_\varepsilon}\cos\theta_\lambda & -e^{j\theta_\eta}\sin\theta_\lambda \\ e^{j\theta_\varepsilon}\sin\theta_\lambda & e^{j\theta_\eta}\cos\theta_\lambda \end{bmatrix} \begin{bmatrix} w & x \\ 0 & z \end{bmatrix} \begin{bmatrix} e^{j\theta_\varsigma}\cos\theta_\rho & e^{j\theta_\varsigma}\sin\theta_\rho \\ -e^{j\theta_\omega}\sin\theta_\rho & e^{j\theta_\omega}\cos\theta_\rho \end{bmatrix} = \begin{bmatrix} p & 0 \\ 0 & s \end{bmatrix} \tag{7.18}$$

where

$$\begin{aligned} W &= \sqrt{w_R^2 + w_I^2} \\ X &= \sqrt{x_R^2 + x_I^2} \\ \theta_w &= \tan^{-1}(w_I/w_R) \\ \theta_x &= \tan^{-1}(x_I/x_R) \\ \theta_\varepsilon &= -(\theta_x - \theta_w)/2 \\ \theta_\eta &= \theta_\varsigma = (\theta_x - \theta_w)/2 \\ \theta_\omega &= (\theta_w - \theta_x)/2 \\ \alpha &= (z+W)\sqrt{X^2 + (z-W)^2} \\ \beta &= (z-W)\sqrt{X^2 + (z+W)^2} \\ \gamma &= X(\sqrt{X^2 + (z+W)^2} - \sqrt{X^2 + (z-W)^2}) \\ \theta_\lambda &= (\alpha - \beta)/\gamma \\ \theta_\rho &= \gamma/(\alpha + \beta) \end{aligned} \tag{7.19}$$

Defining the conjugates

$$\begin{aligned} w^* &= w_R - jw_I \\ x^* &= x_R - jx_I \end{aligned} \tag{7.20}$$

then

$$\begin{aligned} ww^* &= w_R^2 + w_I^2 \\ xx^* &= x_R^2 + x_I^2 \\ e^{j\theta_\varepsilon} &= \sqrt{\sqrt{\frac{w^*x^*}{wx}}} \\ e^{j\theta_\eta} &= e^{j\varsigma} = \sqrt{\sqrt{\frac{w^*x}{wx^*}}} \\ e^{j\theta_\omega} &= \sqrt{\sqrt{\frac{wx^*}{w^*x}}} \\ \cos\theta_\lambda &= \frac{\gamma}{\sqrt{\gamma^2+(\alpha-\beta)^2}} \\ \sin\theta_\lambda &= \frac{\alpha-\beta}{\sqrt{\gamma^2+(\alpha-\beta)^2}} \\ \cos\theta_\rho &= \frac{\alpha+\beta}{\sqrt{\gamma^2+(\alpha+\beta)^2}} \\ \sin\theta_\rho &= \frac{\gamma}{\sqrt{\gamma^2+(\alpha+\beta)^2}} \end{aligned} \tag{7.21}$$

Then the rotation angles can be computed as

$$\begin{aligned} L'_{2p-1,2p-1} &= e^{j\theta_\varepsilon}\cos\theta_\lambda \\ L'_{2p-1,2p} &= -e^{j\theta_\eta}\sin\theta_\lambda \\ L'_{2p,2p-1} &= e^{j\theta_\varepsilon}\sin\theta_\lambda \\ L'_{2p,2p} &= e^{j\theta_\eta}\cos\theta_\lambda \\ R'_{2p-1,2p-1} &= e^{j\theta_\varsigma}\cos\theta_\rho \\ R'_{2p-1,2p} &= e^{j\theta_\varsigma}\sin\theta_\rho \\ R'_{2p,2p-1} &= -e^{j\theta_\omega}\sin\theta_\rho \\ R'_{2p,2p} &= e^{j\theta_\omega}\cos\theta_\rho \end{aligned} \tag{7.22}$$

Applying the rotation angles to the on-diagonal submatrix, since the left and right rotation angles are based on the same 2×2 submatrix, then

$$\begin{aligned} (e^{j\theta_\varepsilon})(e^{j\theta_\varsigma}) &= e^{-j\theta_w} = \frac{w^*}{\sqrt{ww^*}} \\ (e^{j\theta_\varepsilon})(e^{j\theta_\omega}) &= e^{-j\theta_x} = \frac{x^*}{\sqrt{xx^*}} \\ (e^{j\theta_\eta})(e^{j\theta_\omega}) &= 0 \end{aligned} \tag{7.23}$$

Then the equations for p and s become

$$\begin{aligned} p &= (\sqrt{ww^*})\cos\theta_\lambda\cos\theta_\rho - (\sqrt{xx^*})\cos\theta_\lambda\sin\theta_\rho + z\sin\theta_\lambda\sin\theta_\rho \\ s &= (\sqrt{ww^*})\sin\theta_\lambda\sin\theta_\rho + (\sqrt{xx^*})\sin\theta_\lambda\cos\theta_\rho + z\cos\theta_\lambda\cos\theta_\rho \end{aligned} \tag{7.24}$$

For an off-diagonal submatrix $M'_{p,q}$ defined as

$$M'_{p,q} = \begin{bmatrix} m'_{2p-1,2q-1} & m'_{2p-1,2q} \\ m'_{2p,2q-1} & m'_{2p,2q} \end{bmatrix} \tag{7.25}$$

the left rotation angles $L'_{2p-1,2p-1}$, $L'_{2p-1,2p}$, $L'_{2p,2p-1}$, and $L'_{2p,2p}$, and the right rotation angles $R'_{2q-1,2q-1}$, $R'_{2q-1,2q}$, $R'_{2q,2q-1}$, and $R'_{2q,2q}$ are applied to produce submatrix $M''_{p,q}$ represented as

$$M''_{p,q} = \begin{bmatrix} m''_{2p-1,2q-1} & m''_{2p-1,2q} \\ m''_{2p,2q-1} & m''_{2p,2q} \end{bmatrix} \tag{7.26}$$

For simplification of notation, setting

$$\begin{aligned}
w &= m'_{2p-1,2q-1} \\
x &= m'_{2p-1,2q} \\
y &= m'_{2p,2q-1} \\
z &= m'_{2p,2q} \\
p &= m''_{2p-1,2q-1} \\
q &= m''_{2p-1,2q} \\
r &= m''_{2p,2q-1} \\
s &= m''_{2p,2q} \\
L'^{11} &= L'_{2p-1,2p-1} \\
L'^{12} &= L'_{2p-1,2p} \\
L'^{21} &= L'_{2p,2p-1} \\
L'^{22} &= L'_{2p,2p} \\
R'^{11} &= R'_{2q-1,2q-1} \\
R'^{12} &= R'_{2q-1,2q} \\
R'^{21} &= R'_{2q,2q-1} \\
R'^{22} &= R'_{2q,2q}
\end{aligned} \tag{7.27}$$

then

$$\begin{aligned}
p &= (L'^{11}w + L'^{12}y)R'^{11} + (L'^{11}x + L'^{12}z)R'^{21} \\
q &= (L'^{11}w + L'^{12}y)R'^{12} + (L'^{11}x + L'^{12}z)R'^{22} \\
r &= (L'^{21}w + L'^{22}y)R'^{11} + (L'^{21}x + L'^{22}z)R'^{21} \\
s &= (L'^{21}w + L'^{22}y)R'^{11} + (L'^{21}x + L'^{22}z)R'^{22}
\end{aligned} \tag{7.28}$$

The operator utilization for various steps, including angle calulation and rotation for an on-diagonal submatrix and rotation of an off-diagonal submatrix are shown in Table 7.2. The computational graph for on-diagonal angle calculation and rotation is shown in Figure 7.4. The computational graph for off-diagonal rotation is shown in Figure 7.5.

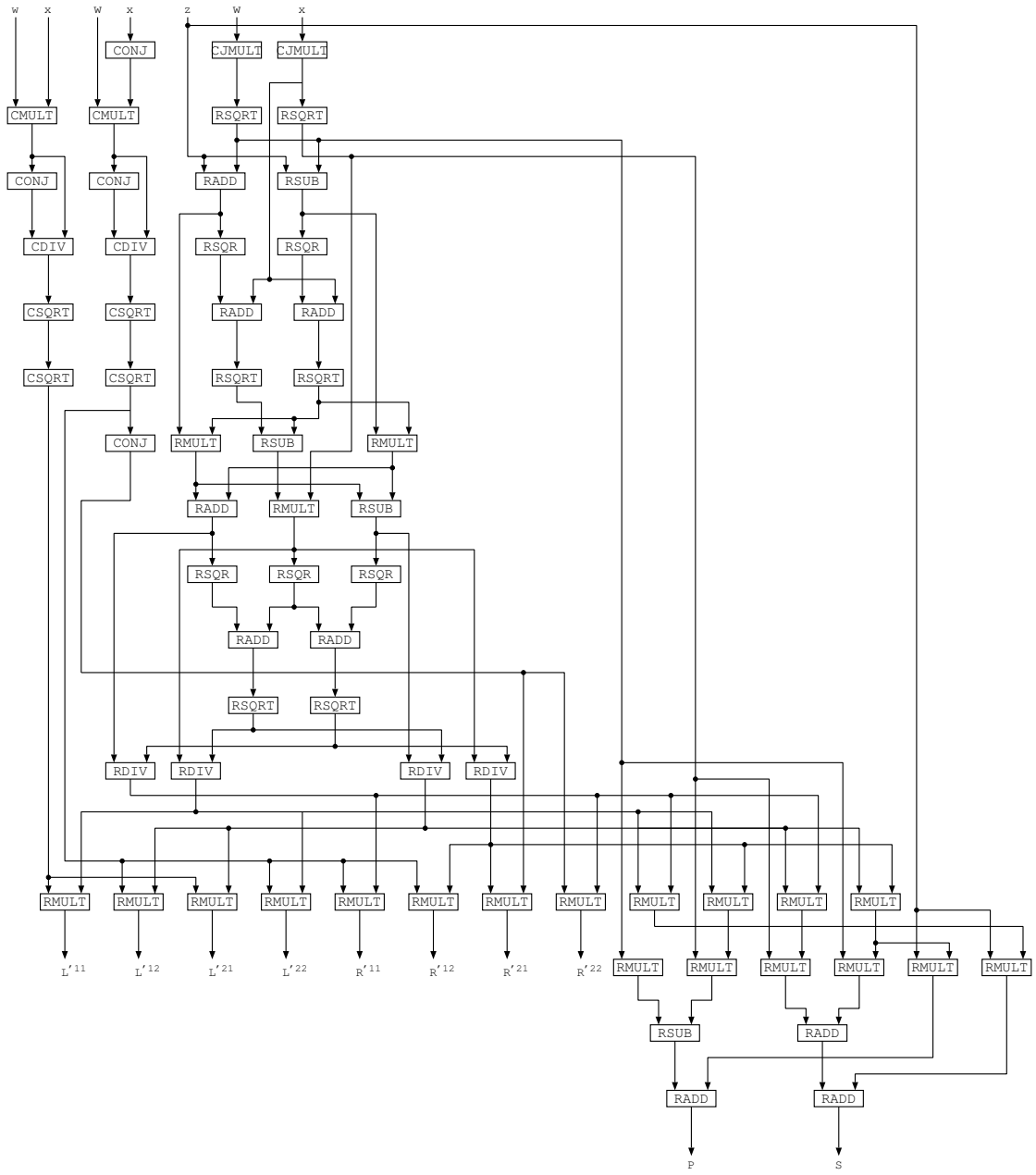


Figure 7.4: Computational graph for on-diagonal angle calculation and rotation

Operator	Left operand	Right operand	Mnemonic	On diag.	Off diag.
$+/-$	real	real	RADD/RSUB	14	—
	complex	complex	CADD/CSUB	—	12
$\times$	real	real	RMULT	13	—
	complex	complex	CMULT	2	24
	complex	real	CRMULT	8	—
	complex	conjugate	CJMULT	2	—
square	real	—	RSQR	5	—
$\div$	real	real	RDIV	4	—
	complex	complex	CDIV	2	—
$\surd$	real	—	RSQRT	6	—
	complex	—	CSQRT	4	—
conjugate	complex	—	CONJ	4	—

Table 7.2: Diagonalization operator utilization

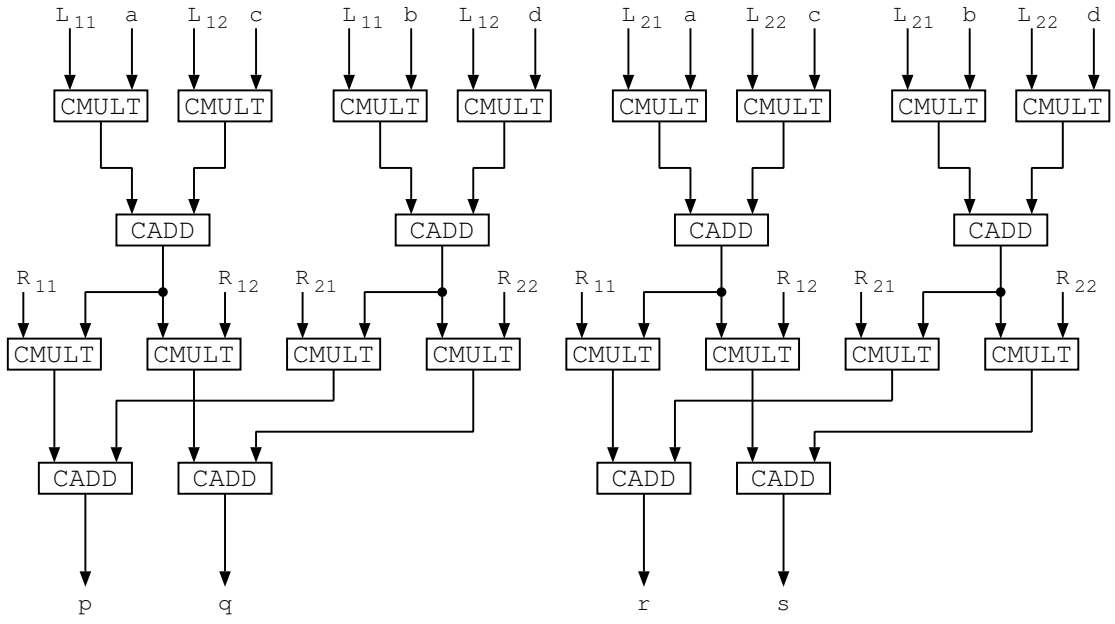


Figure 7.5: Computational graph for off-diagonal rotation

7.1.3 Element exchange

The element exchange step reorders the elements $m_{p,q}$ of an $n \times n$ matrix M such that elements contained within off-diagonal submatrices $M_{p,q}$ are shifted in step to on-diagonal submatrices $M_{p,p}$. Successive triangularization and diagonalization steps are performed to "zero-off" the off-diagonal elements until the only non-zero elements lie on the main diagonal. The element exchange step consists of: (i) *intra-matrix exchange*; and (ii) *inter-matrix exchange*.

7.1.3.1 Intra-matrix exchange

Intra-matrix exchange consists of rearranging the elements within a 2×2 submatrix $M_{p,q}$ to allow successive columns/rows to be exchanged when inter-matrix exchange is performed. Assuming a general 2×2 submatrix $M_{p,q}$ defined as

$$M_{p,q} = \begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \tag{7.29}$$

the algorithm for intra-matrix exchange is shown below:

Intra-matrix exchange

if $p = 1$ and $q = 1$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix}$$

else if $q = 1$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-1,2q} & m_{2p-1,2q-1} \\ m_{2p,2q} & m_{2p,2q-1} \end{bmatrix}$$

else if $p = 1$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p,2q-1} & m_{2p,2q} \\ m_{2p-1,2q-1} & m_{2p-1,2q} \end{bmatrix}$$

else

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p,2q} & m_{2p,2q-1} \\ m_{2p-1,2q} & m_{2p-1,2q-1} \end{bmatrix}$$

7.1.3.2 Inter-matrix exchange

Inter-matrix exchange consists of exchanging elements between different submatrices in order to bring off-diagonal elements in step into on-diagonal submatrices. Assuming a general 2×2 submatrix $M_{p,q}$ defined as

$$M_{p,q} = \begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \tag{7.30}$$

the algorithm for inter-matrix exchange is shown below:

Inter-matrix exchange

if $p = 1$ and $q = 1$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q+1} \\ m_{2p+1,2q-1} & m_{2p+1,2q+1} \end{bmatrix}$$

else if $p < \frac{n}{2}$ and $q = 1$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p,2q-1} & m_{2p,2q+1} \\ m_{2p+1,2q-1} & m_{2p+1,2q+1} \end{bmatrix}$$

else if $p = \frac{n}{2}$ and $q = 1$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-2,2q-1} & m_{2p-2,2q+1} \\ m_{2p,2q-1} & m_{2p,2q+1} \end{bmatrix}$$

else if $p = 1$ and $q < \frac{n}{2}$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-1,2q-2} & m_{2p-1,2q+1} \\ m_{2p+1,2q-2} & m_{2p+1,2q+1} \end{bmatrix}$$

else if $p = 1$ and $q = \frac{n}{2}$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-1,2q-2} & m_{2p-1,2q} \\ m_{2p+1,2q-2} & m_{2p+1,2q} \end{bmatrix}$$

else if $p = \frac{n}{2}$ and $q = \frac{n}{2}$ then

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-2,2q-2} & m_{2p-2,2q} \\ m_{2p,2q-2} & m_{2p,2q} \end{bmatrix}$$

else

$$\begin{bmatrix} m_{2p-1,2q-1} & m_{2p-1,2q} \\ m_{2p,2q-1} & m_{2p,2q} \end{bmatrix} \rightarrow \begin{bmatrix} m_{2p-2,2q-2} & m_{2p-2,2q+1} \\ m_{2p+1,2q-2} & m_{2p+1,2q+1} \end{bmatrix}$$

An example of computing the CSVD of an 8×8 matrix, consisting of 7 sweeps, is shown in Figure 7.6.

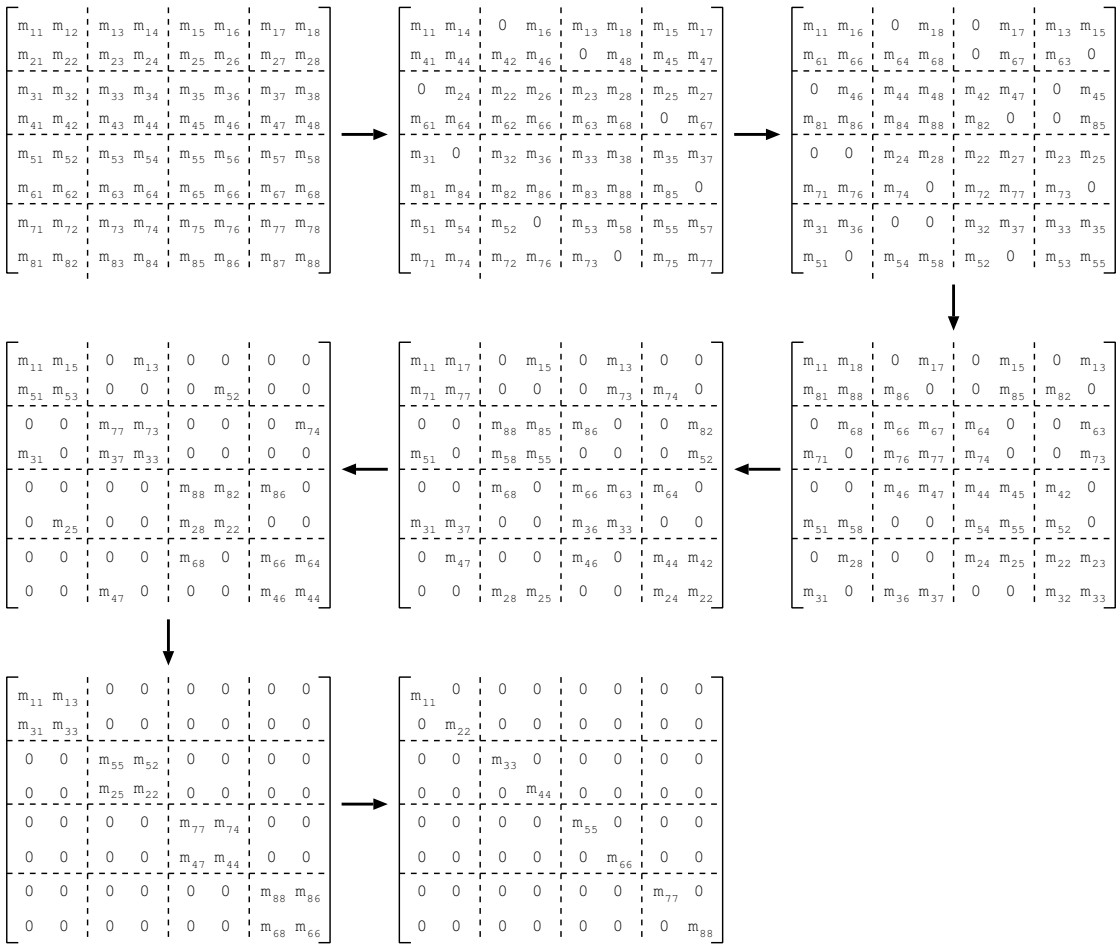


Figure 7.6: Computation of the CSVD of a 8×8 matrix

7.2 FPGA implementation

A CSVD arithmetic unit can be designed as a network of $RCNS_{2j,3}$ on-line floating-point operators, with appropriate normalization, and appropriate synchronization of signals. Since synchronization is handled by inserting D flip-flops, which do not affect the cost since the number of look up tables (LUT's) is larger, synchronization is not included in the total cost. The invidual cost per module type, the total number of modules of each type used, the total cost, and the on-line delay δ, which takes into consideration normalization, are shown in Table 7.3, assuming 24-digit significands and 8-bit exponents. The same mnemonics as in Table 7.1 for operations are used. As a note, arithmetic operations which consider one complex number and one real number operand (such as CRMULT), or operations that only consider real number operands (such as RDIV) are designed as $RCNS_{2j,3}$ on-line floating point arithmetic operations with imaginary input digits preset to zero. This has the effect of optimization of the operation at the synthesis level, as unused modules (such as a BSD multiplier where it is known one of the digits is set to zero) are removed.

The total cost for an on-diagonal submatrix is $C_{on} = 10426 + 15822 = 26248$ CLB slices. The total cost for an off-diagonal submatrix is $C_{off} = 5856 + 12144 = 18000$ CLB slices.

The total on-line delay per sweep is computed as the sum of on-line delays of individual modules along the critical path. Since updating the off-diagonal

Operator	CLB slices	Triangularization				Diagonalization				δ
		On diag.		Off diag.		On diag.		Off diag.		
		#	Cost	#	Cost	#	Cost	#	Cost	
RADD/RSUB	108	1	108	—	—	14	1512	—	—	4
CADD/CSUB	108	2	216	4	432	—	—	12	1296	4
RMULT	260	2	520	—	—	13	3380	—	—	6
CMULT	452	4	1808	12	5424	2	904	24	10848	6
CRMULT	356	6	2136	—	—	8	2848	—	—	6
CJMULT	224	2	448	—	—	2	448	—	—	6
RSQR	176	—	—	—	—	5	880	—	—	6
RDIV	353	2	706	—	—	4	1412	—	—	6
CDIV	545	2	1090	—	—	2	1090	—	—	6
CRDIV	449	2	898	—	—	—	—	—	—	6
RSQRT	286	3	858	—	—	6	1716	—	—	6
CSQRT	406	4	1624	—	—	4	1624	—	—	6
CONJ	2	7	14	—	—	4	8	—	—	1
Total cost			10426		5856		15822		12144	

Table 7.3: Cost of $RCNS_{2j,3}$ on-line floating point CSVD ($m = 24, e = 8$)

submatrix elements takes longer than for the on-diagonal submatrix elements, once the most significant digit of each element of the off-diagonal element is updated, by applying the diagonalization rotation angles, element exchange is performed at the digit level, and another sweep starts. The cycle dependancy graph for an on-diagonal submatrix as well as an off-diagonal submatrix using the $RCNS_{2j,3}$ on-line floating-point arithmetic modules is shown in Figure 7.7. The total on-line delay for a sweep is 121 cycles.

