On the Design of an On-line Complex Matrix Inversion Unit

Robert McIlhenny
Computer Science Department
California State University, Northridge
Northridge, CA 91330
rmcilhen@csun.edu

Miloš D. Ercegovac
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
milos@cs.ucla.edu

Abstract— **In this paper, we present a novel implementation for the inverse of an n-by-n matrix consisting of complex elements, using complex number on-line arithmetic, based on adopting a redundant complex number system (RCNS) to represent complex operands as a single number. We present comparisons with (i) a real number on-line arithmetic approach, and (ii) a real number arithmetic parallel approach, to demonstrate a significant improvement in cost and delay.**

I. Introduction

The inverse of a matrix is an important operation in the field of linear algebra, as well as various business and input-output models. The inverse of an n-by-n matrix A is denoted A^{-1} and satisfies $AA^{-1} = A^{-1}A = I_n$, where I_n is the n-by-n identiy matrix containing 1's along the main diagonal and 0's elsewhere. The inverse serves as a way to "divide" matrices, in that if $AB = C$, then $A = CB^{-1}$ and $B = A^{-1}C$. If a matrix A has no inverse (the determinant of A is 0), it is called *singular*. Various techniques for inverting an n-by-n matrix include: (i) the adjoint-matrix method [2], (ii) LU decomposition [8], and (iii) Gauss-Jordan elimination [7]. Due to the regular structure and relatively low cost of the Gauss-Jordan elimination method, it will be utilized for implementing the complex matrix inversion unit.

The