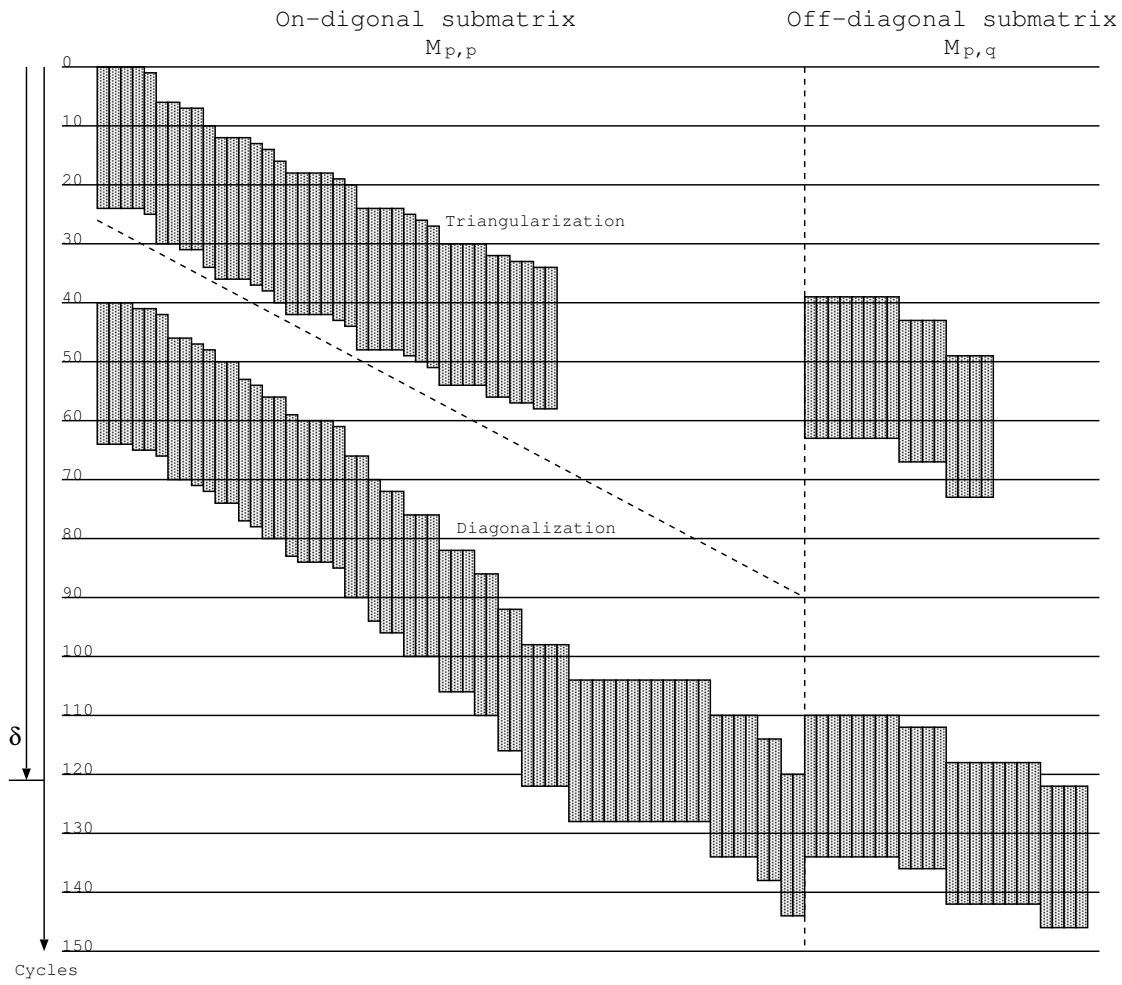


Figure 7.7: Cycle dependancy graph for on-line CSVD

7.3 Real network approaches

Alternative network approaches for the design of an $n \times n$ CSVD arithmetic unit include: (i) a real network of radix 2 on-line floating-point arithmetic operators; and (ii) a real network of Xilinx CORE parallel operators. Comparisons in terms of overall cost measured in CLB slices and total latency measured in cycles are given.

7.3.1 Radix-2 on-line network

A CSVD arithmetic unit can be alternatively designed as a network of radix-2 on-line floating-point operators, with appropriate normalization, and appropriate synchronization of signals. Each of the complex number arithmetic modules shown in Figures 7.2, 7.3, 7.4, and 7.5 is reduced to a network of radix-2 on-line floating-point operators, which were derived in Appendix A, and who whose parameters are summarized in Table 7.4 for general m-digit significand and e-bit exponent precision. As a note, design and implementation of basic radix-2 floating-point on-line arithmetic operators mapped to LSI logic gates was realized in [Tu90]. However, since the target technology in this dissertation is reconfigurable hardware, and due to the difficulty to accurately determine equivalent CLB slice cost given the gate count of a design (since 2 to 12 gates can map to a CLB slice, based on the number of outputs), a comparison with the results in [Tu90] cannot accurately be made.

Operator	CLB slices	δ
$+, -$	$\lceil 1.5m + 4e + 2 \rceil$	3
$\times$	$6m + 3e + 2$	4
square	$4m + 3e + 2$	4
$\div$	$6m + 3e + 9$	5
$\sqrt{}$	$4m + 3e + 15$	4

Table 7.4: Parameters of radix-2 on-line floating point arithmetic operators

The individual cost per module type, the total number of modules of each type used, and the on-line delay δ, which takes into consideration normalization, assuming no cancellation of leading digits for addition/subtraction are shown in Table 7.5, assuming 24-digit singificands and 8-bit exponents.

Operator	CLB slices	Triangularization				Diagonalization				δ
		On diag.		Off diag.		On diag.		Off diag.		
		#	Cost	#	Cost	#	Cost	#	Cost	
$+/-$	70	25	1750	32	2240	25	1750	48	3360	4
$\times$	170	25	4250	48	8160	30	5100	64	10880	5
square	122	8	976	—	—	13	1586	—	—	5
$\div$	177	10	1770	—	—	8	1416	—	—	6
$\sqrt{}$	135	9	1215	—	—	12	1620	—	—	5
Total cost			9961		10400		11472		14240	

Table 7.5: Cost of radix-2 on-line floating point CSVD ($m = 24, e = 8$)

The total cost for an on-digonal submatrix is $C_{on} = 9961 + 11472 = 21433$ CLB slices. The total cost for an off-diagonal submatrix is $C_{off} = 10400 + 14240 = 24640$ CLB slices. The total latency is computed the same as in Section 7.2. The on-line delay for a sweep is 130 cycles.

7.3.2 Radix-2 parallel network

A CSVD arithmetic unit can be alternatively designed as a network of radix-2 parallel arithmetic operators. The library of Xilinx CORE arithmetic modules [Xil01], which can be scaled in terms of digit precision is used. Since the modules are defined for fixed-point arithmetic, appropriate exponent handling units are used to support floating-point arithmetic. The individual cost per module type, the total number of modules of each type used, and the latency, are shown in Table 7.6, assuming 24-bit operands and 8-bit exponents.

Operator	CLB slices	Triangularization				Diagonalization				Latency (cycles)
		On diag.		Off diag.		On diag.		Off diag.		
		#	Cost	#	Cost	#	Cost	#	Cost	
$+/-$	30	25	750	32	960	25	750	48	1440	1
$\times$	320	33	10560	48	15360	43	13760	64	20480	5
$\div$	936	10	9360	—	—	8	7488	—	—	24
$\sqrt{}$	422	9	3798	—	—	12	5064	—	—	25
Total cost			24468		16320		27062		21920	

Table 7.6: Cost of radix-2 parallel CSVD ($m = 24$, $e = 8$)

The total cost for an on-digonal submatrix is $C_{on} = 24468+27062 = 51530$ CLB slices. The total cost for an off-diagonal submatrix is $C_{off} = 16320+21920 = 38240$ CLB slices.

The total latency per sweep is computed as the sum of the latencies of individual modules along the critical path. Unlike in the on-line arithmetic cases, for the parallel arithmetic case there is no overlapping of dependent operations, since operations are based on loading parallel operands at the start of the operation, and

outputing parallel results at the end of the operation. Element exchange is performed in parallel after the elements of the off-diagonal submatrices are updated, and another sweep starts. The cycle dependancy graph for an on-diagonal submatrix and an off-diagonal submatrix using the Xilinx CORE parallel arithmetic modules [Xil01] is shown in Figure 7.8. The total latency per sweep is 202 cycles.

7.3.3 Cost comparison

An $n \times n$ matrix consists of $\frac{n}{2}$ on-diagonal submatrices and $\left(\frac{n}{2}\right)^2 - \frac{n}{2}$ off-diagonal submatrices. Then the total cost for the CSVD of an $n \times n$ matrix, assuming n is even is

$$C_{n\times n} = \left(\frac{n}{2}\right) C_{on} + \left(\left(\frac{n}{2}\right)^2 - \frac{n}{2}\right) C_{off} \tag{7.31}$$

The $RCNS_{2j,3}$ on-line approach, the radix 2 on-line approach, and the radix 2 parallel approach are compared in terms of cost measured in CLB slices for the design of various $n \times n$ CSVD arithmetic units. The results are shown in Table 7.7.

n	On-diagonal submatrices	Off-diagonal submatrices	$RCNS_{2j,3}$ on-line	Radix 2 on-line	Radix 2 parallel
2	1	—	26248	21433	51530
4	2	2	88496	92146	179540
8	4	12	320992	381412	665000
16	8	248	4673984	6282184	9895760

Table 7.7: Comparison of costs for CSVD ($m = 24$, $e = 8$)

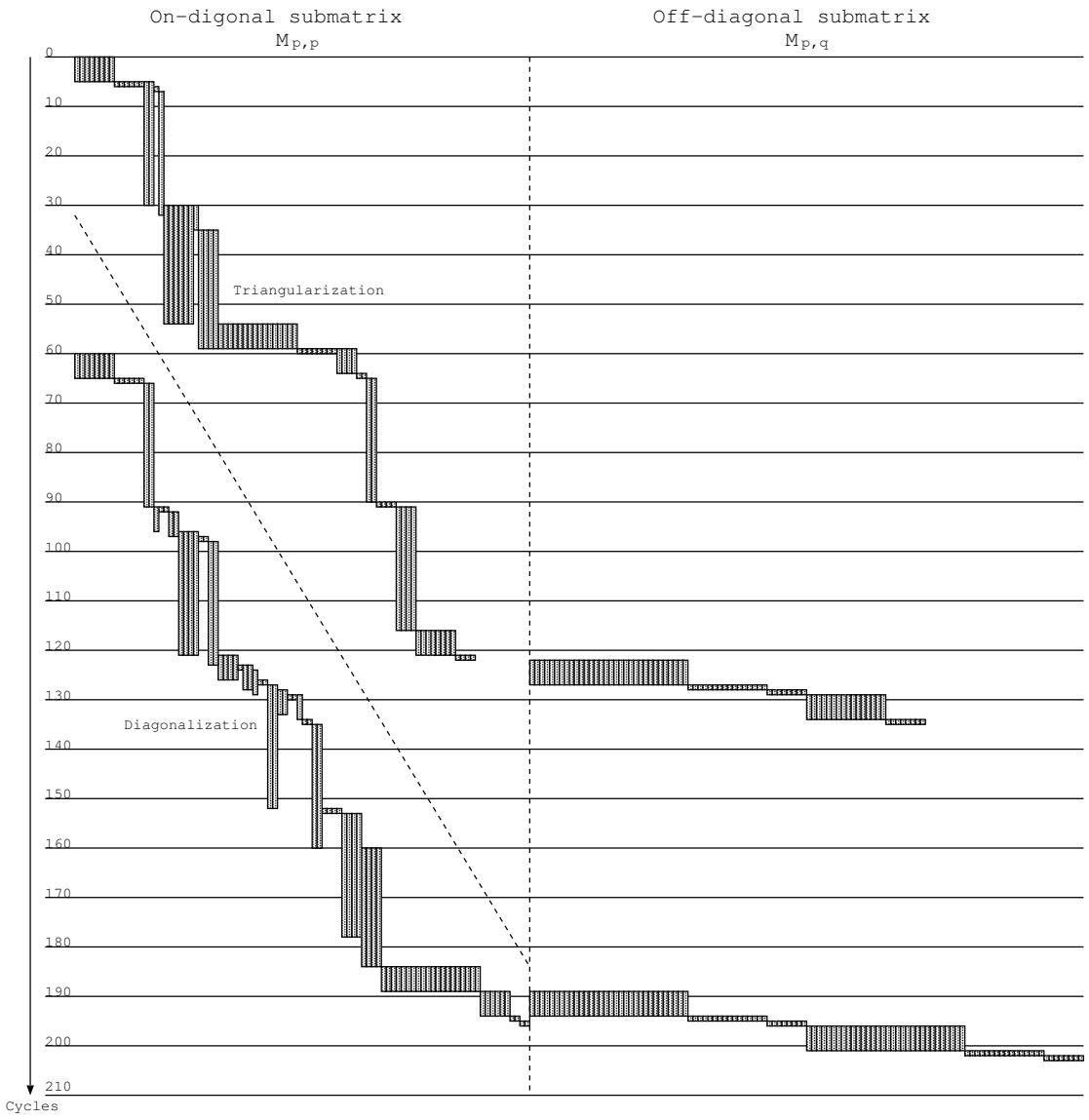


Figure 7.8: Cycle dependency graph for parallel CSVD

7.3.4 Delay comparison

The total latency is computed as the latency (or on-line delay) of a single sweep multiplied by the number of sweeps plus (only in the on-line approaches) the significand precision. A $n \times n$ CSVD requires $n - 1$ sweeps. Assuming that actual cycle times are set to be equivalent, then the delay will be measured in terms of cycles.

Table 7.8 compares the $RCNS_{2j,3}$ on-line approach, the radix 2 on-line approach, and the radix 2 parallel approach in terms of total cycles for the design of various $n \times n$ CSVD arithmetic units.

n	$RCNS_{2j,3}$ on-line	Radix 2 on-line	Radix 2 parallel
2	144	153	202
4	386	413	606
8	870	933	1414
16	1838	1973	3030

Table 7.8: Comparison of total cycles for CSVD ($m = 24$, $e = 8$)

7.4 Concluding Remarks

From the derivation of the algorithm and the implementation of the complex singular value decomposition (CSVD), we make the following general observations.

1. The $RCNS_{2j,3}$ approach achieves approximately a 25% increase in cost for an individual on-diagonal 2×2 submatrix computation, and approximately a

36% decrease in cost for an individual off-diagonal 2×2 submatrix computation, compared to the equivalent radix-2 on-line network approach, assuming significand precision $m = 24$ and exponent precision $e = 8$. As the size n of the matrix increases, the required number of off-diagonal computations grows at a rate $O(n^2)$, whereas the required number of on-diagonal compuations only grows at the rate $O(n)$. Specifically, for $n \geq 4$, the $RCNS_{2j,3}$ approach achieves a significantly lower cost than the equivalent radix-2 on-line network approach.

2. The $RCNS_{2j,3}$ approach achieves approximately a 47% decrease in cost for an individual on-diagonal 2×2 submatrix computation, and approximately a 52% decrease in cost for an individual off-diagonal 2×2 submatrix computation. As the size n of the matrix increases, the $RCNS_{2j,3}$ approach achieves a significantly lower cost than the equivalent radix-2 parallel network approach.

3. The $RCNS_{2j,3}$ approach achieves approximately a 6% reduction in latency in comparison to the radix-2 on-line network approach, due to a lower total on-line delay along the critical path of the $RCNS_{2j,3}$ network.

4. The $RCNS_{2j,3}$ approach achieves approximately a 39% reduction in latency in comparison to the radix-2 parallel network approach, due to the ability to overlap dependent operations in the $RCNS_{2j,3}$ approach, whereas the radix-2 parallel network approach requires all digits of an operation to be produced before a successive operation can start.

CHAPTER 8

Summary and Further Research

8.1 Summary

In this dissertation, we have presented a Redundant Complex Number System with radix 2 and largest digit 3, denoted $RCNS_{2j,3}$ for the specification and implementation of complex-number on-line floating-point arithmetic operators. This interleaves the real and imaginary components of a complex operation, such that the production of real and imaginary digits of the results alternate cycles. This yields the same equivalent output bit rate as equivalent networks of radix-2 on-line floating-point operators. Therefore, no significant gain in overall latency was achieved.

However, a significant reduction in cost for most operators is achieved for the $RCNS_{2j,3}$ approach due to the sharing of modules for the real and imaginary components, whereas the real network approach uses dedicated hardware for the real and imaginary components. Also, the implementations utilize digit-set recoding from digit set $\{-3,\ldots,3\}$ to digit set $\{-2,\ldots,2\}$ before the vector-digit multiplication step, eliminating the need for the production of vector multiples $3X$ and $-3X$ which require an extra addition step. The radix-2 network approach does

not have the ability to utilize digit-set recoding, since the digit set $\{-1, 0, 1\}$ is both minimally and maximally redundant.

In comparison, the $RCNS_{2j,3}$ achieves a 40% decrease in cost for the design of a complex number on-line floating-point multipler, a 35% decrease in cost for the design of a complex number on-line floating-point square unit, a 23% decrease in cost for the design of a complex number on-line floating-point conjugate multiplier, a 67% decrease in cost for the design of a complex number on-line floating-point divider, and a 59% decrease in cost for the design of a complex number on-line floating-point square root unit.

The designs for basic complex number floating-point arithmetic units were applied to the design of an arithmetic unit that computes the complex singular value decomposition (CSVD) of an $n \times n$ matrix. The design was compared to an equivalent radix-2 on-line network approach to demonstrate a 25% decrease in cost for a 2×2 on-diagonal processor, and a 36% decrease in cost for t 2×2 off-diagonal processor. The design was also compared to an equivalent radix-2 parallel network approach to demonstrate a 47% decrease in cost for a 2×2 on-diagonal processor, and a 52% decrease in cost for a 2×2 off-diagonal processor.

In terms of overall latency of the CSVD, the $RCNS_{2j,3}$ approach demonstrated a 6% decrease in latency compared to the equivalent radix-2 on-line network approach, and a 39% decrease in latency compared to the equivalent radix-2 parallel network approach using conventional arithmetic.

8.2 Future Research

We suggest the following topics for consideration of future research.

- Optimization of cost by reusing on-line modules within a network. The approach presented in Chapter 7 does not share modules during the triangularization or diagonalization operations within a sweep, but only shares modules between sweeps. A significant reduction in cost can be achieved by reusing modules by means of multiplexing input operands from multiple sources and utilizing multiple enable signals for modules.

- Dynamic reconfiguration of complex number on-line modules to be used as real number on-line modules with optimal latency. Since the $RCNS_{2j,3}$ arithmetic modules only differ from corresponding radix 4 modules in the handling and negating of carry-out signals, the modules can be dynamically reconfigured by introducing a flag bit for switching modes and inserting multiplexers to support multiple carry digit sources. This was done for a parallel combination radix$2j$/radix 4 multiplier in [AAH97]. Since the diagonalization rotation step for an on-diagonal matrix consists mainly of real number operations, the complex number operators from the triangularization rotation step for an on-diagonal submatrix can be dynamically reconfigured as real number operators for the diagonalization rotation step for the same on-diagonal submatrix. This can significantly reduce the overall cost.

- Optimal implementations of other complex number applications, such as transformations and digital filters. The complex singular value decomposition is only one of various applications utilizing complex number operators. Other applications include the Fast Fourier Transform and the complex FIR filter.

- Analysis of power consumption and comparison with other on-line and parallel approaches. The digit-wide interconnections lead to potentially reducing switching activity which is a key factor in the equation for power consumption.

APPENDIX A

Radix-2 On-line Floating-point Arithmetic Operations

In this section, we derive the cost and performance of radix-2 on-line floating-point arithmetic operations using radix-2 borrow-save encoding as described in Section 2.3. The area and delay characteristics obtained in this appendix are used in comparison with the $RCNS_{2j,3}$ on-line floating-point operators in Chapters 3-6, as well as the $RNCS_{2j,3}$ network design for a CSVD arithmetic unit in Chapter 7. Section A.1 gives the details for primitive radix-2 on-line operations, including (i) radix multiplier; (ii) borrow-save digit multiplier, and (iii) borrow-save digit adder. Section A.2 presents radix-2 on-line floating-point addition. Section A.3 presents radix-2 on-line floating-point multiplication. Section A.4 presents radix-2 on-line floating-point square. Section A.5 presents radix-2 on-line floating-point division. Section A.6 presents radix-2 on-line floating-point square root. In each of Sections A.2 to A.6, the algorithm, on-line delay, selection function, and implementation with associated cost and critical delay are shown.

A.1 Primitive modules

Radix-2 on-line floating point arithmetic units are implemented as a series of digit slice modules, each performing operations on a pair of digits. This is similar

to the design presented in Section 2.6, except that connections are only made between nearest neighbor modules. Primitive modules for radix-2 on-line arithmetic include: (i) borrow-save digit multipliers, (ii) borrow-save digit adders, and (iii) various latches and flip-flops (both bit-wide and digit-wide) for "appending" digits to vectors and performing radix multiplication (left shift). Each operation will be discussed at both a high-level description and at the binary level. Parameters including CLB cost and critical delay are also given.

A.1.1 Vector append

Appending a radix-2 borrow-save digit x_k to a vector $X[k-1]$ is identical to the design for the $RCNS_{2j,3}$ vector append step presented in Section 2.6.1, except that the latches input and output two-bit digits (as opposed to four-bit digits in the $RCNS_{2j,3}$ representation). Therefore, the steps will not be repeated in detail. The cost, assuming m-digit precision is $\lceil \frac{3m}{2} \rceil$ CLB slices. The critical delay $T_{CKO} + T_{ICK} + T_{NET} = (2.1 + T_{NET})$ns.

A.1.2 Radix multiplication

Performing the operation $Z = 2X[k]$, where $X[k] = (x_1 x_2, \ldots, x_k)$, consists of left shifting the digits, such that the bits of each digit $z_i = (z_i^+, z_i^-)$ are set as

$$\begin{aligned} z_i^+ &= x_{i+1}^+ \\ z_i^- &= x_{i+1}^- \end{aligned} \tag{A.1}$$

It can be implemented using a digit-wide register of D flip-flops, with a cost, assuming m-digit precision, of m CLB slices and a critical delay of $T_{CKO} + T_{ICK}$=2.1ns.

A.1.3 Vector-digit multiplication

Vector-digit multiplication is defined such that given a digit vector $X[k] = (x_1, \ldots, x_k)$, with digits $x_i \in \{-1, 0, 1\}$ and digit $y_k \in \{-1, 0, 1\}$, we want to compute $Z[k] = X[k]y_k$, where $Z[k] = (z_1, \ldots, z_k)$ and each $z_i \in \{-1, 0, 1\}$. This can be performed in parallel, where for each digit z_i,

$$\begin{aligned} z_i^+ &= x_i^+ \overline{x_i^-} y_k^+ \overline{y_k^-} + \overline{x_i^+} x_i^- \overline{y_k^+} y_k^- \\ z_i^- &= x_i^+ \overline{x_i^-} y_k^+ y_k^- + \overline{x_i^+} x_i^- y_k^+ \overline{y_k^-} \end{aligned} \tag{A.2}$$

A radix-2 vector-digit multiplier can be implemented as a series of radix-2 borrow-save digit multipliers. Each radix-2 borrow-save digit multiplier takes as input two radix-2 digits $x_i = (x_i^+, x_i^-)$ and $y_k = (y_k^+, y_k^-)$, and produces digit $z_i = (z_i^+, z_i^-)$, as shown in Figure A.1. The cost for a m-digit borrow-save vector-digit multiplier is m CLB slices and the critical delay is $T_{ILO} + 2T_{NET} = (0.6 + 2T_{NET})$ns.

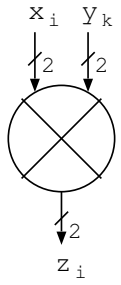


Figure A.1: Radix-2 borrow-save digit multiplier

A.1.4 Vector addition

Addition of vectors is used to update the residual based on new input digits. For the general addition of radix-2 vectors: $Z[k] = X[k] + Y[k]$, this can be realized by computing intermediate carry digits c_{i-1} and sum digits s_i of individual elements x_i and y_i, where $x_i + y_i = 2c_{i-1} + s_i$, such that

$$(c_{i-1}, s_i) = \begin{cases} (\overline{1}, 0) & \text{if } x_i + y_i = -2 \\ (\overline{1}, 1) & \text{if } x_i + y_i = -1 \text{ and } c_i = -1 \\ (0, \overline{1}) & \text{if } x_i + y_i = -1 \text{ and } c_i \neq -1 \\ (0, 0) & \text{if } x_i + y_i = 0 \\ (0, 1) & \text{if } x_i + y_i = 1 \text{ and } c_i \neq 1 \\ (1, \overline{1}) & \text{if } x_i + y_i = 1 \text{ and } c_i = 1 \\ (1, 0) & \text{if } x_i + y_i = 2 \end{cases} \tag{A.3}$$

Then $z_i = c_i + s_i$. The coding allows each output digit $z_i \in \{-1, 0, 1\}$ to be computed in parallel. For implementation, borrow-save addition consists of a network of PPM (plus-plus-minus) and MMP (minus-minus-plus) adder cells [GHM89], as described in Section 2.6. A radix-2 vector adder can be implemented as a series of radix-2 borrow-save digit adders. The two commonly used types of digit adders, two-input (denoted 2:1) and three-input (denoted 3:1) are described below.

A.1.4.1 2:1 borrow-save digit adder

A radix-2 2:1 borrow-save digit adder, as shown in Figure A.2, takes two radix-2 borrow-save digits x_i and y_i, carry-in bits t^+_{i+1} and t^-_{i+1}, and produces sum digit z_i and carry-out bits t^+_i and t^-_i. The symbol and corresponding schematics are shown in Figure A.2. The cost is 2 CLB slices and the delay is $2T_{ILO} + 2T_{NET} = (1.2 + 2T_{NET})$ns.

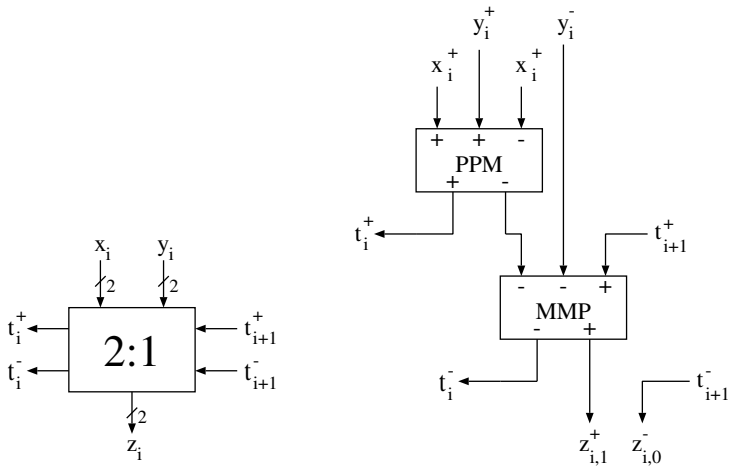


Figure A.2: Radix-2 2:1 borrow-save digit adder

A.1.4.2 3:1 borrow-save digit adder

A radix-2 3:1 borrow-save digit adder, as shown in Figure A.3, takes three radix-2 borrow-save digits w_i, x_i and y_i, carry-in bits $t^+_{i+1,1}$, $t^-_{i+1,1}$, $t^+_{i+1,0}$, and $t^-_{i+1,0}$, and produces sum digit z_i and carry-out bits $t^+_{i,1}$, $t^-_{i,0}$, $t^+_{i,0}$, and $t^-_{i,0}$. The cost is 4 CLB slices and the delay is $3T_{ILO} + 4T_{NET} = (1.8 + 4T_{NET})$ns.

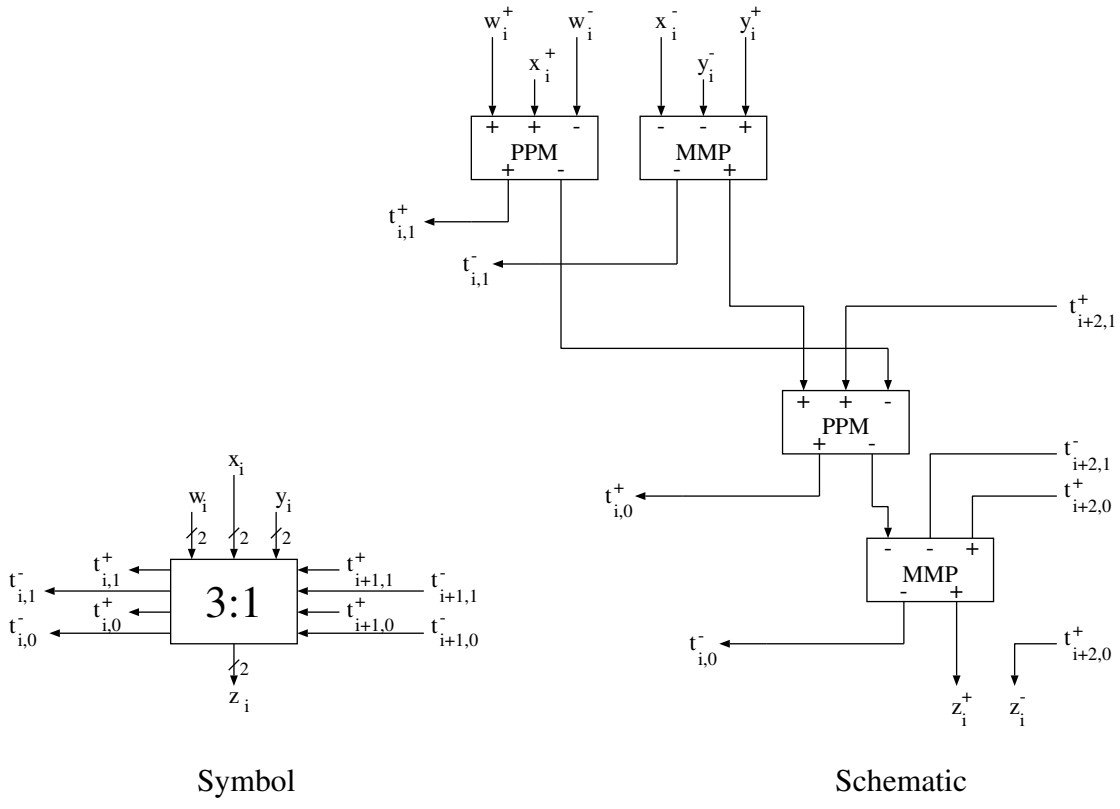


Figure A.3: Radix-2 3:1 borrow-save digit adder

A.2 Radix-2 On-line Floating-point Addition

Radix-2 on-line floating-point addition $z = x + y$ is defined such that given inputs $x = X \cdot 2^{e_x}$ and $y = Y \cdot 2^{e_y}$, the output $z = Z \cdot 2^{e_z}$ is produced such that

$$\begin{aligned} Z &= X + Y \\ e_z &= \max(e_x, e_y) \end{aligned} \tag{A.4}$$

The algorithm for radix-2 on-line floating-point addition is shown below, where the minimum on-line delay $\delta = 3$ [Tu90], and the selection points m_d $(d = -1, 0, 1)$ for the output digit selection function $z = SEL_{add}(\overline{W[k]})$ are shown in Table A.1.

Radix-2 On-line Floating-Point Addition

/* Initialization */

$e_d = e_x - e_y$

e_z=max(e_x, e_y)

$W[-\delta + 1] = 0$

$z_0 = 0$

for $k = -\delta + 2$ to 0 do

$$(x'_{k+\delta-1}, y'_{k+\delta-1}) = \begin{cases} (0, y_{k+\delta-1}) & \text{if } e_d < 0 \\ (x_{k+\delta-1}, 0) & \text{if } e_d > 0 \\ (x_{k+\delta-1}, y_{k+\delta-1}) & \text{if } e_d = 0 \end{cases}$$

$W[k] = 2(W[k-1]) + 2^{-\delta+1}(x'_{k+\delta-1} + y'_{k+\delta-1})$

end for

/* Recurrence */

for $k = 1$ to m do

$$(x'_{k+\delta-1}, y'_{k+\delta-1}) = \begin{cases} (0, y_{k+\delta-1}) & \text{if } k \le |e_d| \text{ and } e_d < 0 \\ (x_{k+\delta-1}, 0) & \text{if } k \le |e_d| \text{ and } e_d \ge 0 \\ (x_{k+\delta-1-|e_d|}, y_{k+\delta-1}) & \text{if } k > |e_d| \text{ and } e_d < 0 \\ (x_{k+\delta-1}, y_{k+\delta-1-|e_d|}) & \text{if } k > |e_d| \text{ and } e_d \ge 0 \end{cases}$$

$W[k] = 2(W[k-1] - z_{k-1}) + 2^{-\delta+1}(x'_{k+\delta-1} + y'_{k+\delta-1})$

$z_k = SEL_{add}(\overline{W[k]})$

$e_z = NORM(z_k, e_z, \delta)$

end for

	-1	0	1
L_d	$-3/2$	$-1/2$	$1/2$
U_d	$-1/2$	$1/2$	$3/2$
m_d	$-3/2$	$-1/2$	$1/2$

Table A.1: Selection points for radix-2 on-line floating-point addition

The design of a radix-2 on-line floating-point adder, using radix-2 borrow-save encoding, is shown in Figure A.4. The cost is summarized in Table A.2, assuming e-bit exponent and m-digit significand precision. The SUBE unit computes the difference of the exponents, not considering exponent underflow/overflow. The ALIGN unit performs alignment of operand y'. The SWAP unit exchanges the operands if necessary. The NORM unit normalizes the result by updating the output exponent e_z.

Module	CLB slices
SUBE	e
ALIGN	$1.5m$
SWAP	e
PPM/MMP	2
NORM	$2e$
Total cost	$1.5m + 4e + 2$

Table A.2: Cost of radix-2 on-line floating-point adder

The total cost is $1.5m + 4e + 2$ CLB slices. The critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

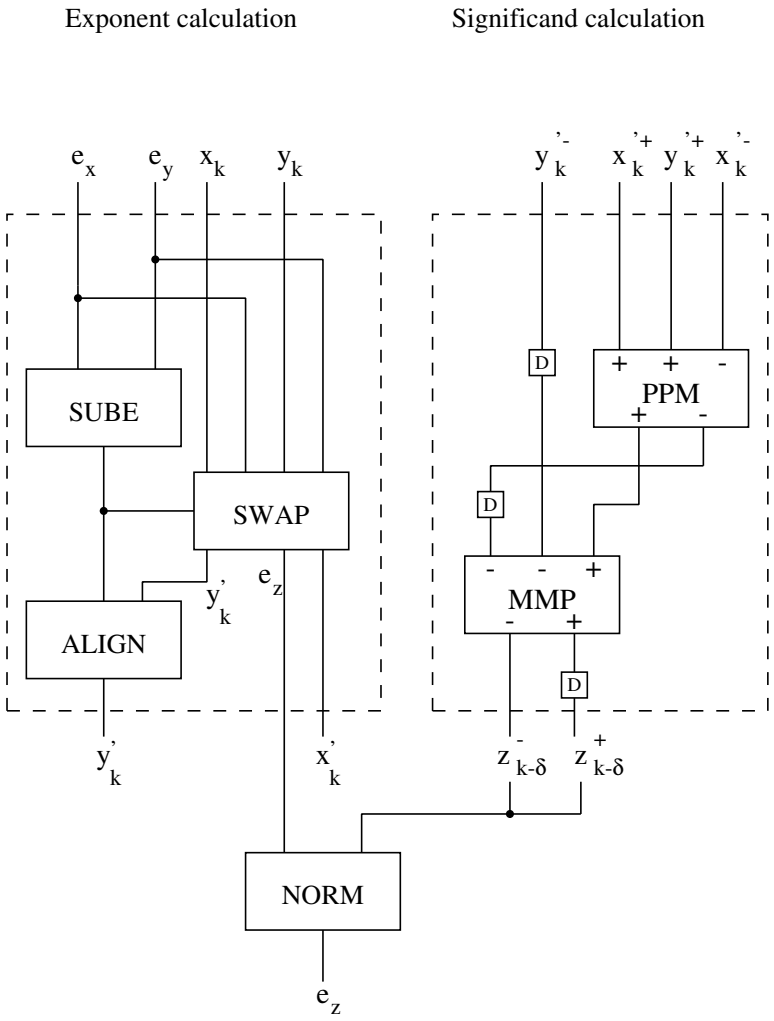


Figure A.4: Radix-2 on-line floating-point adder

A.3 Radix-2 On-line Floating-point Multiplication

Radix-2 on-line floating-point multiplication $z = xy$ is defined such that given inputs $x = X \cdot 2^{e_x}$ and $y = Y \cdot 2^{e_y}$, the output $z = Z \cdot 2^{e_z}$ is produced such that

$$\begin{aligned} Z &= XY \\ e_z &= e_x + e_y \end{aligned} \tag{A.5}$$

The algorithm for radix-2 on-line floating-point multiplication is shown below, with on-line delay $\delta = 4$ [Tu90]. The selection points m_d $(d = -1, 0, 1)$ for the output digit selection function $z = SEL_{mul}(\overline{W[k]})$ are shown in Table A.3.

Radix-2 On-line Floating-Point Multiplication

/* Initialization */
$e_z = e_x + e_y$
$W[-\delta + 1] = 0$
$X[-\delta + 1] = 0$
$Y[-\delta + 1] = 0$
$z_0 = 0$

for $k = -\delta + 2$ to 0 do
$X[k] = X[k-1] + x_{k+\delta-1}2^{-k-\delta+1}$
$W[k] = 2(W[k-1]) + 2^{-\delta+1}(X[k]y_{k+\delta-1} + Y[k-1]x_{k+\delta-1})$
$Y[k] = Y[k-1] + y_{k+\delta-1}2^{-k-\delta+1}$
end for

/* Recurrence */
for $k = 1$ to m do
$X[k] = X[k-1] + x_{k+\delta-1}2^{-k-\delta+1}$
$W[k] = 2(W[k-1] - z_{k-1}) + 2^{-\delta+1}(X[k]y_{k+\delta-1} + Y[k-1]x_{k+\delta-1})$
$z_k = SEL_{mul}(\overline{W[k]})$
$Y[k] = Y[k-1] + y_{k+\delta-1}2^{-k-\delta+1}$
$e_z = NORM(z_k, e_z, \delta)$
end for

A radix-2 on-line floating-point multiplier can be designed as a series of modular slices, where each slice consists of two borrow-save digit multipliers, a 3:1

	-1	0	1
L_d	$-7/4$	$-3/4$	$1/4$
U_d	$-1/4$	$3/4$	$7/4$
m_d	$-3/2$	$-1/2$	$1/2$

Table A.3: Selection points for radix-2 on-line floating-point multiplication

borrow-save digit adder, a pair of digit-wide latches, a D flip-flop, and a digit-wide register of D flip-flops. The most significant digit of the recurrence is determined using one MSREC unit which performs output digit selection as well as handles potential most significant carry-out bits from the adder. The ADDE unit adds the two input exponents to produce the exponent of the output, not considering exponent underflow/overflow. The NORM unit normalizes the result by updating the output exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point multiplier is shown in Figure A.5. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table A.4. The total cost is $6m+3e+2$ CLB slices and the critical delay is $T_{CKO}+T_{ICK}+4T_{ILO}+5T_{NET} = (4.5+5T_{NET})$ns.

Module	Count	CLB slices
ADDE	1	e
BSD mult. ($\otimes$)	$2m$	$2m$
BSD adder (3:1)	m	$4m$
MSREC	1	2
NORM	1	$2e$
Total cost		$6m+3e+2$

Table A.4: Cost of radix-2 on-line floating-point multiplier

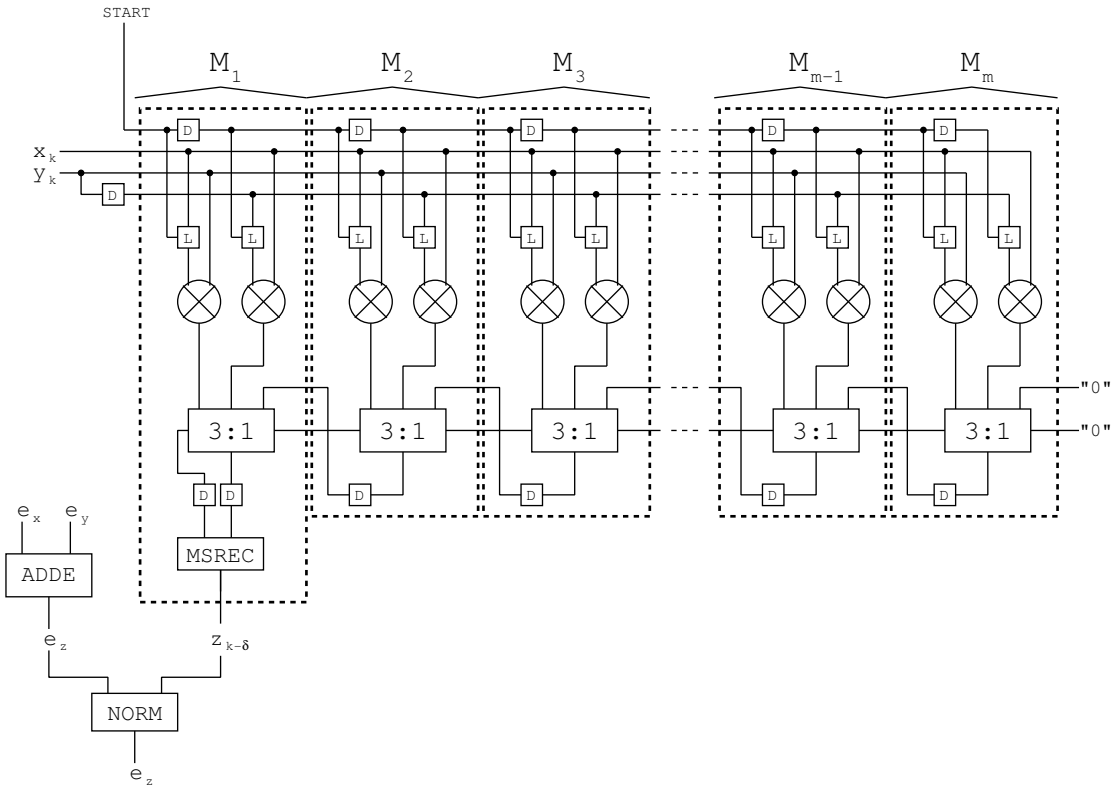


Figure A.5: Radix-2 on-line floating-point multiplier

A.4 Radix-2 On-line Floating-point Square

Radix-2 on-line floating-point square $z = x^2$ is defined such that given input $x = X \cdot 2^{e_x}$, the output $z = Z \cdot 2^{e_z}$ is produced such that

$$\begin{aligned} Z &= X^2 \\ e_z &= 2e_x \end{aligned} \tag{A.6}$$

The algorithm for radix-2 on-line floating-point square is shown below, with on-line delay $\delta = 4$ [Tu90]. The selection points m_d $(d = -1, 0, 1)$ for the output digit selection function $z = SEL_{sq}(\overline{W[k]})$ are shown in Table A.5.

Radix-2 On-line Floating-Point Square

/* Initialization */
$e_z = 2e_x$
$W[-\delta + 1] = 0$
$X[-\delta + 1] = 0$
$z_0 = 0$

for $k = -\delta + 2$ to 0 do
$W[k] = 2(W[k-1]) + 2^{-\delta+1}(2X[k-1] + x_{k+\delta-1}2^{-k-\delta+1})x_{k+\delta-1})$
$X[k] = X[k-1] + x_{k+\delta-1}2^{-k-\delta+1}$
end for

/* Recurrence */
for $k = 1$ to m do
$W[k] = 2(W[k-1] - z_{k-1}) + 2^{-\delta+1}(2X[k-1] + x_{k+\delta-1}2^{-k-\delta+1})x_{k+\delta-1})$
$z_k = SEL_{sq}(\overline{W[k]})$
$X[k] = X[k-1] + y_{k+\delta-1}2^{-k-\delta+1}$
$e_z = NORM(z_k, e_z, \delta)$
end for

A radix-2 on-line floating-point square unit can be designed as a series of modular slices, where each slice consists of one borrow-save digit multiplier, a 2:1 borrow-save digit adder, a digit-wide latches, a D flip-flop, and a digit-wide regis-

	-1	0	1
L_d	$-7/4$	$-3/4$	$1/4$
U_d	$-1/4$	$3/4$	$7/4$
m_d	$-3/2$	$-1/2$	$1/2$

Table A.5: Selection points for radix-2 on-line floating-point square

ter of D flip-flops. The most significant digit of the recurrence is determined using one MSREC unit which performs output digit selection as well as handles potential most significant carry-out bits from the adder. The MULT2E unit multiplies the input exponent by 2 to produce the exponent of the output. The NORM unit normalizes the result by updating the output exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point square unit is shown in Figure A.6. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table A.6.

Module	Count	CLB slices
MULT2E	1	e
2-to-1 MUX	m	m
BSD mult. ($\otimes$)	m	m
BSD adder (2:1)	m	$2m$
MSREC	1	2
NORM	1	$2e$
Total cost		$4m + 3e + 2$

Table A.6: Cost of radix-2 on-line floating-point square unit

The total cost is $4m + 3e + 2$ CLB slices and the critical delay is $T_{CKO} + T_{ICK} + 4T_{ILO} + 5T_{NET} = (4.5 + 5T_{NET})$ns.

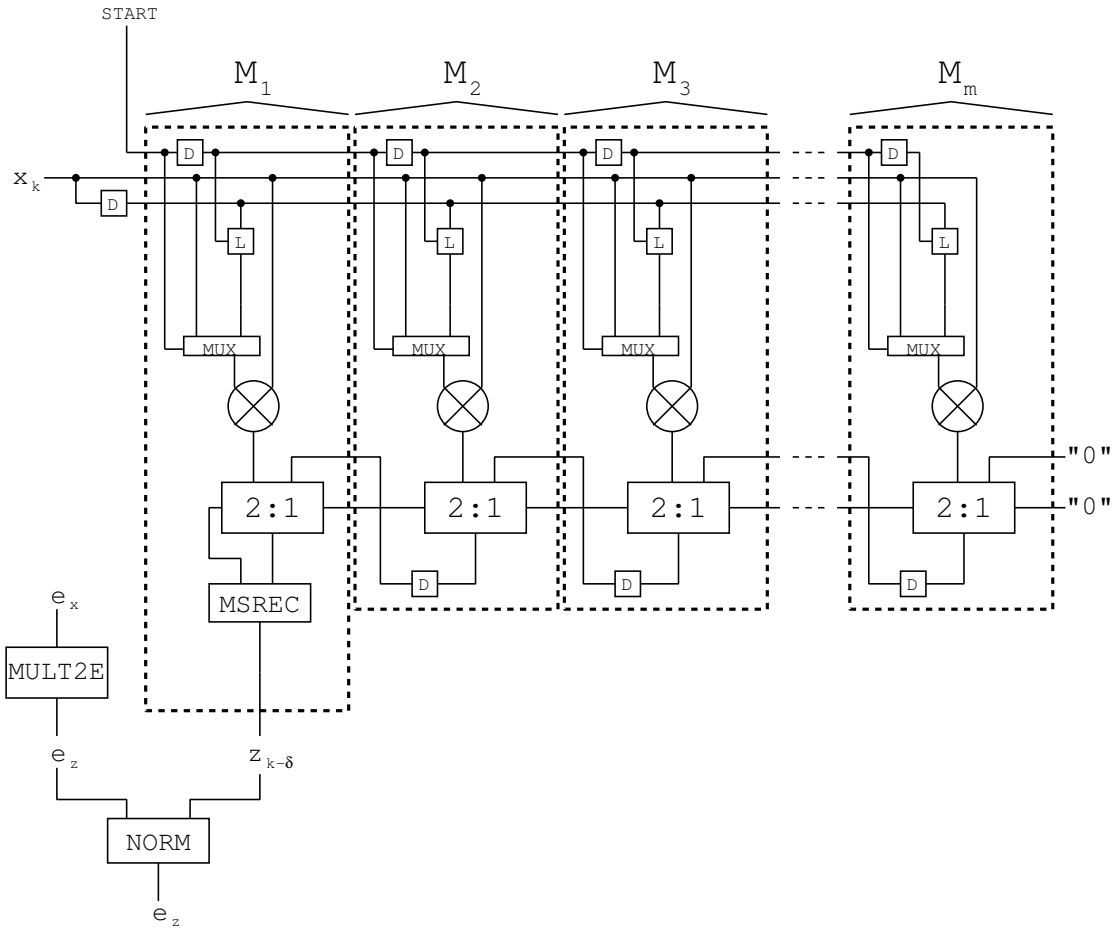


Figure A.6: Radix-2 on-line floating-point square unit

A.5 Radix-2 On-line Floating-point Division

Radix-2 on-line floating-point division $z = x/y$ is defined such that given inputs $x = X \cdot 2^{e_x}$ and $y = Y \cdot 2^{e_y}$, the output $z = Z \cdot 2^{e_z}$ is produced such that

$$\begin{aligned} Z &= \tfrac{X}{Y} \\ e_z &= e_x - e_y \end{aligned} \tag{A.7}$$

The algorithm for radix-2 on-line floating-point division is shown below, with on-line delay $\delta = 5$ [Tu90]. The selection points m_d $(d = -1, 0, 1)$ for the output digit selection function $z = SEL_{div}(\overline{W[k]})$ are shown in Table A.7.

Radix-2 On-line Floating-Point Division

/* Initialization */
$e_z = e_x - e_y$
$W[-\delta + 1] = 0$
$Y[-\delta + 1] = 0$
$Z[0] = 0$
$z_0 = 0$

for $k = -\delta + 2$ to 0 do
$W[k] = 2W[k-1] + 2^{-\delta+1} x_{k+\delta-1}$
$Y[k] = Y[k-1] + y_{k+\delta-1} 2^{-k-\delta+1}$
end for

/* Recurrence */
for $k = 1$ to m do
$W[k] = 2(W[k-1] - Y[k-1] z_{k-1}) + 2^{-\delta+1}(x_{k+\delta-1} - Z[k-1] y_{k+\delta-1})$
$z_k = SEL_{div}(\overline{W[k]})$
$Z[k] = Z[k-1] + z_k 2^{-k}$
$Y[k] = Y[k-1] + y_{k+\delta-1} 2^{-k-\delta+1}$
$e_z = NORM(e_z, z_k, \delta)$
end for

A radix-2 on-line floating-point dividier can be designed as a series of modular slices, where each slice consists of two borrow-save digit multipliers, a 3:1 borrow-

	-1	0	1
L_d	$-2Y[k] + 2^{-3}$	$-Y[k] + 2^{-2}$	2^{-3}
U_d	-2^{-3}	$Y[k] - 2^{-3}$	$2Y[k] - 2^{-3}$
m_d	$-\frac{7}{8}$	$-\frac{3}{8}$	$\frac{1}{4}$

Table A.7: Selection points for radix-2 on-line floating-point division

save digit adder, a pair of digit-wide latches, a D flip-flop, and a digit-wide register of D flip-flops. The most significant digit of the recurrence is determined using one MSREC unit which handlea potential most significant carry-out bits from the adders. The SELDIV unit selects the output digit z_k based on the truncated residual, denoted $\overline{W[k]}$. The SUBE unit subtracts the two input exponents to produce the exponent of the output, not considering exponent underflow/overflow. The digit $x_{k+\delta-1}$ is appended to the most significant end of the vector product $Z[k-1]y_{k+\delta-1}$. The NORM unit normalizes the result by updating the output exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point divider is shown in Figure A.7. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table A.8.

Module	Count	CLB slices
SUBE	1	e
BSD mult. ($\otimes$)	$2m$	$2m$
BSD adder (3:1)	m	$4m$
SELDIV	1	9
NORM	1	$2e$
Total cost		$6m + 3e + 9$

Table A.8: Cost of radix-2 on-line floating-point divider

The total cost is $6m+3e+9$ CLB slices and the critical delay is $T_{CKO}+T_{ICK}+4T_{ILO}+5T_{NET}=(4.5+5T_{NET})$ns.

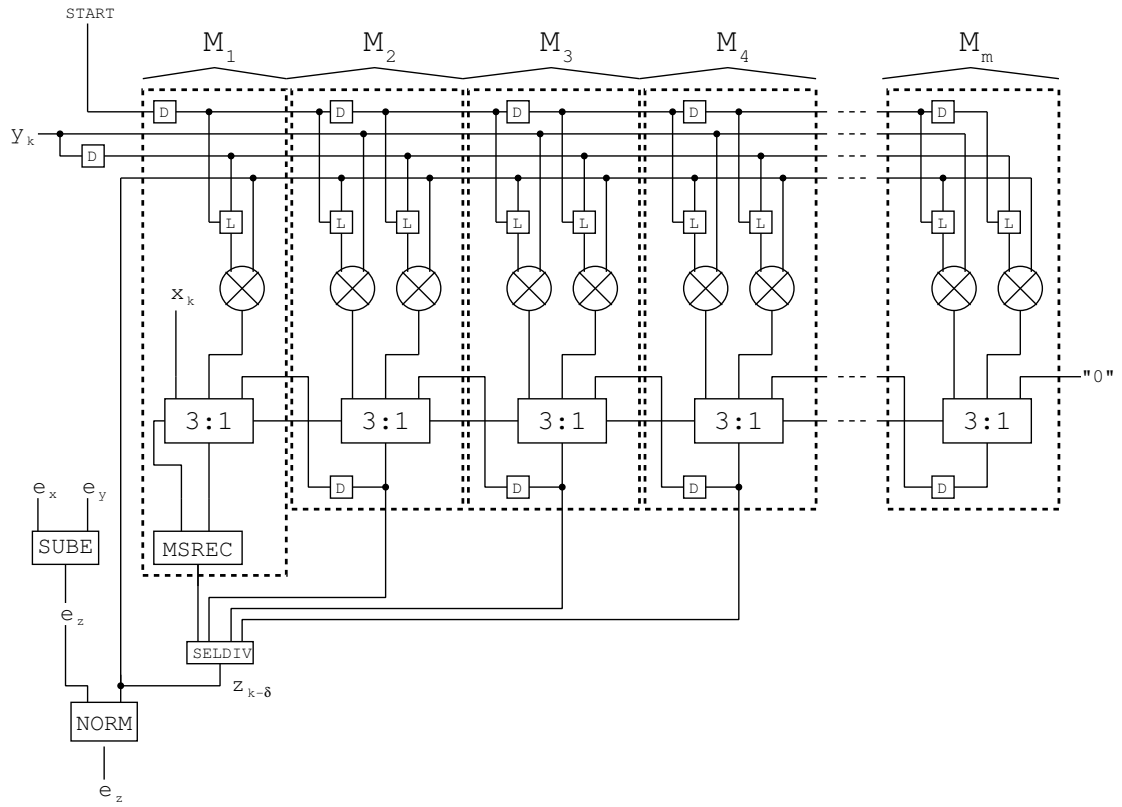


Figure A.7: Radix-2 on-line floating-point divider

A.6 Radix-2 On-line Floating-point Square Root

Radix-2 on-line floating-point square root $z = \sqrt{x}$ is defined such that given input $x = X \cdot 2^{e_x}$, the output $z = Z \cdot 2^{e_z}$ is produced such that

$$\begin{aligned} Z &= \sqrt{X} \\ e_z &= \lceil \tfrac{e_x}{2} \rceil \end{aligned} \tag{A.8}$$

To ensure e_z is an integer, if e_x is odd, then the mantissa X is initially scaled by 2^{-1} and e_x is incremented by 1 to make e_x even. The algorithm for radix-2 on-line floating-point square root is shown below, with on-line delay $\delta = 4$ [Tu90]. The selection points m_d $(d = -1, 0, 1)$ for the output digit selection function $z = SEL_{sqrt}(\overline{W[k]})$ are shown in Table A.9.

Radix-2 On-line Floating-Point Square Root

/* Initialization */

$e_z = \lceil \frac{e_x}{2} \rceil$

$W[-\delta + 1] = Z[0] = 0$

$x_0 = z_0 = 0$

for $k = -\delta + 2$ to 0 do

$$x'_{k+\delta-1} = \begin{cases} x_{k+\delta-2} & \text{if } e_x \text{ is odd} \\ x_{k+\delta-1} & \text{if } e_x \text{ is even} \end{cases}$$

$W[k] = 2W[k-1] + 2^{-\delta+1} x'_{k+\delta-1}$

end for

/* Recurrence */

for $k = 1$ to m do

$$x'_{k+\delta-1} = \begin{cases} x_{k+\delta-2} & \text{if } e_x \text{ is odd} \\ x_{k+\delta-1} & \text{if } e_x \text{ is even} \end{cases}$$

$W[k] = 2(W[k-1] - (2Z[k-2] + z_{k-1}2^{-k+1})z_{k-1}) + 2^{-\delta+1} x'_{k+\delta-1}$

$z_k = SEL_{sqrt}(\overline{W[k]}, \overline{2Z[k-2]})$

$Z_k = Z[k-1] + z_k 2^{-k}$

$e_z = NORM(e_z, z_k, \delta)$

end for

	-1	0	1
L_d	$-2(2Z[k-2])+2^{-2}$	$-(2Z[k-2])+2^{-2}$	2^{-2}
U_d	-2^{-2}	$(2Z[k-2])-2^{-2}$	$2(2Z[k-2])-2^{-2}$
m_d	$-\frac{7}{4}$	$-\frac{3}{4}$	$\frac{1}{2}$

Table A.9: Selection points for radix-2 on-line floating-point square root

A radix-2 on-line floating-point square root unit can be designed as a series of modular slices, where each slice consists of one borrow-save digit multiplier, a 2:1 borrow-save digit adder, a digit-wide latch, a D flip-flop, a digit-wide register of D flip-flops, and a 2-to-1 MUX for appropriately appending digit z_{k-1} to vector $2Z[k-2]$. The most sigificant digits of the recurrence is determined using one MSREC unit which handles potential most significant carry-out bits from the adder. The DIV2E unit divides the operand exponent by two to produce the exponent of the output, not considering exponent underflow/overflow. A single 2-to-1 MUX determines whether digit $x_{k+\delta-1}$ or $x_{k+\delta}$ is used each cycle, depending on whether the operand exponent is odd or even. The selected digit is appended to the most signficant end of the vector product $(2Z[k-2]+z_{k-1}2^{-k+1})z_{k-1}$. The SELSQRT unit selects the output digit z_k based on $\overline{W[k]}$. The NORM unit normalizes the result by updating the exponent e_z. The design of a m-digit significand and e-bit exponent on-line floating-point square unit is shown in Figure 6.1. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table A.10. The total cost is $4m+3e+15$ CLB slices and the critical delay is $T_{CKO}+T_{ICK}+4T_{ILO}+5T_{NET}=(4.5+5T_{NET})$ns.

Module	Count	CLB slices
DIV2E	1	e
2-to-1 MUX	m	m
BSD mult. ($\otimes$)	m	m
2:1 BSD adder	$m-1$	$2m-2$
3:1 BSD adder	1	4
MSREC	1	2
SELSQRT	1	11
NORM	1	$2e$
Total cost		$4m+3e+15$

Table A.10: Cost of radix-2 on-line floating-point square root unit

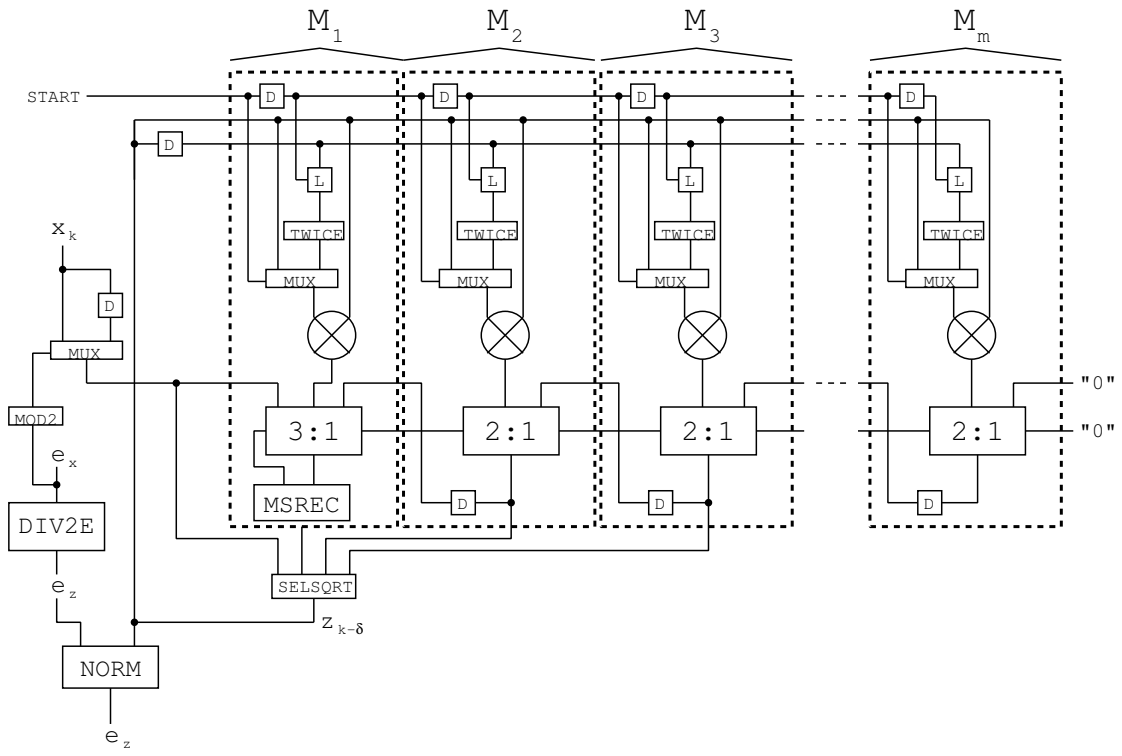


Figure A.8: Radix-2 on-line floating-point square root unit

APPENDIX B

VHDL Source Code for Modules

In this appendix, we present the VHDL source code for all implemented modules. Each module was tested for correctness using the Xilinx Foundation 3.1i design tool [Xil01], compiling the VHDL code into a netlist, performing behavioral simulation with an exhaustive testbench, synthesizing the netlist onto a Virtex FPGA to calculate and verify the CLB slice cost, and performing timing analysis to determine the critical path.

Networks of on-line arithmetic modules were constructed using the Astra design tool [Ast02], which translates command-line arithmetic expressions into networks of arithmetic modules. The specific library to be used can be determined at run time, as well as the precision of modules. Astra outputs the VHDL code corresponding to the arithmetic operators within the input expressions, as well as an estimate of CLB slice cost and the total latency of the network, independent of the actual FPGA mapping.

Section B.1 lists the VHDL source code for conventional binary modules. Section B.2 lists the VHDL source code for radix-2 on-line floating-point library modules. Section B.3 lists the VHDL source code for $RCNS_{2j,3}$ on-line floating-point library modules.

B.1 Conventional Binary Modules

```
-------------------------------------------------------------------
--  Module adde
--  Adds two e-bit exponents
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY adde IS
  GENERIC (e: natural);
    PORT (clk: IN std_logic;
          x,y: IN std_logic_vector(1 TO e);
          z: OUT std_logic_vector(1 TO e));
END adde;

ARCHITECTURE rtl OF adde IS

  SIGNAL cout: std_logic_vector(1 to e+1);

    BEGIN
      PROCESS(clk)
        BEGIN
          FOR i IN 1 TO e LOOP
            z(i) <= x(i) XOR y(i) XOR cout(i+1);
            cout(i) <= (x(i) AND y(i)) OR
                       (cout(i+1) AND (x(i) OR y(i)));
          END LOOP;
          cout(e+1) <= '0';
        END PROCESS;
    END rtl;

-------------------------------------------------------------------
-- Module dff1
-- 1-bit D flip-flop
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dff1 IS
  PORT (d, clk, reset: IN std_logic;
        q: OUT std_logic);
END dff1;

ARCHITECTURE rtl OF dff1 IS
  BEGIN
    PROCESS(clk, reset)
      BEGIN
        IF (reset='1') THEN
          q <= '0';
        ELSIF (clk'event AND clk='1') THEN
          q <= d;
        END IF;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module dffm
--  Sequence of m 1-bit D flip-flops
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY dffm IS
  GENERIC (m: natural);
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
END dffm;

ARCHITECTURE rtl OF dffm IS

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  SIGNAL temp_q: std_logic_vector(0 TO m);

  BEGIN
    f1: FOR i IN 1 TO m GENERATE
      d1: dff1 PORT MAP (clk=>clk, reset=>reset, d=>temp_q(i-1),
                         q=>temp_q(i));
    END GENERATE;

    temp_q(0) <= d;
    q <= temp_q(m);
END rtl;

-------------------------------------------------------------------
--  Module div2e
--  Divides an e-bit exponent by 2
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY div2e IS
  GENERIC (e: natural);
    PORT (clk: IN std_logic;
          x: IN std_logic_vector(1 TO e);
          z: OUT std_logic_vector(1 TO e));
END div2e;

ARCHITECTURE rtl OF div2e IS
  BEGIN
    PROCESS(clk)
      BEGIN
        z <= x(1) & x(1) & x(2 TO e-1);
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module mmp
--  Minus-minus-plus 1-bit adder
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mmp IS
  PORT (min1, min2, pin1: IN std_logic;
        mout, pout: OUT std_logic);
END mmp;

ARCHITECTURE rtl OF mmp IS
  BEGIN
    mout <= NOT((min1 NOR min2) OR (pin1 AND (min1 NAND min2)));
```

```
    pout <= min1 XOR min2 XOR pin1;
  END rtl;

-------------------------------------------------------------------
--  Module mult2e
--  Multiplies an e-bit exponent by 2
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mult2e IS
  GENERIC (e: natural);
    PORT (clk: IN std_logic;
          x: IN std_logic_vector(1 TO e);
          z: OUT std_logic_vector(1 TO e));
END mult2e;

ARCHITECTURE rtl OF mult2e IS
  BEGIN
    PROCESS(clk)
      BEGIN
        z <= x(1) & x(3 to e) & '0';
      END PROCESS;
  END rtl;
-------------------------------------------------------------------
-- Module ppm
-- Plus-plus-minus 1-bit adder
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ppm IS
  PORT (pin1, pin2, min1: IN std_logic;
        pout, mout: OUT std_logic);
END ppm;

ARCHITECTURE rtl OF ppm IS
  BEGIN
    pout <= (pin1 AND pin2) OR (NOT min1 AND (pin1 OR pin2));
    mout <= pin1 XOR pin2 XOR min1;
  END rtl;

-------------------------------------------------------------------
--  Module regm
--  Register of m 1-bit D flip-flops
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY regm is
  GENERIC (m: natural);
    PORT (clk, reset: IN std_logic;
          d: IN std_logic_vector(1 TO m);
          q: OUT std_logic_vector(1 TO m));
END regm;

ARCHITECTURE rtl OF regm IS
  BEGIN
    PROCESS(clk, reset)
      BEGIN
        IF (reset='1') THEN
```

```
          q <= (OTHERS=>'0');
        ELSIF (clk'event AND clk='1') THEN
          q <= d;
        END IF;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module sube
--  Subtracts two e-bit exponents
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY sube IS
  GENERIC (e: natural);
    PORT (clk: IN std_logic;
          x,y: IN std_logic_vector(1 TO e);
          z: OUT std_logic_vector(1 TO e));
END sube;

ARCHITECTURE rtl OF sube IS

  SIGNAL cout: std_logic_vector(1 TO e+1);

  BEGIN
    PROCESS(clk)
      BEGIN
        FOR i IN 1 TO e LOOP
          z(i) <= x(i) XOR NOT y(i) XOR cout(i+1);
          cout(i) <= (x(i) AND NOT y(i)) OR
                     (cout(i+1) AND (x(i) OR NOT y(i)));
        END LOOP;
        cout(e+1) <= '1';
     END PROCESS;
  END rtl;
```

B.2 Radix-2 On-line Floating-point Library Modules

```
-------------------------------------------------------------------
--  Module align_r2
--  Radix-2 operand alignment module
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY align_r2 IS
  GENERIC (m: natural; e: natural);
    PORT (clk, reset: IN std_logic;
          en: IN std_logic_vector(1 TO e);
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
END align_r2;

ARCHITECTURE rtl OF align_r2 IS

  COMPONENT decoder_256
    PORT (clk: IN std_logic;
          x: IN std_logic_vector(1 TO 8);
          z: OUT std_logic_vector(0 TO 255));
  END COMPONENT;

  COMPONENT mux21_r2
    PORT( clk : IN std_logic;
          x0,x1: IN std_logic_vector(1 TO 2);
          sel: in std_logic;
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dff1d_r2
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  SIGNAL temp_q: std_logic_vector(1 TO 2*m+2);
  SIGNAL mux_in: std_logic_vector(1 TO 2*m);
  SIGNAL dec_en: std_logic_vector(0 TO 255);
  SIGNAL en_in: std_logic_vector(0 to 7);

  BEGIN
    d1: decoder_256 PORT MAP (clk=>clk, x=>en_in, z=>dec_en);
    f1: FOR i IN 0 TO m-1 GENERATE
      m1: mux21_r2
        PORT MAP (clk=>clk, x0=>temp_q(2*i+3 to 2*i+4),
                  x1=>d, sel=>dec_en(i),
                  z=>mux_in(2*i+1 to 2*i+2));
      d2: dff1d_r2
        PORT MAP (clk=>clk, reset=>reset,
                  d=>mux_in(2*i+1 to 2*i+2),
                  q=>temp_q(2*i+1 to 2*i+2));
    END GENERATE;

    temp_q(2*m+1 to 2*m+2) <= "00";
    q <= temp_q(1 to 2);
  END rtl;
```

```
------------------------------------------------------------------
--  Module bsadd21_r2
--  Radix-2 borrow-save 2-digit adder
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bsadd21_r2 IS
  PORT (clk: IN std_logic;
        x,y: IN std_logic_vector(1 TO 2);
        pin, min: IN std_logic;
        pout, mout: OUT std_logic;
        z: OUT std_logic_vector(1 TO 2));
END bsadd21_r2;

ARCHITECTURE rtl OF bsadd21_r2 IS

  COMPONENT ppm
    PORT (pin1, pin2, min1: IN std_logic;
          pout, mout: OUT std_logic);
  END COMPONENT;

  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;

  SIGNAL tpout, tmout : std_logic_vector(1 to 2);

  BEGIN
    p1: ppm
      PORT MAP (pin1=>x(1), pin2=>y(1), min1=>x(2),
                pout=>tpout(1), mout=>tmout(1));
    m1: mmp
      PORT MAP (min1=>tmout(1), min2=>y(2), pin1=>pin,
                mout=>tmout(2), pout=>tpout(2));

    pout <= tpout(1);
    mout <= tmout(2);
    z(1) <= tpout(2);
    z(2) <= min;
  END rtl;

------------------------------------------------------------------
--  Module bsadd31_r2
--  Radix-2 borrow-save 3-digit adder
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY bsadd31_r2 IS
  PORT (clk: IN std_logic;
        w,x,y: IN std_logic_vector(1 TO 2);
        pin, min: IN std_logic_vector(1 TO 2);
        pout, mout: OUT std_logic_vector(1 TO 2);
        z: OUT std_logic_vector(1 TO 2));
END bsadd31_r2;

ARCHITECTURE rtl OF bsadd31_r2 IS

  COMPONENT ppm
```

```
    PORT (pin1, pin2, min1: IN std_logic;
          pout, mout: OUT std_logic);
  END COMPONENT;
  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;
  SIGNAL tpout, tmout: std_logic_vector(1 TO 4);
  BEGIN
    p1: ppm
      PORT MAP (pin1=>w(1), pin2=>x(1), min1=>w(2),
                pout=>tpout(1), mout=>tmout(1));
    m1: mmp
      PORT MAP (min1=>x(2), min2=>y(2), pin1=>y(1),
                mout=>tmout(2), pout=>tpout(2));
    p2: ppm
      PORT MAP (pin1=>tpout(2), pin2=>pin(1), min1=>tmout(1),
                pout=>tpout(3), mout=>tmout(3));
    m2: mmp
      PORT MAP (min1=>tmout(3), min2=>min(1), pin1=>pin(2),
                mout=>tmout(4), pout=>tpout(4));
    pout(1) <= tpout(1);
    mout(1) <= tmout(2);
    pout(2) <= tpout(3);
    mout(2) <= tmout(4);
    z(1) <= tpout(4);
    z(2) <= min(2);
  END rtl;

-------------------------------------------------------------------
--  Module bsdcomp_r2
--  Radix-2 borrow-save digit comparator
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bsdcomp_r2 IS
  PORT (x,y: IN std_logic_vector(1 TO 2);
        gg_in, g_in, e_in, s_in, ss_in: IN std_logic;
        gg_out, g_out, e_out, s_out, ss_out: OUT std_logic);
END bsdcomp_r2;

ARCHITECTURE rtl OF bsdcomp_r2 IS
  BEGIN
    gg_out <= gg_in OR (x(1) AND NOT x(2) AND NOT y(1) AND y(2)
              AND (e_in OR g_in));
    ss_out <= ss_in OR (NOT x(1) AND x(2) AND y(1) AND NOT y(2)
              AND (e_in OR s_in));
    g_out <= (g_in AND NOT (NOT x(1) AND x(2) AND y(1) AND
              NOT y(2))) OR (e_in AND ((NOT(x(1) XOR x(2))
              AND NOT y(1) AND y(2)) OR (x(1) AND NOT x(2)
              AND NOT(y(1) XOR y(2)))));
    s_out <= (s_in AND NOT (x(1) AND NOT x(2) AND NOT y(1) AND
              y(2))) OR (e_in AND ((NOT x(1) AND x(2) AND
```

```
              NOT(y(1) XOR y(2))) OR (NOT(x(1) XOR x(2)) AND
              y(1) AND NOT y(2))));
    e_out <= (e_in AND ((NOT(x(1) XOR x(2)) AND NOT(y(1)
             XOR y(2))) OR (x(1) AND NOT x(2) AND y(1) AND
             NOT y(2)) OR (NOT x(1) AND x(2) AND NOT y(1) AND
             y(2)))) OR (s_in AND x(1) AND NOT x(2) AND NOT y(1)
             AND y(2)) OR (g_in AND NOT x(1) AND x(2) AND
             y(1) AND NOT y(2));
  END rtl;

-------------------------------------------------------------------
--  Module bsdmult_r2
--  Radix-2 borrow-save digit multiplier
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bsdmult_r2 IS
  PORT (clk : IN std_logic;
        x,y: IN std_logic_vector(1 TO 2);
        z: OUT std_logic_vector(1 TO 2));
END bsdmult_r2;

ARCHITECTURE rtl OF bsdmult_r2 IS
  BEGIN
    z(1) <= (x(1) AND NOT x(2) AND y(1) AND NOT y(2)) OR
            (NOT x(1) AND x(2) AND NOT y(1) AND y(2));
    z(2) <= (x(1) AND NOT x(2) AND NOT y(1) AND y(2)) OR
            (NOT x(1) AND x(2) AND y(1) AND NOT y(2));
  END rtl;

-------------------------------------------------------------------
--  Module dff1d_r2
--  Radix-2 digit-wide D flip-flop
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff1d_r2 IS
  PORT (clk, reset : IN std_logic;
        d: IN std_logic_vector(1 TO 2);
        q: OUT std_logic_vector(1 TO 2));
END dff1d_r2;

ARCHITECTURE rtl OF dff1d_r2 IS
  BEGIN
    PROCESS(clk, reset)
      BEGIN
        IF (reset='1') THEN
          q <= ( OTHERS=>'0' );
        ELSIF (clk'event AND clk='1') THEN
          q <= d;
        END IF;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module latch1d_r2
--  Radix-2 digit-wide latch
```

```
-------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY latch1d_r2 IS
  PORT (clk, reset, l : IN std_logic;
        d: IN std_logic_vector(1 TO 2);
        q: OUT std_logic_vector(1 TO 2));
END latch1d_r2;
ARCHITECTURE rtl OF latch1d_r2 IS
  BEGIN
    PROCESS (clk, reset)
      BEGIN
        IF (reset='1') THEN
          q <= (OTHERS=>'0');
        ELSIF (clk'event AND clk='1') THEN
          IF (l='1') THEN
            q <= d;
          END IF;
        END IF;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module msrec_r2
--  Most significant digit recoder
-------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY msrec_r2 IS
  PORT (clk : IN std_logic;
        pout, mout: IN std_logic_vector(1 TO 2);
        old_res : IN std_logic_vector(1 TO 2);
        new_res : OUT std_logic_vector(1 TO 2));
END msrec_r2;
ARCHITECTURE rtl OF msrec_r2 IS
  SIGNAL cout: std_logic_vector(1 TO 2);
  SIGNAL first_zero, first_one, first_negone: std_logic;
  SIGNAL second_one, second_negone: std_logic;
  BEGIN
    cout(1) <= pout(1) XOR mout(1) XOR pout(2);
    cout(2) <= mout(2);
    first_one <= cout(1) AND NOT cout(2);
    first_negone <= NOT cout(1) AND cout(2);
    first_zero <= NOT(cout(1) XOR cout(2));
    second_one <= old_res(1) AND NOT old_res(2);
    second_negone <= NOT old_res(1) AND old_res(2);
    new_res(1) <= (first_one AND second_negone) OR
                  (first_zero AND second_one);
    new_res(2) <= (first_negone AND second_one) OR
                  (first_zero AND second_negone);
  END rtl;

-------------------------------------------------------------------
--  Module mux21_r2
```

```
--  Radix-2 2-to-1 multiplexer
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux21_r2 IS
  PORT (clk: IN std_logic;
        x0,x1: IN std_logic_vector(1 TO 2);
        sel: in std_logic;
        z: OUT std_logic_vector(1 TO 2));
END mux21_r2;

ARCHITECTURE rtl OF mux21_r2 IS
  BEGIN
    PROCESS(clk)
      BEGIN
        z(1) <= (x0(1) AND NOT sel) OR (x1(1) AND sel);
        z(2) <= (x0(2) AND NOT sel) OR (x1(2) AND sel);
      END PROCESS;
  END rtl;

------------------------------------------------------------------
--  Module norm
--  Normalization of result w/ e-bit exponent
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY norm_r2 IS
  GENERIC (e: natural; ol_delay: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          x: IN std_logic_vector(1 to 2);
          exp_x: IN std_logic_vector(1 to e);
          enable_out: OUT std_logic;
          exp_z: OUT std_logic_vector(1 to e));
END norm_r2;

ARCHITECTURE rtl OF norm_r2 IS

  COMPONENT adde
    GENERIC (e: natural);
      PORT (clk: IN std_logic;
            x,y: IN std_logic_vector(1 TO e);
            z: OUT std_logic_vector(1 TO e));
  END COMPONENT;

  COMPONENT dffm
    GENERIC (m: natural);
      PORT (d, clk, reset: IN std_logic;
            q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT regm
    GENERIC (m: NATURAL);
      PORT (clk, reset: IN std_logic;
            d: IN std_logic_vector(1 to m);
```

```
          q: OUT std_logic_vector(1 to m));
  END COMPONENT;
  COMPONENT latch1
    PORT (clk, reset, l: IN std_logic;
          d: in std_logic;
          q: out std_logic);
  END COMPONENT;
  SIGNAL exp_int, exp_temp, exp_intd: std_logic_vector(1 TO e);
  SIGNAL enable_temp, one1, zero_val, val_temp: std_logic;
  SIGNAL val_in, val_out, norm_in, norm_out, nnorm: std_logic;
  SIGNAL en_reset, en_latch, en_out, en_in: std_logic;
  SIGNAL zero_vec, decr, pos1_vec: std_logic_vector(1 TO e);
  BEGIN
    r1: regm
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, reset=>reset, d=>exp_int, q=>exp_intd);
    e1: adde
      GENERIC MAP (e=>e)
      PORT MAP (clk=>clk, x=>exp_intd, y=>decr, z=>exp_temp);
    d1: dffm
      GENERIC MAP (m=>ol_delay)
      PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>val_temp);
    d2: dff1
      PORT MAP (clk=>clk, reset=>reset, d=>val_in, q=>val_out);
    d3: dff1
      PORT MAP (clk=>clk, reset=>reset, d=>norm_in, q=>norm_out);
    d4: dff1
      PORT MAP (clk=>clk, reset=>en_reset, d=>norm_in,
                        q=>enable_temp);
    d5: latch1
      PORT MAP (clk=>clk, reset=>reset, l=>en_latch, d=>one1,
                q=>en_reset);
    PROCESS(clk)
      BEGIN
        FOR i IN 1 TO e LOOP
          exp_int(i) <= (exp_x(i) AND enable_in) OR
                        (exp_temp(i) AND NOT enable_in);
          decr(i) <= (val_in AND NOT norm_in) OR
                     (pos1_vec(i) AND (norm_in AND NOT val_in));
        END LOOP;
        one1 <= '1';
        en_latch <= reset OR (norm_out AND enable_temp);
        en_in <= norm_out OR en_out;
        zero_val <= NOT(x(1) XOR x(2));
        zero_vec <= (OTHERS=>'0');
        pos1_vec <= zero_vec(1 TO e-1) & '1';
exp_z <= exp_int;
norm_in <= norm_out OR NOT zero_val;
val_in <= val_temp OR val_out;
        nnorm <= NOT norm_in;
        enable_out <= enable_temp;
    END PROCESS;
  END rtl;
```

```
------------------------------------------------------------------
--  Module olAdd_r2
-- Radix-2 on-line floating-point adder
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY oladd_r2 IS
  GENERIC (e: natural; m: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          exp_in0, exp_in1: IN std_logic_vector(e-1 DOWNTO 0);
          man_in0, man_in1: IN std_logic_vector(1 DOWNTO 0);
          enable_out: OUT std_logic;
          exp_out0: OUT std_logic_vector(e-1 DOWNTO 0);
          man_out0: OUT std_logic_vector(1 DOWNTO 0));
END oladd_r2;

ARCHITECTURE rtl OF oladd_r2 IS

  COMPONENT ppm
    PORT (pin1, pin2, min1: IN std_logic;
          pout, mout: OUT std_logic);
  END COMPONENT;

  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1d_r2
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT align_r2
    GENERIC (m: natural; e: natural);
      PORT (clk, reset: IN std_logic;
            en: IN std_logic_vector(1 to e);
            d: IN std_logic_vector(1 to 2);
            q: OUT std_logic_vector(1 to 2));
  END COMPONENT;

  COMPONENT sube
    GENERIC (m: natural);
      PORT (clk: in std_logic;
            x,y: in std_logic_vector(1 to m);
            z: out std_logic_vector(1 to m));
  END COMPONENT;

  COMPONENT swapd_r2
    GENERIC (m: natural);
      PORT (clk: IN std_logic;
            x_in, y_in: IN std_logic_vector(1 TO 2);
            expx, expy: IN std_logic_vector(1 TO e);
```

```
            sel: IN std_logic;
            expx_out: OUT std_logic_vector(1 TO e);
            x_out, y_out: OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT norm_r2
    GENERIC (e: natural; ol_delay: natural);
      PORT (clk, reset, enable_in: IN std_logic;
            x: IN std_logic_vector(1 to 2);
            exp_x: IN std_logic_vector(1 to e);
            enable_out: OUT std_logic;
            exp_z: OUT std_logic_vector(1 to e));
  END COMPONENT;
  SIGNAL tpout, tmout : std_logic_vector(1 TO 2);
  SIGNAL d1out, d2out, d3out: std_logic;
  SIGNAL exp_x, exp_y, exp_temp, del_y: std_logic_vector(1 TO e);
  SIGNAL x,y, xd, yd, z_temp: std_logic_vector(1 TO 2);
  BEGIN
-- exponent calculation
     ye: sube
       GENERIC MAP (m=>e)
       PORT MAP (clk=>clk, x=>exp_in0, y=>exp_in1, z=>del_y);
     sw: swapd_r2
       GENERIC MAP (m=>e)
       PORT MAP (clk=>clk, x_in=>man_in0, y_in=>man_in1,
                 expx=>exp_in0, expy=>exp_in1, sel=>del_y(1),
                 expx_out=>exp_temp, x_out=>x, y_out=>y);
     dx: dff1d_r2
       PORT MAP (clk=>clk, reset=>reset, d=>x, q=>xd);
     ay: align_r2
       GENERIC MAP (m=>m, e=>e)
       PORT MAP (clk=>clk, reset=>reset, en=>del_y, d=>y, q=>yd);
-- significand calculation
     p1: ppm
       PORT MAP (pin1=>xd(1), pin2=>yd(1), min1=>xd(2),
                 pout=>tpout(1), mout=>tmout(1));
     m1: mmp
       PORT MAP (min1=>d2out, min2=>d1out, pin1=>tpout(1),
                 mout=>tmout(2), pout=>tpout(2));
     d1: dff1
       PORT MAP (d=>yd(2), clk=>clk, reset=>reset, q=>d1out);
     d2: dff1
       PORT MAP (d=>tmout(1), clk=>clk, reset=>reset, q=>d2out);
     d3: dff1
       PORT MAP (d=>tpout(2), clk=>clk, reset=>reset, q=>d3out);
     d4: dff1
       PORT MAP (d=>tmout(2), clk=>clk, reset=>reset,
                 q=>z_temp(2));
     d5: dff1
       PORT MAP (d=>d3out, clk=>clk, reset=>reset, q=>z_temp(1));
     dz: dff1d_r2
       PORT MAP (clk=>clk, reset=>reset, d=>z_temp, q=>man_out0);
-- normalization
     n1: norm_r2
```

```
      GENERIC MAP (e=>e, ol_delay=>3)
      PORT MAP (clk=>clk, reset=>reset, enable_in=>enable_in,
                x=>z_temp, exp_x=>exp_temp,
                enable_out=>enable_out, exp_z=>exp_out0);
  END rtl;

-------------------------------------------------------------------
--  Module oldiv_r2
--  Radix-2 on-line floating-point divider
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY oldiv_r2 IS
  GENERIC(e: natural; m: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          exp_in0, exp_in1: IN std_logic_vector(e-1 DOWNTO 0);
          man_in0, man_in1: IN std_logic_vector(1 DOWNTO 0);
          enable_out: OUT std_logic;
          exp_out0: OUT std_logic_vector(e-1 DOWNTO 0);
          man_out0: OUT std_logic_vector(1 DOWNTO 0));
END oldiv_r2;

ARCHITECTURE rtl OF oldiv_r2 IS

  COMPONENT bsadd31_r2
    PORT (clk: IN std_logic;
          w,x,y: IN std_logic_vector(1 TO 2);
          pin, min: IN std_logic_vector(1 TO 2);
          pout, mout: OUT std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT bsdmult_r2
    PORT (clk : IN std_logic;
          x,y: IN std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1d_r2
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dffm
    GENERIC (m: natural);
      PORT (d, clk, reset: IN std_logic;
            q: OUT std_logic);
  END COMPONENT;

  COMPONENT latch1d_r2
    PORT (clk, reset, l : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
```

```
  END COMPONENT;
  COMPONENT msrec_r2
    PORT (clk : IN std_logic;
          pout, mout: IN std_logic_vector(1 TO 2);
          old_res : IN std_logic_vector(1 TO 2);
          new_res : OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT seldiv_r2
    PORT (clk : IN std_logic;
          res: in std_logic_vector(1 to 8);
          q: OUT std_logic_vector(1 to 2));
  END COMPONENT;
  COMPONENT sube
    GENERIC (m: natural);
      PORT (clk: in std_logic;
            x,y: in std_logic_vector(1 to m);
            z: out std_logic_vector(1 to m));
  END COMPONENT;
  COMPONENT norm_r2
    GENERIC (e: natural; ol_delay: natural);
      PORT (clk, reset, enable_in: IN std_logic;
            x: IN std_logic_vector(1 to 2);
            exp_x: IN std_logic_vector(1 to e);
            enable_out: OUT std_logic;
            exp_z: OUT std_logic_vector(1 to e));
  END COMPONENT;
  SIGNAL exp_temp: std_logic_vector(1 TO e);
  SIGNAL x_temp, y_temp, z_int, z_temp: std_logic_vector(1 TO 2);
  SIGNAL pout_d, mout_d: std_logic_vector(1 TO 2);
  SIGNAL y_vector, z_vector: std_logic_vector(1 TO 2*m);
  SIGNAL yzk, zyk: std_logic_vector(1 TO 2*m);
  SIGNAL neg_zyk, neg_yzk: std_logic_vector(1 TO 2*m+8);
  SIGNAL result, res_reg: std_logic_vector(1 TO 2*m+8);
  SIGNAL w_vector: std_logic_vector(1 TO 2*m+10);
  SIGNAL nclk, dely, delz: std_logic;
  SIGNAL yen, zen: std_logic_vector(1 TO m+1);
  SIGNAL pout, mout: std_logic_vector(1 TO 2*m+10);
  BEGIN
-- exponent calculation
    se: sube
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, x=>exp_in0, y=>exp_in1, z=>exp_temp);
-- significand calculation
    xd: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>man_in0, q=>x_temp);
    yd: dffm
      GENERIC MAP (m=>1)
      PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>dely);
    zd: dffm
      GENERIC MAP (m=>5)
      PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>delz);
    pd: dff1d_r2
```

```
      PORT MAP (clk=>clk, reset=>reset, d=>pout(3 TO 4),
                q=>pout_d);
    md: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>mout(3 TO 4),
                q=>mout_d);
    m1: msrec_r2
      PORT MAP (clk=>clk, pout=>pout_d, mout=>mout_d,
                old_res=>res_reg(3 TO 4),
                new_res=>w_vector(3 TO 4));
    s2: seldiv_r2
      PORT MAP (clk=>clk, res=>w_vector(3 TO 10), q=>z_int);
    dz2: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>z_int, q=>z_temp);
    dy: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>man_in1, q=>y_temp);
    dz: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>z_temp, q=>man_out0);

    f1: FOR i IN 1 TO m GENERATE
      d1y: dff1
        PORT MAP (clk=>clk, reset=>reset, d=>yen(i), q=>yen(i+1));
      d1z: dff1
        PORT MAP (clk=>clk, reset=>reset, d=>zen(i), q=>zen(i+1));
      d2: latch1d_r2
        PORT MAP (clk=>clk, reset=>reset, l=>yen(i+1), d=>y_temp,
                  q=>y_vector(2*i-1 TO 2*i));
      d3: latch1d_r2
        PORT MAP (clk=>clk, reset=>reset, l=>zen(i), d=>z_int,
                  q=>z_vector(2*i-1 TO 2*i));
      vd1: bsdmult_r2
        PORT MAP (clk=>clk, x=>y_vector(2*i-1 TO 2*i), y=>z_temp,
                  z=>yzk(2*i-1 TO 2*i));
      vd2: bsdmult_r2
        PORT MAP (clk=>clk, x=>z_vector(2*i-1 TO 2*i), y=>y_temp,
                  z=>zyk(2*i-1 TO 2*i));
    END GENERATE;

    f2: FOR i IN 1 TO m+4 GENERATE
      bs1: bsadd31_r2
        PORT MAP (clk=>clk, w=>w_vector(2*i+1 TO 2*i+2),
                  x=>neg_yzk(2*i-1 TO 2*i),
                  y=>neg_zyk(2*i-1 TO 2*i),
                  pin=>pout(2*i+1 TO 2*i+2),
                  min=>mout(2*i+1 TO 2*i+2),
                  pout=>pout(2*i-1 TO 2*i),
                  mout=>mout(2*i-1 TO 2*i),
                  z=>result(2*i-1 TO 2*i));
      d4: dff1d_r2
        PORT MAP (clk=>clk, reset=>reset, d=>result(2*i-1 TO 2*i),
                  q=>res_reg(2*i-1 TO 2*i));
    END GENERATE;

-- normalization
     n1: norm_r2
       GENERIC MAP (e=>e, ol_delay=>5)
       PORT MAP (clk=>clk, reset=>reset, enable_in=>enable_in,
                 x=>z_temp, exp_x=>exp_temp,
                 enable_out=>enable_out, exp_z=>exp_out0);
```

```
    PROCESS(clk)
      BEGIN
        FOR i IN 1 TO m LOOP
          neg_yzk(2*i-1) <= yzk(2*i);
          neg_yzk(2*i) <= yzk(2*i-1);
          neg_zyk(2*i+7) <= zyk(2*i);
          neg_zyk(2*i+8) <= zyk(2*i-1);
        END LOOP;
        nclk <= NOT clk;
        w_vector(5 TO 2*m+8) <= res_reg(5 TO 2*m+8);
        w_vector(2*m+9 TO 2*m+10) <= "00";
        yen(1) <= dely;
        zen(1) <= delz;
        neg_yzk(2*m+1 TO 2*m+8) <= "00000000";
        neg_zyk(1 TO 8) <= "000000" & x_temp;
        pout(2*m+9 TO 2*m+10) <= "00";
        mout(2*m+9 TO 2*m+10) <= "00";
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module olmult_r2
--  Radix-2 on-line floating-point multiplier
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY olmult_r2 IS
  GENERIC (e: natural; m: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          exp_in0, exp_in1: IN std_logic_vector(1 TO e);
          man_in0, man_in1: IN std_logic_vector(1 TO 2);
          enable_out: OUT std_logic;
          exp_out0: OUT std_logic_vector(1 TO e);
          man_out0: OUT std_logic_vector(1 TO 2));
END olmult_r2;

ARCHITECTURE rtl OF olmult_r2 IS

  COMPONENT bsadd31_r2
    PORT (clk: IN std_logic;
          w,x,y: IN std_logic_vector(1 TO 2);
          pin, min: IN std_logic_vector(1 TO 2);
          pout, mout: OUT std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT bsdmult_r2
    PORT (clk : IN std_logic;
          x,y: IN std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1d_r2
```

```
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT dffm
    GENERIC (m: natural);
      PORT (d, clk, reset: IN std_logic;
            q: OUT std_logic);
  END COMPONENT;
  COMPONENT latch1d_r2
    PORT (clk, reset, l : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT msrec_r2
    PORT (clk : IN std_logic;
          pout, mout: IN std_logic_vector(1 TO 2);
          old_res : IN std_logic_vector(1 TO 2);
          new_res : OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT adde
    GENERIC (m: natural);
      PORT (clk: IN std_logic;
            x,y: IN std_logic_vector(1 to m);
            z: OUT std_logic_vector(1 to m));
  END COMPONENT;
  COMPONENT norm_r2
    GENERIC (e: natural; ol_delay: natural);
      PORT (clk, reset, enable_in: IN std_logic;
            x: IN std_logic_vector(1 to 2);
            exp_x: IN std_logic_vector(1 to e);
            enable_out: OUT std_logic;
            exp_z: OUT std_logic_vector(1 to e));
  END COMPONENT;
  SIGNAL x_temp, y_temp, z_temp: std_logic_vector(1 TO 2);
  SIGNAL x_vector, y_vector: std_logic_vector(1 TO 2*m);
  SIGNAL w_vector, pout, mout: std_logic_vector(1 TO 2*m+8);
  SIGNAL xyk, yxk, result, new_res: std_logic_vector(1 TO 2*m+6);
  SIGNAL delx: std_logic;
  SIGNAL en: std_logic_vector(1 TO m+1);
  SIGNAL exp_temp: std_logic_vector(1 TO e);
  BEGIN
-- exponent calculation
    ae: adde
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, x=>exp_in0, y=>exp_in1, z=>exp_temp);
-- signficand calculation
    dx: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>man_in0, q=>x_temp);
    dy: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>man_in1, q=>y_temp);
    p4: dffm
```

```
      GENERIC MAP (m=>1)
      PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>delx);
    s1: msrec_r2
      PORT MAP (clk=>clk, pout=>pout(1 TO 2), mout=>mout(1 TO 2),
                old_res=>result(1 TO 2), new_res=>new_res(1 TO 2));
    dz: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>z_temp, q=>man_out0);

    f1: FOR i in 1 TO m GENERATE
      d1: dff1
        PORT MAP (clk=>clk, reset=>reset, d=>en(i), q=>en(i+1));
      d2: latch1d_r2
        PORT MAP (clk=>clk, reset=>reset, l=>en(i), d=>man_in0,
                  q=>x_vector(2*i-1 TO 2*i));
      d3: latch1d_r2
        PORT MAP (clk=>clk, reset=>reset, l=>en(i+1), d=>y_temp,
                  q=>y_vector(2*i-1 TO 2*i));
      vd1: bsdmult_r2
        PORT MAP (clk=>clk, x=>x_vector(2*i-1 TO 2*i), y=>y_temp,
                  z=>xyk(2*i+5 TO 2*i+6));
      vd2: bsdmult_r2
        PORT MAP (clk=>clk, x=>y_vector(2*i-1 TO 2*i), y=>x_temp,
                  z=>yxk(2*i+5 TO 2*i+6));
    END GENERATE;

    f2: FOR i IN 1 TO m+3 GENERATE
      bs1: bsadd31_r2
        PORT MAP (clk=>clk, w=>w_vector(2*i+1 TO 2*i+2),
                  x=>xyk(2*i-1 TO 2*i), y=>yxk(2*i-1 TO 2*i),
                  pin=>pout(2*i+1 TO 2*i+2),
                  min=>mout(2*i+1 TO 2*i+2),
                  pout=>pout(2*i-1 TO 2*i),
                  mout=>mout(2*i-1 TO 2*i),
                  z=>result(2*i-1 TO 2*i));
      d4: dff1d_r2
        PORT MAP (clk=>clk, reset=>reset, d=>new_res(2*i-1 TO 2*i),
                  q=>w_vector(2*i-1 TO 2*i));
    END GENERATE;

-- normalization
    n1: norm_r2
      GENERIC MAP (e=>e, ol_delay=>4)
      PORT MAP (clk=>clk, reset=>reset, enable_in=>enable_in,
                x=>z_temp, exp_x=>exp_temp, enable_out=>enable_out,
                exp_z=>exp_out0);

    PROCESS(clk)
      BEGIN
        new_res(3 TO 2*m+6) <= result(3 TO 2*m+6);
        w_vector(2*m+7 TO 2*m+8) <= "00";
        z_temp <= w_vector(1 TO 2);
        xyk(1 TO 6) <= "000000";
        yxk(1 TO 6) <= "000000";
        en(1) <= delx;
        pout(2*m+7 TO 2*m+8) <= "00";
        mout(2*m+7 TO 2*m+8) <= "00";
      END PROCESS;
  END rtl;
```

```
------------------------------------------------------------------
--  Module ol2Sqr
--  Radix-2 on-line floating-point square
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY olsqr_r2 IS
  GENERIC (e: natural; m: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          man_in0: IN std_logic_vector(1 DOWNTO 0);
          exp_in0: IN std_logic_vector(e-1 DOWNTO 0);
          enable_out: OUT std_logic;
          exp_out0: OUT std_logic_vector(e-1 DOWNTO 0);
          man_out0: OUT std_logic_vector(1 DOWNTO 0));
END olsqr_r2;

ARCHITECTURE rtl OF olsqr_r2 IS

  COMPONENT bsadd21_r2
    PORT (clk: IN std_logic;
          x,y: IN std_logic_vector(1 to 2);
          pin, min: IN std_logic;
          pout, mout: OUT std_logic;
          z: OUT std_logic_vector(1 to 2));
  END COMPONENT;

  COMPONENT bsdmult_r2
    PORT (clk : IN std_logic;
          x,y: IN std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1d_r2
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dffm
    GENERIC (m: natural);
      PORT (d, clk, reset: IN std_logic;
            q: OUT std_logic);
  END COMPONENT;

  COMPONENT latch1d_r2
    PORT (clk, reset, l : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT mux21_r2
    PORT( clk : IN std_logic;
          x0,x1: IN std_logic_vector(1 to 2);
          sel: in std_logic;
```

```
        z: OUT std_logic_vector(1 to 2));
  END COMPONENT;
  COMPONENT msrec_r2
    PORT (clk : IN std_logic;
          pout, mout: IN std_logic_vector(1 TO 2);
          old_res : IN std_logic_vector(1 TO 2);
          new_res : OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT mult2e
    GENERIC (m: natural);
      PORT (clk: IN std_logic;
            x: IN std_logic_vector(1 TO m);
            z: OUT std_logic_vector(1 TO m));
  END COMPONENT;
  COMPONENT norm_r2
    GENERIC (e: natural; ol_delay: natural);
      PORT (clk, reset, enable_in: IN std_logic;
            x: IN std_logic_vector(1 to 2);
            exp_x: IN std_logic_vector(1 to e);
            enable_out: OUT std_logic;
            exp_z: OUT std_logic_vector(1 to e));
  END COMPONENT;
  SIGNAL x_temp, z_temp: std_logic_vector(1 TO 2);
  SIGNAL xvec_mux: std_logic_vector(1 TO 2*m+6);
  SIGNAL x_vector: std_logic_vector(1 TO 2*m);
  SIGNAL twice_x: std_logic_vector(1 TO 2*m+2);
  SIGNAL w_vector: std_logic_vector(1 TO 2*m+8);
  SIGNAL xxk, result, new_res: std_logic_vector(1 TO 2*m+6);
  SIGNAL en: std_logic_vector(1 TO m+2);
  SIGNAL pout, mout: std_logic_vector(1 TO m+4);
  SIGNAL tpout, tmout: std_logic_vector(1 TO 2);
  SIGNAL nclk, delx: std_logic;
  SIGNAL exp_temp: std_logic_vector(1 TO e);
  BEGIN
-- exponent calculation
    me: mult2e
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, x=>exp_in0, z=>exp_temp);
-- signficand calculation
    dx: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>man_in0, q=>x_temp);
    m1: msrec_r2
      PORT MAP (clk=>clk, pout=>tpout(1 to 2), mout=>tmout(1 to 2),
                old_res=>result(1 to 2), new_res=>new_res(1 to 2));
    xp: dffm
      GENERIC MAP (m=>1)
      PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>delx);
    dz: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>z_temp, q=>man_out0);
    f1: FOR i IN 1 TO m GENERATE
      d1: dff1
        PORT MAP (clk=>clk, reset=>reset, d=>en(i), q=>en(i+1));
      d2: latch1d_r2
```

```
        PORT MAP (clk=>clk, reset=>reset, l=>en(i+1), d=>x_temp,
                  q=>x_vector(2*i-1 to 2*i));
    end generate;
    f2: FOR i IN 1 TO m+1 GENERATE
      mux1: mux21_r2
        PORT MAP (clk=>clk, x0=>twice_x(2*i-1 to 2*i), x1=>x_temp,
                  sel=>en(i+1), z=>xvec_mux(2*i+3 to 2*i+4));
    END GENERATE;
    f3: FOR i IN 1 TO m+3 GENERATE
      vd1: bsdmult_r2
        PORT MAP (clk=>clk, x=>xvec_mux(2*i-1 to 2*i), y=>x_temp,
                  z=>xxk(2*i-1 to 2*i));
      bs1: bsadd21_r2
        PORT MAP (clk=>clk, x=>w_vector(2*i+1 to 2*i+2),
                  y=>xxk(2*i-1 to 2*i), pin=>pout(i+1),
                  min=>mout(i+1), pout=>pout(i), mout=>mout(i),
                  z=>result(2*i-1 to 2*i));
      d4: dff1d_r2
        PORT MAP (clk=>clk, reset=>reset, d=>new_res(2*i-1 to 2*i),
                  q=>w_vector(2*i-1 to 2*i));
    END GENERATE;
-- normalization
    n1: norm_r2
      GENERIC MAP (e=>e, ol_delay=>4)
      PORT MAP (clk=>clk, reset=>reset, enable_in=>enable_in,
                x=>z_temp, exp_x=>exp_temp, enable_out=>enable_out,
                exp_z=>exp_out0);
    PROCESS(clk)
      BEGIN
        tpout(1 TO 2) <= '0' & pout(1);
        tmout(1 TO 2) <= '0' & mout(1);
        new_res(3 TO 2*m+6) <= result(3 TO 2*m+6);
        w_vector(2*m+7 TO 2*m+8) <= "00";
        z_temp <= w_vector(1 TO 2);
        xvec_mux(1 TO 4) <= "0000";
        twice_x <= x_vector & "00";
        en(1) <= delx;
        pout(m+4) <= '0';
        mout(m+4) <= '0';
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module ol2Sqrt
--  Radix-2 on-line floating-point square root
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY olsqrt_r2 IS
  GENERIC (e: natural; m: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          exp_in0: IN std_logic_vector(e-1 DOWNTO 0);
          man_in0: IN std_logic_vector(1 DOWNTO 0);
          enable_out: OUT std_logic;
          exp_out0: OUT std_logic_vector(e-1 DOWNTO 0);
```

```
        man_out0: OUT std_logic_vector(1 DOWNTO 0));
END olsqrt_r2;

ARCHITECTURE rtl OF olsqrt_r2 IS

  COMPONENT bsadd21_r2
    PORT (clk: IN std_logic;
          x,y: IN std_logic_vector(1 TO 2);
          pin, min: IN std_logic;
          pout, mout: OUT std_logic;
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT bsadd31_r2
    PORT (clk: IN std_logic;
          w,x,y: IN std_logic_vector(1 TO 2);
          pin, min: IN std_logic_vector(1 TO 2);
          pout, mout: OUT std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT bsdmult_r2
    PORT (clk : IN std_logic;
          x,y: IN std_logic_vector(1 TO 2);
          z: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1d_r2
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT dffm
    GENERIC (m: natural);
      PORT (d, clk, reset: IN std_logic;
            q: OUT std_logic);
  END COMPONENT;

  COMPONENT latch1d_r2
    PORT (clk, reset, l : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
  END COMPONENT;

  COMPONENT mux21_r2
    PORT( clk : IN std_logic;
          x0,x1: IN std_logic_vector(1 to 2);
          sel: in std_logic;
          z: OUT std_logic_vector(1 to 2));
  END COMPONENT;

  COMPONENT msrec_r2
    PORT (clk : IN std_logic;
          pout, mout: IN std_logic_vector(1 TO 2);
          old_res : IN std_logic_vector(1 TO 2);
```

```
        new_res : OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT selsqrt_r2
    PORT (clk : IN std_logic;
          res: IN std_logic_vector(1 to 6);
          rad: IN std_logic_vector(1 to 2);
          q: OUT std_logic_vector(1 to 2));
  END COMPONENT;
  COMPONENT div2e
    GENERIC (m: natural);
      PORT (clk: IN std_logic;
            x: IN std_logic_vector(1 TO m);
            z: OUT std_logic_vector(1 TO m));
  END COMPONENT;
  COMPONENT norm_r2
    GENERIC (e: natural; ol_delay: natural);
      PORT (clk, reset, enable_in: IN std_logic;
            x: IN std_logic_vector(1 to 2);
            exp_x: IN std_logic_vector(1 TO e);
            enable_out: OUT std_logic;
            exp_z: OUT std_logic_vector(1 TO e));
  END COMPONENT;
  SIGNAL z_int, z_temp, x_temp: std_logic_vector(1 TO 2);
  SIGNAL pout_d, mout_d, x_sel: std_logic_vector(1 TO 2);
  SIGNAL zvec_mux: std_logic_vector(1 TO 2*m+2);
  SIGNAL z_vector: std_logic_vector(1 TO 2*m);
  SIGNAL twice_z, zzk, neg_zzk: std_logic_vector(1 TO 2*m+2);
  SIGNAL w_vector: std_logic_vector(1 TO 2*m+4);
  SIGNAL result, res_reg : std_logic_vector(1 TO 2*m+2);
  SIGNAL en: std_logic_vector(1 TO m+1);
  SIGNAL pout, mout: std_logic_vector(4 TO m+2);
  SIGNAL tpout, tmout: std_logic_vector(1 TO 8);
  SIGNAL nclk, delz: std_logic;
  SIGNAL x_ext: std_logic_vector(1 TO 6);
  SIGNAL exp_temp: std_logic_vector(1 TO e);
  BEGIN
-- exponent calculation
    de: div2e
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, x=>exp_in0, z=>exp_temp);
-- significand calculation
    dx: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>man_in0, q=>x_temp);
    mx: mux21_r2
      PORT MAP (clk=>clk, x0=>x_temp, x1=>man_in0, sel=>exp_in0(0),
                z=>x_sel);
    pd: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>tpout(3 TO 4),
                q=>pout_d);
    md: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>tmout(3 TO 4),
                q=>mout_d);
```

```
    s1: msrec_r2
      PORT MAP (clk=>clk, pout=>pout_d, mout=>mout_d,
                old_res=>res_reg(3 TO 4),
                new_res=>w_vector(3 TO 4));
    xp: dffm
      GENERIC MAP (m=>3)
      PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>delz);
    dz1: selsqrt_r2
      PORT MAP (clk=>clk, res=>w_vector(3 TO 8),
                rad=>x_sel, q=>z_int);
    dz2: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>z_int, q=>z_temp);

    f1: FOR i in 1 TO m GENERATE
      d1: dff1
        PORT MAP (clk=>clk, reset=>reset, d=>en(i), q=>en(i+1));
      d2: latch1d_r2
        PORT MAP (clk=>clk, reset=>reset, l=>en(i+1), d=>z_temp,
                  q=>z_vector(2*i-1 TO 2*i));
    END GENERATE;

    f2: FOR i in 1 TO m+1 GENERATE
      mux1: mux21_r2
        PORT MAP (clk=>clk, x0=>twice_z(2*i-1 TO 2*i), x1=>z_temp,
                  sel=>en(i), z=>zvec_mux(2*i-1 TO 2*i));
      vd1: bsdmult_r2
        PORT MAP (clk=>clk, x=>zvec_mux(2*i-1 TO 2*i), y=>z_temp,
                  z=>zzk(2*i-1 TO 2*i));
    END GENERATE;

    f3: FOR i in 1 TO 3 GENERATE
      bs1: bsadd31_r2
        PORT MAP (clk=>clk, w=>w_vector(2*i+1 TO 2*i+2),
                  x=>x_ext(2*i-1 TO 2*i), y=>zzk(2*i-1 TO 2*i),
                  pin=>tpout(2*i+1 TO 2*i+2),
                  min=>tmout(2*i+1 TO 2*i+2),
                  pout=>tpout(2*i-1 TO 2*i),
                  mout=>tmout(2*i-1 TO 2*i),
                  z=>result(2*i-1 TO 2*i));
    END GENERATE;

    f4: FOR i in 4 TO m+1 GENERATE
      bs2: bsadd21_r2
        PORT MAP (clk=>clk, x=>w_vector(2*i+1 TO 2*i+2),
                  y=>zzk(2*i-1 TO 2*i), pin=>pout(i+1),
                  min=>mout(i+1), pout=>pout(i), mout=>mout(i),
                  z=>result(2*i-1 TO 2*i));
    END GENERATE;

    f5: FOR i in 1 TO m+1 GENERATE
      d3: dff1d_r2
        PORT MAP (clk=>clk, reset=>reset, d=>result(2*i-1 TO 2*i),
                  q=>res_reg(2*i-1 TO 2*i));
    END GENERATE;

-- normalization
    n1: norm_r2
      GENERIC MAP (e=>e, ol_delay=>5)
      PORT MAP (clk=>clk, reset=>reset, enable_in=>enable_in,
                x=>z_temp, exp_x=>exp_temp, enable_out=>enable_out,
                exp_z=>exp_out0);
```

```
    PROCESS(clk)
      BEGIN
        FOR i IN 1 TO m LOOP
          neg_zzk(2*i-1) <= zzk(2*i);
          neg_zzk(2*i) <= zzk(2*i-1);
        END LOOP;
        nclk <= NOT clk;
        x_ext <= "0000" & x_sel;
        tpout(7 TO 8) <= '0' & pout(4);
        tmout(7 TO 8) <= '0' & mout(4);
        twice_z <= z_vector & "00";
        w_vector(5 TO 2*m+2) <= res_reg(5 TO 2*m+2);
        w_vector(2*m+3 TO 2*m+4) <= "00";
        en(1) <= delz;
        pout(m+2) <= '0';
        mout(m+2) <= '0';
        man_out0 <= z_temp;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module olsub_r2
 -- Radix-2 on-line floating-point subtracter
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY olsub_r2 IS
  GENERIC (e: natural; m: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          exp_in0, exp_in1: IN std_logic_vector(e-1 DOWNTO 0);
          man_in0, man_in1: IN std_logic_vector(1 DOWNTO 0);
          enable_out: OUT std_logic;
          exp_out0: OUT std_logic_vector(e-1 DOWNTO 0);
          man_out0: OUT std_logic_vector(1 DOWNTO 0));
END olsub_r2;

ARCHITECTURE rtl OF olsub_r2 IS

  COMPONENT ppm
    PORT (pin1, pin2, min1: IN std_logic;
          pout, mout: OUT std_logic);
  END COMPONENT;

  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1d_r2
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 2);
          q: OUT std_logic_vector(1 TO 2));
```

```
  END COMPONENT;
  COMPONENT align_r2
    GENERIC (m: natural; e: natural);
      PORT (clk, reset: IN std_logic;
            en: IN std_logic_vector(1 to e);
            d: IN std_logic_vector(1 to 2);
            q: OUT std_logic_vector(1 to 2));
  END COMPONENT;
  COMPONENT sube
    GENERIC (m: natural);
      PORT (clk: in std_logic;
            x,y: in std_logic_vector(1 to m);
            z: out std_logic_vector(1 to m));
  END COMPONENT;
  COMPONENT swapd_r2
    GENERIC (m: natural);
      PORT (clk: IN std_logic;
            x_in, y_in: IN std_logic_vector(1 TO 2);
            expx, expy: IN std_logic_vector(1 TO e);
            sel: IN std_logic;
            expx_out: OUT std_logic_vector(1 TO e);
            x_out, y_out: OUT std_logic_vector(1 TO 2));
  END COMPONENT;
  COMPONENT norm_r2
    GENERIC (e: natural; ol_delay: natural);
      PORT (clk, reset, enable_in: IN std_logic;
            x: IN std_logic_vector(1 to 2);
            exp_x: IN std_logic_vector(1 to e);
            enable_out: OUT std_logic;
            exp_z: OUT std_logic_vector(1 to e));
  END COMPONENT;
  SIGNAL tpout, tmout : std_logic_vector(1 TO 2);
  SIGNAL d1out, d2out, d3out: std_logic;
  SIGNAL exp_x, exp_y, exp_temp, del_y: std_logic_vector(1 TO e);
  SIGNAL x,y, xd, yd, z_temp: std_logic_vector(1 TO 2);
  SIGNAL not_man_in1: std_logic_vector(1 TO 2);
  BEGIN
-- exponent calculation
    ye: sube
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, x=>exp_in0, y=>exp_in1, z=>del_y);
    sw: swapd_r2
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, x_in=>man_in0, y_in=>not_man_in1,
                expx=>exp_in0, expy=>exp_in1, sel=>del_y(1),
                expx_out=>exp_temp, x_out=>x, y_out=>y);
    dx: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>x, q=>xd);
    ay: align_r2
      GENERIC MAP (m=>m, e=>e)
      PORT MAP (clk=>clk, reset=>reset, en=>del_y, d=>y, q=>yd);
```

```
-- significand calculation
    p1: ppm
      PORT MAP (pin1=>xd(1), pin2=>yd(1), min1=>xd(2),
                pout=>tpout(1), mout=>tmout(1));
    m1: mmp
      PORT MAP (min1=>d2out, min2=>d1out, pin1=>tpout(1),
                mout=>tmout(2), pout=>tpout(2));
    d1: dff1
      PORT MAP (d=>yd(2), clk=>clk, reset=>reset, q=>d1out);
    d2: dff1
      PORT MAP (d=>tmout(1), clk=>clk, reset=>reset, q=>d2out);
    d3: dff1
      PORT MAP (d=>tpout(2), clk=>clk, reset=>reset, q=>d3out);
    d4: dff1
      PORT MAP (d=>tmout(2), clk=>clk, reset=>reset, q=>z_temp(2));
    d5: dff1
      PORT MAP (d=>d3out, clk=>clk, reset=>reset, q=>z_temp(1));
    dz: dff1d_r2
      PORT MAP (clk=>clk, reset=>reset, d=>z_temp, q=>man_out0);
-- normalization
    n1: norm_r2
      GENERIC MAP (e=>e, ol_delay=>3)
      PORT MAP (clk=>clk, reset=>reset, enable_in=>enable_in,
                x=>z_temp, exp_x=>exp_temp, enable_out=>enable_out,
                exp_z=>exp_out0);

    not_man_in1 <= NOT man_in1(1) & NOT man_in1(0);

  END rtl;

-------------------------------------------------------------------
--  Module seldiv_r2
--  Radix-2 borrow-save division selection function
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY seldiv_r2 IS
  PORT (clk : IN std_logic;
        res: IN std_logic_vector(1 TO 8);
        q: OUT std_logic_vector(1 TO 2));
END seldiv_r2;

ARCHITECTURE rtl OF seldiv_r2 IS

  COMPONENT bsdcomp_r2
    PORT (x,y: IN std_logic_vector(1 TO 2);
          gg_in, g_in, e_in, s_in, ss_in: IN std_logic;
          gg_out, g_out, e_out, s_out, ss_out: OUT std_logic);
  END COMPONENT;

  SIGNAL neg_1, pos_1: std_logic_vector(1 to 8);
  SIGNAL ngg, ng, ne, ns, nss, pgg, pg, pe, ps,
         pss: std_logic_vector(0 to 4);

  BEGIN
    f1: FOR i IN 1 TO 4 GENERATE
      neg1: bsdcomp_r2
        PORT MAP (x=>res(2*i-1 to 2*i), y=>neg_1(2*i-1 to 2*i),
                  gg_in=>ngg(i-1), g_in=>ng(i-1), e_in=>ne(i-1),
```

```
                s_in=>ns(i-1), ss_in=>nss(i-1), gg_out=>ngg(i),
                g_out=>ng(i), e_out=>ne(i), s_out=>ns(i),
                ss_out=>nss(i));
    pos1: bsdcomp_r2
      PORT MAP (x=>res(2*i-1 to 2*i), y=>pos_1(2*i-1 to 2*i),
                gg_in=>pgg(i-1), g_in=>pg(i-1), e_in=>pe(i-1),
                s_in=>ps(i-1), ss_in=>pss(i-1), gg_out=>pgg(i),
                g_out=>pg(i), e_out=>pe(i), s_out=>ps(i),
                ss_out=>pss(i));
  END GENERATE;

  PROCESS(clk)
    BEGIN
      ngg(0) <= '0';
      ng(0) <= '0';
      ne(0) <= '1';
      ns(0) <= '0';
      nss(0) <= '0';
      pgg(0) <= '0';
      pg(0) <= '0';
      pe(0) <= '1';
      ps(0) <= '0';
      pss(0) <= '0';
      pos_1(1 to 8) <= "00001010"; -- q=1 if W[k] >= 3/16
      neg_1(1 to 8) <= "00010000"; -- q=-1 if W[k] < -1/4
      q(1) <= pgg(4) OR pg(4) OR pe(4);
      q(2) <= nss(4) OR ns(4);
    END PROCESS;
END rtl;

------------------------------------------------------------------
--  Module selsqrt_r2
--  Radix-2 borrow-save square root selection function
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY selsqrt_r2 is
  PORT (clk : IN std_logic;
        res: IN std_logic_vector(1 TO 6);
        rad: IN std_logic_vector(1 TO 2);
        q: OUT std_logic_vector(1 TO 2));
END selsqrt_r2;

ARCHITECTURE rtl OF selsqrt_r2 IS

  COMPONENT bsdcomp_r2
    PORT (x,y: IN std_logic_vector(1 TO 2);
          gg_in, g_in, e_in, s_in, ss_in: IN std_logic;
          gg_out, g_out, e_out, s_out, ss_out: OUT std_logic);
  END COMPONENT;

  SIGNAL neg_1, pos_1: std_logic_vector(1 TO 6);
  SIGNAL ngg, ng, ne, ns, nss, pgg, pg, pe, ps,
         pss: std_logic_vector(0 TO 3);

  BEGIN
    f1: FOR i IN 1 TO 3 GENERATE
```

```
      neg1: bsdcomp_r2
        PORT MAP (x=>res(2*i-1 to 2*i), y=>neg_1(2*i-1 to 2*i),
                  gg_in=>ngg(i-1), g_in=>ng(i-1), e_in=>ne(i-1),
                  s_in=>ns(i-1), ss_in=>nss(i-1), gg_out=>ngg(i),
                  g_out=>ng(i), e_out=>ne(i), s_out=>ns(i),
                  ss_out=>nss(i));
      pos1: bsdcomp_r2
        PORT MAP (x=>res(2*i-1 to 2*i), y=>pos_1(2*i-1 to 2*i),
                 gg_in=>pgg(i-1), g_in=>pg(i-1), e_in=>pe(i-1),
                 s_in=>ps(i-1), ss_in=>pss(i-1), gg_out=>pgg(i),
                 g_out=>pg(i), e_out=>pe(i), s_out=>ps(i),
                 ss_out=>pss(i));
    END GENERATE;

    PROCESS(clk)
      BEGIN
        ngg(0) <= '0';
        ng(0) <= '0';
        ne(0) <= '1';
        ns(0) <= '0';
        nss(0) <= '0';
        pgg(0) <= '0';
        pg(0) <= '0';
        pe(0) <= '1';
        ps(0) <= '0';
        pss(0) <= '0';
        pos_1(1 to 6) <= "00" & (NOT rad(1) OR rad(2)) &
                         '0' & (rad(1) XOR rad(2)) & '0';
        -- q=1 if W[k] >= (2-x)/8
        neg_1(1 to 6) <= "000" & (rad(1) OR NOT rad(2)) &
                         '0' & (rad(1) XOR rad(2));
        -- q=-1 if W[k] < (-2-x)/8
        q(1) <= pgg(3) OR pg(3) OR pe(3);
        q(2) <= nss(3) OR ns(3);
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module swapd_r2
--  Swaps two Radix-2 digits
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY swapd_r2 IS
  GENERIC (e: natural);
    PORT (clk: IN std_logic;
          x_in, y_in: IN std_logic_vector(1 TO 2);
          expx, expy: IN std_logic_vector(1 TO e);
          sel: IN std_logic;
          expx_out: OUT std_logic_vector(1 TO e);
          x_out, y_out: OUT std_logic_vector(1 TO 2));
END swapd_r2;

ARCHITECTURE rtl OF swapd_r2 IS
  BEGIN
```

```
  PROCESS(clk)
    BEGIN
      IF (sel='0') THEN
        x_out <= x_in;
        y_out <= y_in;
        expx_out <= expx;
      ELSE
        x_out <= y_in;
        y_out <= x_in;
        expx_out <= expy;
      END IF;
    END PROCESS;
END rtl;
```

B.3 $RCNS_{2j,3}$ On-line Floating-point Library Modules

```
-------------------------------------------------------------------
--  Module align_r2j
--  Radix 2j operand alignment module
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY align_r2j IS
  GENERIC (m: natural; e: natural);
    PORT (clk, reset: IN std_logic;
          en: IN std_logic_vector(1 TO e);
          d: IN std_logic_vector(1 TO 4);
          q: OUT std_logic_vector(1 TO 4));
END align_r2j;

ARCHITECTURE rtl OF align_r2j IS

  COMPONENT decoder_256
    PORT (clk: IN std_logic;
          x: IN std_logic_vector(1 TO 8);
          z: OUT std_logic_vector(0 TO 255));
  END COMPONENT;

  COMPONENT mux21_r2j
    PORT (clk: IN std_logic;
          x0,x1: IN std_logic_vector(1 TO 4);
          sel: IN std_logic;
          z: OUT std_logic_vector(1 TO 4));
  END COMPONENT;

  COMPONENT dff1d_r2j
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 4);
          q: OUT std_logic_vector(1 TO 4));
  END COMPONENT;

  SIGNAL temp_q: std_logic_vector(1 TO 4*m+4);
  SIGNAL mux_in: std_logic_vector(1 TO 4*m);
  SIGNAL dec_en: std_logic_vector(0 TO 255);
  SIGNAL zero_en: std_logic_vector(1 to 8-e);
  SIGNAL en_ext: std_logic_vector(1 to 8);

  BEGIN
    d1: decoder_256
      PORT MAP (clk=>clk, x=>en_ext, z=>dec_en);
    f1: FOR i IN 0 TO m-1 GENERATE
      m1: mux21_r2j
        PORT MAP (clk=>clk, x0=>temp_q(4*i+5 to 4*i+8),
                  x1=>d, sel=>dec_en(i),
                  z=>mux_in(4*i+1 to 4*i+4));
      d2: dff1d_r2j
        PORT MAP (clk=>clk, reset=>reset,
                  d=>mux_in(4*i+1 to 4*i+4),
                  q=>temp_q(4*i+1 to 4*i+4));
    END GENERATE;

    zero_en <= (others=>'0');
    en_ext <= zero_en & en;
```

```
    temp_q(4*m+1 to 4*m+4) <= "0000";
    q <= temp_q(1 to 4);
  END rtl;

-------------------------------------------------------------------
--  Module bsadd21_r2j
--  Radix 2j borrow-save digit 2-to-1 adder
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bsadd21_r2j IS
  PORT (clk: IN std_logic;
        x,y : IN std_logic_vector(1 TO 4);
        pin, min : IN std_logic;
        pout, mout : OUT std_logic;
        z : OUT std_logic_vector(1 TO 4));
END bsadd21_r2j;

ARCHITECTURE rtl OF bsadd21_r2j IS

  COMPONENT ppm
    PORT (pin1, pin2, min1: IN std_logic;
          pout, mout: OUT std_logic);
  END COMPONENT;

  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;

  SIGNAL tpout, tmout : std_logic_vector(1 to 4);

  BEGIN
    p1: ppm
      PORT MAP (pin1=>x(1), pin2=>y(1), min1=>x(2),
                pout=>tpout(1), mout=>tmout(1));
    p2: ppm
      PORT MAP (pin1=>x(3), pin2=>y(3), min1=>x(4),
                pout=>tpout(2), mout=>tmout(2));
    m1: mmp
      PORT MAP (min1=>tmout(1), min2=>y(2), pin1=>tpout(2),
                mout=>tmout(3), pout=>tpout(3));
    m2: mmp
      PORT MAP (min1=>tmout(2), min2=>y(4), pin1=>pin,
                mout=>tmout(4), pout=>tpout(4));

    PROCESS(clk)
      BEGIN
        pout <= NOT tpout(1);
        mout <= NOT tmout(3);
        z <= tpout(3) & tmout(4) & tpout(4) & min;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module bsadd31_r2j
--  Radix 2j borrow-save 3-digit adder
-------------------------------------------------------------------
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bsadd31_r2j IS
  PORT (clk: IN std_logic;
        w,x,y : IN std_logic_vector(1 TO 4);
        pin, min : IN std_logic_vector(1 TO 2);
        pout, mout : OUT std_logic_vector(1 TO 2);
        z : OUT std_logic_vector(1 TO 4));
END bsadd31_r2j;

ARCHITECTURE rtl OF bsadd31_r2j IS

  COMPONENT ppm
    PORT (pin1, pin2, min1: IN std_logic;
          pout, mout: OUT std_logic);
  END COMPONENT;

  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;

  SIGNAL tpout, tmout: std_logic_vector(1 to 8);

  BEGIN
    p1: ppm
      PORT MAP (pin1=>w(1), pin2=>x(1), min1=>w(2),
                pout=>tpout(1), mout=>tmout(1));
    m1: mmp
      PORT MAP (min1=>x(2), min2=>y(2), pin1=>y(1),
                mout=>tmout(2), pout=>tpout(2));
    p2: ppm
      PORT MAP (pin1=>w(3), pin2=>x(3), min1=>w(4),
                pout=>tpout(3), mout=>tmout(3));
    m2: mmp
      PORT MAP (min1=>x(4), min2=>y(4), pin1=>y(3),
                mout=>tmout(4), pout=>tpout(4));
    p3: ppm
      PORT MAP (pin1=>tpout(2), pin2=>tpout(3), min1=>tmout(1),
                pout=>tpout(5), mout=>tmout(5));
    p4: ppm
      PORT MAP (pin1=>tpout(4), pin2=>pin(1), min1=>tmout(3),
                pout=>tpout(6), mout=>tmout(6));
    m3: mmp
      PORT MAP (min1=>tmout(5), min2=>tmout(4), pin1=>tpout(6),
                mout=>tmout(7), pout=>tpout(7));
    m4: mmp
      PORT MAP (min1=>tmout(6), min2=>min(1), pin1=>pin(2),
                mout=>tmout(8), pout=>tpout(8));

    pout(1) <= NOT tpout(1);
    mout(1) <= NOT tmout(2);
    pout(2) <= NOT tpout(5);
    mout(2) <= NOT tmout(7);
    z(1) <= tpout(7);
    z(2) <= tmout(8);
    z(3) <= tpout(8);
```

```
    z(4) <= min(2);
  END rtl;

-------------------------------------------------------------------
--  Module bsdcomp_r2j
--  Radix 2j borrow-save digit comparator
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bsdcomp_r2j IS
  PORT (x,y: IN std_logic_vector(1 TO 4);
        gg_in, g_in, e_in, s_in, ss_in: IN std_logic;
        gg_out, g_out, e_out, s_out, ss_out: OUT std_logic);
END bsdcomp_r2j;

ARCHITECTURE rtl OF bsdcomp_r2j IS

  SIGNAL xp3, xp2, xp1, x0, xn1, xn2, xn3: std_logic;
  SIGNAL yp3, yp2, yp1, y0, yn1, yn2, yn3: std_logic;

  BEGIN
    xp3 <= x(1) AND NOT x(2) AND x(3) AND NOT x(4);
    xp2 <= x(1) AND NOT x(2) AND NOT (x(3) XOR x(4));
    xp1 <= (x(3) AND NOT x(4) AND NOT (x(1) XOR x(2))) OR
           (x(1) AND NOT x(2) AND NOT x(3) AND x(4));
    x0 <= NOT (x(1) XOR x(2)) AND NOT (x(3) XOR x(4));
    xn1 <= (NOT x(3) AND x(4) AND NOT(x(1) XOR x(2))) OR
           (NOT x(1) AND x(2) AND x(3) AND NOT x(4));
    xn2 <= NOT x(1) AND x(2) AND NOT (x(3) XOR x(4));
    xn3 <= NOT x(1) AND x(2) AND NOT x(3) AND x(4);

    yp3 <= y(1) AND NOT y(2) AND y(3) AND NOT y(4);
    yp2 <= y(1) AND NOT y(2) AND NOT (y(3) XOR y(4));
    yp1 <= (y(3) AND NOT y(4) AND NOT (y(1) XOR y(2))) OR
           (y(1) AND NOT y(2) AND NOT y(3) AND y(4));
    y0 <= NOT (y(1) XOR y(2)) AND NOT (y(3) XOR y(4));
    yn1 <= (NOT y(3) AND y(4) AND NOT(y(1) XOR y(2))) OR
           (NOT y(1) AND y(2) AND y(3) AND NOT y(4));
    yn2 <= NOT y(1) AND y(2) AND NOT (y(3) XOR y(4));
    yn3 <= NOT y(1) AND y(2) AND NOT y(3) AND y(4);

    gg_out <= gg_in OR ((e_in OR g_in) AND
              ((xp3 AND NOT (yp3 OR yp2)) OR
              (xp2 AND NOT (yp3 OR yp2 OR yp1)) OR
              (xp1 AND (yn1 OR yn2 OR yn3)) OR
              (x0 AND (yn2 OR yn3)) OR
              (xn1 AND yn3)));

    ss_out <= ss_in OR ((e_in OR g_in) AND
              ((xn3 AND NOT (yn3 OR yn2)) OR
              (xn2 AND NOT (yn3 OR yn2 OR yn1)) OR
              (xn1 AND (yp1 OR yp2 OR yp3)) OR
              (x0 AND (yp2 OR yp3)) OR
              (xp1 AND yp3)));

    g_out <= (g_in OR e_in) AND ((xp3 AND yp2) OR (xp2 AND yp1)
             OR (xp1 AND y0) OR (x0 AND yn1) OR (xn1 AND yn2)
```

```
            OR (xn2 AND yn3));
    s_out <= (s_in OR e_in) AND ((xp2 AND yp3) OR (xp1 AND yp2)
            OR (x0 AND yp1) OR (xn1 AND y0) OR (xn2 AND yn1)
            OR (xn3 AND yn2));
    e_out <= e_in AND ((xp3 AND yp3) OR (xp2 AND yp2) OR
            (xp1 AND yp1) OR (x0 AND y0) OR (xn1 AND yn1)
            OR (xn2 AND yn2) OR (xn3 AND yn3));
  END rtl;

-------------------------------------------------------------------
--  Module bsdmult_r2j
--  Radix 2j borrow-save digit multiplier
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY bsdmult_r2j IS
  PORT (clk: IN std_logic;
        x,y : IN std_logic_vector(1 TO 4);
        x_d2 : IN std_logic_vector(1 TO 2);
        z : OUT std_logic_vector(1 TO 4));
END bsdmult_r2j;

ARCHITECTURE rtl OF bsdmult_r2j IS
  BEGIN
    z(1) <= (x(3) AND y(1)) OR (NOT x(3) AND y(2)) OR
            (x(1) AND y(3)) OR (NOT x(1) AND y(4));
    z(2) <= (x(4) AND y(1)) OR (NOT x(4) AND y(2)) OR
            (x(2) AND y(3)) OR (NOT x(2) AND y(4));
    z(3) <= (NOT x_d2(1) AND y(1)) OR (x_d2(1) AND y(2)) OR
            (x(3) AND y(3)) OR (NOT x(3) AND y(4));
    z(4) <= (NOT x_d2(2) AND y(1)) OR (x_d2(2) AND y(2)) OR
            (x(4) AND y(3)) OR (NOT x(4) AND y(4));
  END rtl;

-------------------------------------------------------------------
--  Module dff1d_r2j
--  Radix 2j digit-wide D flip-flop
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff1d_r2j IS
  PORT (clk, reset: IN std_logic;
        d: IN std_logic_vector(1 TO 4);
        q: OUT std_logic_vector(1 TO 4));
END dff1d_r2j;

ARCHITECTURE rtl OF dff1d_r2j IS
  BEGIN
    PROCESS(clk)
      BEGIN
        IF(reset='1') THEN
          q <= (OTHERS=>'0');
        ELSIF(clk'event AND clk='1') THEN
          q <= d;
        END IF;
```

```
    END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module dffmd_r2j
--  Series of Radix 2j m digit-wide D flip-flops
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dffmd_r2j IS
  GENERIC (m: natural);
    PORT (clk, reset: IN std_logic;
          d: IN std_logic_vector(1 TO 4);
          q: OUT std_logic_vector(1 TO 4));
END dffmd_r2j;

ARCHITECTURE rtl OF dffmd_r2j IS

  COMPONENT dff1d_r2j
    PORT (clk, reset : IN std_logic;
          d: IN std_logic_vector(1 TO 4);
          q: OUT std_logic_vector(1 TO 4));
  END COMPONENT;

  SIGNAL temp_q: std_logic_vector(1 TO 4*m+4);

  BEGIN
    f1: FOR i IN 1 TO m GENERATE
      d1: dff1d_r2j
        PORT MAP (clk=>clk, reset=>reset, d=>temp_q(4*i-3 to 4*i),
                  q=>temp_q(4*i+1 to 4*i+4));
    END GENERATE;

    PROCESS(clk)
      BEGIN
        temp_q(1 to 4) <= d;
        q <= temp_q(4*m+1 TO 4*m+4);
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module dsrec_r2j
--  Radix 2j digit-set converter from {-3,...,3} to {-2,...,2}
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dsrec_r2j IS
  PORT (clk, reset: IN std_logic;
        x : IN std_logic_vector(1 TO 4);
        z : OUT std_logic_vector(1 TO 4));
END dsrec_r2j;

ARCHITECTURE rtl OF dsrec_r2j IS

  COMPONENT dffmd_r2j
    GENERIC (m: natural);
      PORT (clk, reset: IN std_logic;
            d: IN std_logic_vector(1 TO 4);
            q: OUT std_logic_vector(1 TO 4));
```

```
  END COMPONENT;

  SIGNAL t: std_logic_vector(1 TO 2);
  SIGNAL nr, x2_d, w, w2_d, z_temp: std_logic_vector(1 TO 4);
  SIGNAL t_zero, w_zero, nclk: std_logic;

  BEGIN
    x2: dffmd_r2j
      GENERIC MAP (m=>2)
      PORT MAP (clk=>clk, reset=>reset, d=>nr, q=>x2_d);
    w2: dffmd_r2j
      GENERIC MAP (m=>2)
      PORT MAP (clk=>clk, reset=>reset, d=>w, q=>w2_d);

    PROCESS(clk)
      BEGIN
--convert to non-redundant format
        nr(1) <= x(1) AND NOT x(2) AND (x(3) OR NOT x(4));
        nr(2) <= NOT x(1) AND x(2) AND (NOT x(3) OR x(4));
        nr(3) <= (x(3) AND NOT x(4) AND (x(1) OR NOT x(2))) OR
                 (x(1) AND NOT x(2) AND NOT x(3) AND x(4));
        nr(4) <= (NOT x(3) AND x(4) AND (NOT x(1) OR x(2))) OR
                 (NOT x(1) AND x(2) AND x(3) AND NOT x(4));
--compute t(k-2)
        t(1) <= x2_d(2) AND (x2_d(4) OR nr(1));
        t(2) <= x2_d(1) AND (x2_d(3) OR nr(2));
        t_zero <= NOT(t(1) OR t(2));
--compute w(k)
        w(1) <= (x2_d(1) AND NOT x2_d(3) AND NOT x0_d(2)) OR
                (x2_d(2) AND NOT x2_d(4) AND x0_d(1));
        w(2) <= (x2_d(1) AND NOT x2_d(3) AND x0_d(2)) OR
                (x2_d(2) AND NOT x2_d(4) AND NOT x0_d(1));
        w(3) <= (x2_d(3) AND NOT x2_d(1)) OR (x2_d(2) AND x2_d(4));
        w(4) <= (x2_d(4) AND NOT x2_d(2)) OR (x2_d(1) AND x2_d(3));
        w_zero <= NOT (w2_d(1) OR w2_d(2) OR w2_d(3) OR w2_d(4));
--compute z(k-2)=t(k-2)+w(k-2)
        z <= (w2_d(1) AND t_zero) OR (w2_d(3) AND t(1));
        z <= (w2_d(4) AND t(2)) OR (w2_d(2) AND t_zero);
        z <= (w2_d(1) AND t(2)) OR (w2_d(3) AND t_zero) OR
             (w_zero AND t(1));
        z <= (w_zero AND t(2)) OR (w2_d(4) AND t_zero) OR
             (w2_d(2) AND t(1));
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module latch1d_r2j
--  Radix 2j digit-wide latch
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY latch1d_r2j IS
  PORT (clk, reset, l: IN std_logic;
        d : IN std_logic_vector(1 to 4);
        q : OUT std_logic_vector(1 to 4));
END latch1d_r2j;

ARCHITECTURE rtl OF latch1d_r2j IS
```

```
  BEGIN
    PROCESS(clk)
      BEGIN
        IF (reset='1') THEN
          q <= (OTHERS=>'0');
ELSIF (clk='1' AND clk'event) THEN
          IF (l='1') THEN
            q <= d;
          END IF;
        END IF;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module msrec_2j
--  Radix 2j most-significant-digit recoder
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY msrec_r2j IS
  PORT (clk: IN std_logic;
        pout, mout: IN std_logic_vector(1 TO 2);
        old_res: IN std_logic_vector(1 TO 4);
        new_res: OUT std_logic_vector(1 TO 4));
END msrec_r2j;

ARCHITECTURE rtl OF msrec_r2j IS

  COMPONENT mmp
    PORT (min1, min2, pin1: IN std_logic;
          mout, pout: OUT std_logic);
  END COMPONENT;

  SIGNAL first_zero, first_one, first_negone, second_two,
         second_negtwo, second_one, second_negone,
         second_three, second_negthree: std_logic;
  SIGNAL cout: std_logic_vector(1 to 4);

  BEGIN
    c1: mmp
      PORT MAP (min1=>pout(1), min2=>mout(1), pin1=>pout(2),
                mout=>cout(2), pout=>cout(3));

    PROCESS(clk)
      BEGIN
        cout(1) <= pout(1);
        cout(4) <= mout(2);
        first_one <= (NOT(cout(1) XOR cout(2)) AND cout(3)
                      AND NOT cout(4)) OR (cout(1) AND
                      NOT cout(2) AND NOT cout(3) AND cout(4));
        first_negone <= (NOT(cout(1) XOR cout(2)) AND NOT cout(3)
                         AND cout(4)) OR (NOT cout(1) AND cout(2)
                         AND cout(3) AND NOT cout(4));
        first_zero <= NOT(cout(1) XOR cout(2)) AND
                      NOT(cout(3) XOR cout(4));
second_two <= old_res(1) AND NOT old_res(2) AND
                      NOT (old_res(3) XOR old_res(4));
second_negtwo <= NOT old_res(1) AND old_res(2) AND
                         NOT (old_res(3) XOR old_res(4));
```

```
        second_one <= (NOT(old_res(1) XOR old_res(2)) AND
                      old_res(3) AND NOT old_res(4)) OR
                      (old_res(1) AND NOT old_res(2) AND
                      NOT old_res(3) AND old_res(4));
second_negone <= (NOT(old_res(1) XOR old_res(2)) AND
                         NOT old_res(3) AND old_res(4)) OR
                         (NOT old_res(1) AND old_res(2) AND
                         old_res(3) AND NOT old_res(4));
        second_three <= old_res(1) AND NOT old_res(2) AND
                        old_res(3) AND NOT old_res(4);
        second_negthree <= NOT old_res(1) AND old_res(2) AND
                           NOT old_res(3) AND old_res(4);
new_res(1) <= (first_negone AND (second_negtwo OR
                      second_negone)) OR (first_zero AND
                      old_res(1));
new_res(2) <= (first_one AND (second_two OR second_one))
                      OR (first_zero AND old_res(2));
new_res(3) <= (first_negone AND (second_negthree OR
                      second_negone)) OR (first_zero AND
                      old_res(3));
new_res(4) <= (first_one AND (second_three OR second_one))
                      OR (first_zero AND old_res(4));
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module mux21_r2j
--  Radix 2j 2-to-1 multiplexer
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux21_r2j IS
  PORT (clk: IN std_logic;
        x0,x1: IN std_logic_vector(1 TO 4);
        sel: IN std_logic;
        z: OUT std_logic_vector(1 TO 4));
END mux21_r2j;

ARCHITECTURE rtl OF mux21_r2j IS
  BEGIN
    PROCESS(clk)
      BEGIN
        z(1) <= (x0(1) AND NOT sel) OR (x1(1) AND sel);
        z(2) <= (x0(2) AND NOT sel) OR (x1(2) AND sel);
        z(3) <= (x0(3) AND NOT sel) OR (x1(3) AND sel);
        z(4) <= (x0(4) AND NOT sel) OR (x1(4) AND sel);
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module mux31_r2j
--  Radix 2j 3-to-1 multiplexer
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux31_r2j IS
```

```
  PORT (clk: IN std_logic;
        x0, x1, x2: IN std_logic_vector(1 TO 4);
        sel0, sel1: IN std_logic;
        z: OUT std_logic_vector(1 TO 4));
END mux31_r2j;

ARCHITECTURE rtl OF mux31_r2j IS
   BEGIN
     PROCESS(clk)
       BEGIN
         z(1) <= (x2(1) AND sel1) OR
                 (x1(1) AND NOT sel1 AND sel0) OR
                 (x0(1) AND NOT sel1 AND NOT sel0);
         z(2) <= (x2(2) AND sel1) OR
                 (x1(2) AND NOT sel1 AND sel0) OR
                 (x0(2) AND NOT sel1 AND NOT sel0);
         z(3) <= (x2(3) AND sel1) OR
                 (x1(3) AND NOT sel1 AND sel0) OR
                 (x0(3) AND NOT sel1 AND NOT sel0);
         z(4) <= (x2(4) AND sel1) OR
                 (x1(4) AND NOT sel1 AND sel0) OR
                 (x0(4) AND NOT sel1 AND NOT sel0);
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module norm_r2j
--  Normalization of result w/ e-bit exponent
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY norm IS
  GENERIC (e: natural; ol_delay: natural);
    PORT (clk, reset, enable_in: IN std_logic;
          x: IN std_logic_vector(1 to 4);
          exp_x: IN std_logic_vector(1 TO e);
          enable_out: OUT std_logic;
          exp_z: OUT std_logic_vector(1 TO e));
END norm;

ARCHITECTURE rtl OF norm IS

  COMPONENT sube
    GENERIC (e: NATURAL);
      PORT (clk: IN std_logic;
            x,y: IN std_logic_vector(1 TO e);
            z: OUT std_logic_vector(1 TO e));
  END COMPONENT;

  COMPONENT dffm
    GENERIC (m: natural);
      PORT (d, clk, reset: IN std_logic;
            q: OUT std_logic);
  END COMPONENT;

  COMPONENT dff1
    PORT (d, clk, reset: IN std_logic;
          q: OUT std_logic);
  END COMPONENT;
```

```
  COMPONENT regm
    GENERIC (m: NATURAL);
      PORT (clk, reset: IN std_logic;
            d: IN std_logic_vector(1 to m);
            q: OUT std_logic_vector(1 to m));
  END COMPONENT;
  SIGNAL exp_int, exp_temp, exp_tempd,
         one: std_logic_vector(1 TO e);
  SIGNAL en_out, zero, val_in, valid: std_logic;
  SIGNAL zero_vec: std_logic_vector(1 TO e-1);
  BEGIN
    e1: sube
      GENERIC MAP (e=>e)
      PORT MAP (clk=>clk, x=>exp_int, y=>one, z=>exp_temp);
    d1: dffm
      GENERIC MAP (m=>ol_delay)
       PORT MAP (clk=>clk, reset=>reset, d=>enable_in, q=>en_out);
    d2: dff1
      PORT MAP (clk=>clk, reset=>reset, d=>val_in, q=>valid);
    r1: regm
      GENERIC MAP (m=>e)
      PORT MAP (clk=>clk, reset=>reset, d=>exp_temp, q=>exp_tempd);
    PROCESS(clk)
      BEGIN
        FOR i in 1 TO e LOOP
          exp_int(i) <= (exp_tempd(i) AND zero AND en_out) OR
                        (exp_x(i) AND NOT en_out) OR
                        (exp_int(i) AND val_in);
        END LOOP;
        zero <= NOT(x(1) XOR x(2)) AND NOT(x(3) XOR x(4));
        zero_vec <= (OTHERS=>'0');
        one <= zero_vec & '1';
exp_z <= exp_int;
val_in <= valid OR (en_out AND NOT zero);
        enable_out <= valid;
    END PROCESS;
  END rtl;

------------------------------------------------------------------
-- Module seldiv_r2j
-- Radix 2j borrow-save division selection function
------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY seldiv_r2j IS
  PORT (clk: IN std_logic;
        div: IN std_logic_vector(1 to 4);
        res: IN std_logic_vector(1 to 8);
        q: OUT std_logic_vector(1 to 4));
END seldiv_r2j;
ARCHITECTURE rtl OF seldiv_r2j IS
  COMPONENT bsdcomp_r2j
```

```
  PORT (x,y: IN std_logic_vector(1 TO 4);
        gg_in, g_in, e_in, s_in, ss_in: IN std_logic;
        gg_out, g_out, e_out, s_out, ss_out: OUT std_logic);
END COMPONENT;

SIGNAL neg_3, neg_2, neg_1, zero_0: std_logic_vector(1 TO 8);
SIGNAL pos_1, pos_2, pos_3: std_logic_vector(1 TO 8);
SIGNAL ngg3, ng3, ne3, ns3, nss3: std_logic_vector(0 TO 2);
SIGNAL ngg2, ng2, ne2, ns2, nss2: std_logic_vector(0 TO 2);
SIGNAL ngg1, ng1, ne1, ns1, nss1: std_logic_vector(0 TO 2);
SIGNAL gg0,  g0,  e0, s0, ss0: std_logic_vector(0 TO 2);
SIGNAL pgg1, pg1, pe1, ps1, pss1: std_logic_vector(0 TO 2);
SIGNAL pgg2, pg2, pe2, ps2, pss2: std_logic_vector(0 TO 2);
SIGNAL pgg3, pg3, pe3, ps3, pss3: std_logic_vector(0 TO 2);
SIGNAL div0, div1, div2, div3: std_logic;

BEGIN
  f1: FOR i IN 1 TO 2 GENERATE
    neg3: bsdcomp_r2j
      PORT MAP (x=>res(4*i-3 to 4*i), y=>neg_3(4*i-3 to 4*i),
                gg_in=>ngg3(i-1), g_in=>ng3(i-1), e_in=>ne3(i-1),
                s_in=>ns3(i-1), ss_in=>nss3(i-1),
                gg_out=>ngg3(i), g_out=>ng3(i), e_out=>ne3(i),
                s_out=>ns3(i), ss_out=>nss3(i));
    neg2: bsdcomp_r2j
      PORT MAP (x=>res(4*i-3 to 4*i), y=>neg_2(4*i-3 to 4*i),
                gg_in=>ngg2(i-1), g_in=>ng2(i-1), e_in=>ne2(i-1),
                s_in=>ns2(i-1), ss_in=>nss2(i-1),
                gg_out=>ngg2(i), g_out=>ng2(i), e_out=>ne2(i),
                s_out=>ns2(i), ss_out=>nss2(i));
    neg1: bsdcomp_r2j
      PORT MAP (x=>res(4*i-3 to 4*i), y=>neg_1(4*i-3 to 4*i),
                gg_in=>ngg1(i-1), g_in=>ng1(i-1), e_in=>ne1(i-1),
                s_in=>ns1(i-1), ss_in=>nss1(i-1),
                gg_out=>ngg1(i), g_out=>ng1(i), e_out=>ne1(i),
                s_out=>ns1(i), ss_out=>nss1(i));
    zero: bsdcomp_r2j
      PORT MAP (x=>res(4*i-3 to 4*i), y=>zero_0(4*i-3 to 4*i),
                gg_in=>gg0(i-1), g_in=>g0(i-1), e_in=>e0(i-1),
                s_in=>s0(i-1), ss_in=>ss0(i-1),
                gg_out=>gg0(i), g_out=>g0(i), e_out=>e0(i),
                s_out=>s0(i), ss_out=>ss0(i));
    pos1: bsdcomp_r2j
      PORT MAP (x=>res(4*i-3 to 4*i), y=>pos_1(4*i-3 to 4*i),
                gg_in=>pgg1(i-1), g_in=>pg1(i-1), e_in=>pe1(i-1),
                s_in=>ps1(i-1), ss_in=>pss1(i-1),
                gg_out=>pgg1(i), g_out=>pg1(i), e_out=>pe1(i),
                s_out=>ps1(i), ss_out=>pss1(i));
    pos2: bsdcomp_r2j
      PORT MAP (x=>res(4*i-3 to 4*i), y=>pos_2(4*i-3 to 4*i),
                gg_in=>pgg2(i-1), g_in=>pg2(i-1), e_in=>pe2(i-1),
                s_in=>ps2(i-1), ss_in=>pss2(i-1),
                gg_out=>pgg2(i), g_out=>pg2(i), e_out=>pe2(i),
                s_out=>ps2(i), ss_out=>pss2(i));
    pos3: bsdcomp_r2j
```

```
    PORT MAP (x=>res(4*i-3 to 4*i), y=>pos_3(4*i-3 to 4*i),
              gg_in=>pgg3(i-1), g_in=>pg3(i-1), e_in=>pe3(i-1),
              s_in=>ps3(i-1), ss_in=>pss3(i-1),
              gg_out=>pgg3(i), g_out=>pg3(i), e_out=>pe3(i),
              s_out=>ps3(i), ss_out=>pss3(i));
END GENERATE;

PROCESS(clk)
  BEGIN
   ngg3(0) <= '0';
   ng3(0) <= '0';
   ne3(0) <= '1';
   ns3(0) <= '0';
   nss3(0) <= '0';
   ngg2(0) <= '0';
   ng2(0) <= '0';
   ne2(0) <= '1';
   ns2(0) <= '0';
   nss2(0) <= '0';
   ngg1(0) <= '0';
   ng1(0) <= '0';
   ne1(0) <= '1';
   ns1(0) <= '0';
   nss1(0) <= '0';
   gg0(0) <= '0';
   g0(0) <= '0';
   e0(0) <= '1';
   s0(0) <= '0';
   ss0(0) <= '0';
   pgg1(0) <= '0';
   pg1(0) <= '0';
   pe1(0) <= '1';
   ps1(0) <= '0';
   pss1(0) <= '0';
   pgg2(0) <= '0';
   pg2(0) <= '0';
   pe2(0) <= '1';
   ps2(0) <= '0';
   pss2(0) <= '0';
   pgg3(0) <= '0';
   pg3(0) <= '0';
   pe3(0) <= '1';
   ps3(0) <= '0';
   pss3(0) <= '0';

   div0 <= NOT(div(1) XOR div(2)) AND NOT(div(3) XOR div(4));
   div1 <= NOT(div(1) XOR div(2)) AND div(3) AND NOT div(4);
   div2 <= div(1) AND NOT div(2) AND NOT(div(3) XOR div(4));
   div3 <= div(1) AND NOT div(2) AND div(3) AND NOT div(4);

   neg_3(1 to 8)<= '0' & (div1 OR div2 OR div3) & '0' &
                   (div0 OR div3) & '0' & (div0 OR div2) &
                   '0' & div0;
   neg_2(1 to 8)<= '0' & (div2 OR div3) & '0' &
                   (div0 OR div1) & '0' & div1 & '0' & div0;
```

```
      neg_1(1 to 8)<= "000" & (div1 OR div2 OR div3) & '0' &
                      div0 & '0' & div0;
      zero_0(1 TO 8)<= "00000" & (div1 OR div2 OR div3) &
                      '0' & div0;
      pos_1(1 to 8)<= "0000" & (div1 OR div2 OR div3) &
                     '0' & div0 & '0';
      pos_2(1 to 8)<= "00" & (div1 OR div2 OR div3) & '0' & div0 &
                     '0' & div0 & '0';
      pos_3(1 to 8)<= (div2 OR div3) & '0' & (div0 OR div1) &
                     '0' & div1 & '0' & div0 & '0';

      q(1)<= pgg2(2) OR pg2(2);
      q(2)<= ngg1(2) OR ng1(2);
      q(3)<= pgg3(2) OR pg3(2) OR ((pgg1(2) OR pg1(2)) AND
             (pss2(2) OR ps2(2)));
      q(4)<= ngg3(2) OR ng3(2) OR ((ngg1(2) OR ng1(2)) AND
             (ss0(2) OR s0(2)));

    END PROCESS;
  END rtl;

-------------------------------------------------------------------
-- Module selsqrt_r2j
-- Radix 2j borrow-save square root selection function
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY selsqrt_r2j IS
  PORT (clk: IN std_logic;
        rad: IN std_logic_vector(1 to 4);
        res: IN std_logic_vector(1 to 8);
        q: OUT std_logic_vector(1 to 4));
END selsqrt_r2j;

ARCHITECTURE rtl OF selsqrt_r2j IS

  COMPONENT bsdcomp_r2j
    PORT (x,y: IN std_logic_vector(1 TO 4);
          gg_in, g_in, e_in, s_in, ss_in: IN std_logic;
          gg_out, g_out, e_out, s_out, ss_out: OUT std_logic);
  END COMPONENT;

  SIGNAL neg_3, neg_2, neg_1, zero_0: std_logic_vector(1 TO 8);
  SIGNAL pos_1, pos_2, pos_3: std_logic_vector(1 TO 8);
  SIGNAL ngg3, ng3, ne3, ns3, nss3: std_logic_vector(0 TO 2);
  SIGNAL ngg2, ng2, ne2, ns2, nss2: std_logic_vector(0 TO 2);
  SIGNAL ngg1, ng1, ne1, ns1, nss1: std_logic_vector(0 TO 2);
  SIGNAL gg0,  g0,  e0, s0, ss0: std_logic_vector(0 TO 2);
  SIGNAL pgg1, pg1, pe1, ps1, pss1: std_logic_vector(0 TO 2);
  SIGNAL pgg2, pg2, pe2, ps2, pss2: std_logic_vector(0 TO 2);
  SIGNAL pgg3, pg3, pe3, ps3, pss3: std_logic_vector(0 TO 2);
  SIGNAL rad0, rad1, rad2, rad3: std_logic;

  BEGIN
    f1: FOR i IN 1 TO 2 GENERATE
      neg3: bsdcomp_r2j
        PORT MAP (x=>res(4*i-3 to 4*i), y=>neg_3(4*i-3 to 4*i),
                  gg_in=>ngg3(i-1), g_in=>ng3(i-1), e_in=>ne3(i-1),
```

```
            s_in=>ns3(i-1), ss_in=>nss3(i-1),
            gg_out=>ngg3(i), g_out=>ng3(i), e_out=>ne3(i),
            s_out=>ns3(i), ss_out=>nss3(i));
  neg2: bsdcomp_r2j
    PORT MAP (x=>res(4*i-3 to 4*i), y=>neg_2(4*i-3 to 4*i),
            gg_in=>ngg2(i-1), g_in=>ng2(i-1), e_in=>ne2(i-1),
            s_in=>ns2(i-1), ss_in=>nss2(i-1),
            gg_out=>ngg2(i), g_out=>ng2(i), e_out=>ne2(i),
            s_out=>ns2(i), ss_out=>nss2(i));
  neg1: bsdcomp_r2j
    PORT MAP (x=>res(4*i-3 to 4*i), y=>neg_1(4*i-3 to 4*i),
            gg_in=>ngg1(i-1), g_in=>ng1(i-1), e_in=>ne1(i-1),
            s_in=>ns1(i-1), ss_in=>nss1(i-1),
            gg_out=>ngg1(i), g_out=>ng1(i), e_out=>ne1(i),
            s_out=>ns1(i), ss_out=>nss1(i));
  zero: bsdcomp_r2j
    PORT MAP (x=>res(4*i-3 to 4*i), y=>zero_0(4*i-3 to 4*i),
            gg_in=>gg0(i-1), g_in=>g0(i-1), e_in=>e0(i-1),
            s_in=>s0(i-1), ss_in=>ss0(i-1),
            gg_out=>gg0(i), g_out=>g0(i), e_out=>e0(i),
            s_out=>s0(i), ss_out=>ss0(i));
  pos1: bsdcomp_r2j
    PORT MAP (x=>res(4*i-3 to 4*i), y=>pos_1(4*i-3 to 4*i),
            gg_in=>pgg1(i-1), g_in=>pg1(i-1), e_in=>pe1(i-1),
            s_in=>ps1(i-1), ss_in=>pss1(i-1),
            gg_out=>pgg1(i), g_out=>pg1(i), e_out=>pe1(i),
            s_out=>ps1(i), ss_out=>pss1(i));
  pos2: bsdcomp_r2j
    PORT MAP (x=>res(4*i-3 to 4*i), y=>pos_2(4*i-3 to 4*i),
            gg_in=>pgg2(i-1), g_in=>pg2(i-1), e_in=>pe2(i-1),
            s_in=>ps2(i-1), ss_in=>pss2(i-1),
            gg_out=>pgg2(i), g_out=>pg2(i), e_out=>pe2(i),
            s_out=>ps2(i), ss_out=>pss2(i));
  pos3: bsdcomp_r2j
    PORT MAP (x=>res(4*i-3 to 4*i), y=>pos_3(4*i-3 to 4*i),
            gg_in=>pgg3(i-1), g_in=>pg3(i-1), e_in=>pe3(i-1),
            s_in=>ps3(i-1), ss_in=>pss3(i-1),
            gg_out=>pgg3(i), g_out=>pg3(i), e_out=>pe3(i),
            s_out=>ps3(i), ss_out=>pss3(i));
END GENERATE;

PROCESS(clk)
  BEGIN
   ngg3(0) <= '0';
   ng3(0) <= '0';
   ne3(0) <= '1';
   ns3(0) <= '0';
   nss3(0) <= '0';
   ngg2(0) <= '0';
   ng2(0) <= '0';
   ne2(0) <= '1';
   ns2(0) <= '0';
   nss2(0) <= '0';
   ngg1(0) <= '0';
```

```
    ng1(0) <= '0';
    ne1(0) <= '1';
    ns1(0) <= '0';
    nss1(0) <= '0';
    gg0(0) <= '0';
    g0(0) <= '0';
    e0(0) <= '1';
    s0(0) <= '0';
    ss0(0) <= '0';
    pgg1(0) <= '0';
    pg1(0) <= '0';
    pe1(0) <= '1';
    ps1(0) <= '0';
    pss1(0) <= '0';
    pgg2(0) <= '0';
    pg2(0) <= '0';
    pe2(0) <= '1';
    ps2(0) <= '0';
    pss2(0) <= '0';
    pgg3(0) <= '0';
    pg3(0) <= '0';
    pe3(0) <= '1';
    ps3(0) <= '0';
    pss3(0) <= '0';

    rad0 <= NOT(rad(1) XOR rad(2)) AND NOT(rad(3) XOR rad(4));
    rad1 <= NOT(rad(1) XOR rad(2)) AND rad(3) AND NOT rad(4);
    rad2 <= rad(1) AND NOT rad(2) AND NOT(rad(3) XOR rad(4));
    rad3 <= rad(1) AND NOT rad(2) AND rad(3) AND NOT rad(4);

    neg_3(1 to 8)<= '0' & (rad1 OR rad2) & '0' &
                    (rad0 OR rad3) & '0' & (rad0 OR rad2) &
                    '0' & rad0;
    neg_2(1 to 8)<= '0' & (rad2 OR rad3) & '0' &
                    (rad0 OR rad1) & '0' & (rad1 OR rad3) &
                    '0' & rad0;
    neg_1(1 to 8)<= '0' & rad3 & '0' & (rad1 OR rad2) &
                    '0' & rad0 & '0' & rad0;
    zero_0(1 TO 8)<= "000" & rad3 & '0' & (rad1 OR rad2) &
                     '0' & rad0;
    pos_1(1 to 8)<= "0000" & (rad1 OR rad2 OR rad3) &
                    '0' & rad0 & '0';
    pos_2(1 to 8)<= "00" & (rad1 OR rad2 OR rad3) &
                    '0' & rad0 & '0' & rad0 & '0';
    pos_3(1 to 8)<= (rad2 OR rad3) & '0' & (rad0 OR rad1) &
                    '0' & rad1 & '0' & rad0 & '0';

    q(1)<= pgg2(2) OR pg2(2);
    q(2)<= ngg1(2) OR ng1(2);
    q(3)<= pgg3(2) OR pg3(2) OR ((pgg1(2) OR pg1(2)) AND
           (pss2(2) OR ps2(2)));
    q(4) <= ngg3(2) OR ng3(2) OR ((ngg1(2) OR ng1(2)) AND
            (ss0(2) OR s0(2)));
  END PROCESS;
END rtl;
```

```
-------------------------------------------------------------------
--  Module swapd_r2j
--  Swaps two radix 2j digits and corresponding e-bit exponents
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY swapd_r2j IS
  GENERIC (e: NATURAL);
    PORT (clk: IN std_logic;
          x_in, y_in: IN std_logic_vector(1 TO 4);
          expx, expy: IN std_logic_vector(1 TO e);
          sel: IN std_logic;
          expx_out: OUT std_logic_vector(1 TO e);
          x_out, y_out: OUT std_logic_vector(1 TO 4));
END swapd_r2j;

ARCHITECTURE rtl OF swapd_r2j IS
  BEGIN
    PROCESS(clk)
      BEGIN
        IF (sel='0') THEN
          x_out <= x_in;
          y_out <= y_in;
          expx_out <= expx;
        ELSE
          x_out <= y_in;
          y_out <= x_in;
          expx_out <= expy;
        END IF;
      END PROCESS;
  END rtl;

-------------------------------------------------------------------
--  Module twice_r2j
--  Radix 2j borrow-save digit multiplication by 2
-------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY twice_r2j IS
  PORT (clk: IN std_logic;
        x, x_d2: IN std_logic_vector(1 TO 2);
        z: OUT std_logic_vector(1 TO 4));
END twice_r2j;

ARCHITECTURE rtl OF twice_r2j IS
  BEGIN
    z <= x(1) & x(2) & NOT x_d2(1) & NOT x_d2(2);
  END rtl;
```

Bibliography

[AAH97] T. Aoki, H. Amada, and T. Higuchi, "Real/Complex Reconfigurable Arithmetic Using Redundant Complex Number Systems," *Proceedings of the 13th Symposium on Computer Arithmetic*, 1997, pp. 200-207.

[AFG91] G. Adams, A.M. Finn, and M.F. Griffin, "A fast implementation of the complex singular value decomposition on the connection machine", *IEEE Transactions on Acostic Speech and Signal Processing Proceedings*, 1991, pp. 1129-1132.

[AOH95] T. Aoki, Y. Ohi, and T. Higuchi, "Redundant complex number arithmetic for high-speed signal processing," *1995 IEEE Workshop on VLSI Signal Processing*, Oct. 1995, pp. 523-532.

[Ast02] http://arith.cs.ucla.edu/Astra.

[BMP88] B. Barazesh, J. Michalina and A. Picco, "A VLSI signal processor with complex arithmetic capability," *IEEE Transactions on Circuits and Systems*, Vol. 35, No. 5, May 1988, pp. 495-505.

[BOT97] A. Berkeman, V. Owall, and M. Torkelson, "A fast complex tree multiplier using distributed arithmetic," *Proceedings of the 15th NORCHIP Seminar*, 1997, pp. 121-126.

[BL85] R. Brent and F. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays", *SIAM Journal on Scientific and Statistical Computing*, Vol. 6, No. 1, Jan. 1985, pp. 69-84.

[BLV85] R. Brent, F. Luk, and C. van Loan, "Computation of the singular value decomposition using mesh-connected processors," *Journal of VLSI and Computer Systems*, Vol. 1, No, 3, 1985, pp. 242-270.

[BG69] P.A. Businger and G.H. Golub, "Singular value decomposition of a complex matrix", *Communications of the ACM*, Vol. 12, No. 10, Oct. 1969, p. 564.

[CP00] Y-N Chang and K. Parhi, "High-Performace Digit-Serial Complex Multiplier," *IEEE Transactions on Circuits and Systems–II: Analog and Digital Signal Processing*, Vol. 47, No. 6, June 2000, pp. 570-572.

[DHK91] J. Duprat, Y. Herreros, and S. Kla, "New redundant representations of complex numbers and vectors," *Proceedings of the 10th Symposium on computer Arithmetic*, 1991, pp. 2-9.

[EL88a] M.D. Ercegovac and T. Lang, "On-line arithmetic: a design methodology and applications in digital signal processing," *1988 Workshop on VLSI Signal Processing*, 1988, pp. 252-263.

[EL88b] M.D. Ercegovac and T. Lang, "On-line scheme for computing rotation factors," *Journal of parallel and distributed computing*, 1988, pp. 209-227.

[Erc78] M.D. Ercegovac, "An on-line square rooting algorithm," *Proceedings of the 4th Symposium on Computer Arithmetic*, 1978, pp. 183-189.

[Erc84b] M.D. Ercegovac, "On-line Arithmetic: An Overview," *Real Time Signal Processing VII SPIE-495*, 1984, pp. 86-93.

[EWS99] M.A. Elliot, G.A. Walker, A. Swift, and K. Vandenborne, "Spectral Quantitation by Principal Component Analysis using Complex Singular Value Decomposition," *Magnetic Resonance in Medicine*, Vol. 41, 1999, pp. 450-455.

[Fer93] J.S. Fernando, *Design Alternatives for Recursive Digital Filters Using On-line Arithmetic*, Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, 1993.

[Fro96] C. Frougny, "Parallel and On-Line Addition in Negative Base and Some Complex Number Systems," *Euro-Par '96 Parallel Processing*, Vol. 2, 1996, pp. 175-182.

[GT96] B. Girau and A. Tisserand, "On-line Arithmetic-Based Reprogrammable Hardware Implementation of Multilayer Perceptron Back-Propagation," *Proceedings of MicroNeuro '96*, 1996, pp. 168-175.

[GV89] G.H. Golub and C.F Van Loan, *Matrix Computations, 2nd edition*, Johns Hopkins Univ. Press, Baltimore, MD, 1989.

[Gor81] A. Gorji-Sinaki, *Error-coded algorithms for on-line arithmetic*, Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, 1981.

[GE81] A. Gorji-Sinaki and M.D. Ercegovac, "Design of a digit-slice on-line arithmetic unit," *Proceedings of the 5th Symposium on Computer Arithmetic*, 1981, pp. 72-80.

[GHM89] A. Guyot, Y. Herreros, and J.M. Muller, "JANUS, an On-line Multiplier/divider for manipulating large numbers," *Proceedings of the 9th Symposium on Computer Arithmetic*, 1989, pp. 106-111.

[HC92] N. Hemkumar and J. Cavallaro, "A Systolic VLSI Architecture for Complex SVD," *Proceedings of the 1992 IEEE International Symposium on Circuits and Systems*, May 1992, pp. 1061-1064.

[HD96] S-F Hsiao and J-M Delosme, "Parallel singular value decomposition of complex matrices using multidimensional CORDIC algorithms," *IEEE Transactions on Signal Processing*, Vol. 44, No. 3, Mar. 1996, pp. 685-697.

[HLD00] S-F Hsaio, C-Y Lau, and J-M Delosme, "Redundant constant-factor implementation of multi-dimensional CORDIC and its application to complex SVD," *Journal of VLSI Signal Processing*, Vol. 25, 2000, pp. 155-166.

[HT95] S. He and M. Torkelson, "A Pipelined Bit-Serial Complex Multiplier Using Distributed Arithmetic," *1995 IEEE Symposium on Circuits and Systems*, 1995, pp. 2313-2316.

[Irw77] M.J. Irwin, *An arithmetic unit for on-line computation*, Ph.D. dissertation, Department of Computer Science, University of Illinois, Champaign-Urbana, 1977.

[KJM85] R. Krishnan, G.A. Jullien, and W.C. Miller, "Complex Digital Signal Processing using Quadratic Residue Number Systems," *IEEE Transactions on Acoustic, Speech, and Signal Processing*, Vol.—34, No. 1, Feb. 1986.

[Knu73] D.E. Knuth, "The Art of Computer Programming," Vol. 2, 1973.

[LSE99] D. Lau, A. Schneider, M. Ercegovac, and J. Villasenor, "FPGA-based structures for on-line FFT and DCT," *7th Annual IEEE Symposium on Field-programmable Custom Computing Machines*, Apr. 1999, pp. 310-311.

[Leu81] S.H. Leung, "Application on Residue Number Systems to Complex Digital Filters," *Proceedings of the Fifteenth Asilomar Conference on Circuits, Systems, and Computers*, Nov. 1981, pp. 70-74.

[Lew99] D. Lewis, "Complex Logarithmic Number System Arithmetic Using High-Radix Redundant CORDIC Algorithms," *14th IEEE Symposium on Computer Arithmetic*, Apr. 1999, pp. 194-203.

[LS89] H.X. Lin and H.J. Sips, "On-line CORDIC algorithms", *Proceedings of the 9th Symposium on Computer Arithmetic*, 1989, pp. 26-33.

[MS99] E. Mosanya and E. Sanchez, "A FPGA-based hardware implementation of Generalized Profile Search using online arithmetic," *AGM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1999, pp. 101-111.

[Nie96] A.M. Nielsen, *Number systems and Digit Serial Arithmetic*, Ph.D. thesis, Dept. of Mathematics and Computer Science, Odense University, Denmark. 1996.

[OVS94] V. Oklobdzija, D. Villeger, and T. Soulas, "An Integrated Multiplier for Complex Numbers," *Journal of VLSI Signal Processing*, Vol. 7, 1994, pp. 213-222.

[RE81] C.S. Raghavendra and M.D. Ercegovac, "A simulator for on-line arithmetic," *Proceedings of the 5th Symposium on Computer Arithmetic*, 1981, pp. 92-98.

[SP84] M.A. Soderstrand and G.D. Poe, "Applications of Quadratic-like Complex Residue Number System Arithmetic to Ultrasonics," *Proceedings of 1984 ICASSP*, Mar. 1984.

[SSB98] 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, June 1998, pp. 904-909.

[SSS89] A. Skavantzos, Z. Sakari, and T. Stouraitis, "A complex DSP processor using polynomial encoding," *IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 2, 1989, pp. 1310-1313.

[SP84] M. Soderstrand and G.D. Poe, "Applications of Quadratic-like Complex Residue Number System Arithmetic to Ultrasonics," *Proceedings of the 1984 International Conference on Acoustics, Speech and Signal Processing*, Mar. 1984.

[SZY98] R.D. Susanto, Q. Zheng, and X-H. Yan, "Complex Singular Value Decomposition," *Journal of Atmospheric and Oceanic Technology*, Vol. 15, No. 3, 1998, pp. 764-774.

[TD97] A. Tisserand and M. Dimmler, "FPGA Implementation of Real-Time Digital Controllers using On-Line Arithmetic," *7th International Workshop on Field-programmable Logic and Applications*, Sept. 1997, pp. 472-481.

[TE77] K.S. Trivedi and M.D. Ercegovac, "On-line Algorithms for Division and Multiplication," *IEEE Transactions on Computers*, Vol. C-26, No. 7, July 1977, pp. 681-687.

[TE98] A. Tenca and M.D. Ercegovac, "A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 216-225.

[TMP99] A, Tisserand, P. Marchal, and C. Piguet, "An on-line arithmetic based FPGA for low power custom computing," *9th International Workshop on Field Programmable Logic and Applications*, 1999, pp. 264-273.

[Tu86] D.M. Tullsen, *A Very Large Scale Integration Implemenation of an On-line Arithmetic Unit*, M.S. Thesis, Computer Science Department, University of California, Los Angeles, 1986.

[Tu90] P.K-G. Tu, *On-line arithmetic algorithms for efficient implementaion*, Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, 1990.

[Wat81] O. Watanuki, "Floating-point on-line arithmetic: algorithms," *Proceedings of the 5th Symposium on Computer Arithmetic*, 1981, pp. 81-86.

[WDC95] B. Wei, H. Du, and H. Chen, "A Complex-Number Multiplier Using Radix-4 Digits," *Proceedings of the 12th Symposium on Computer Arithmetic*, 1995, pp. 84-90.

[Xil01] Xilinx Corporation, "Xilinx Data Book," 2001.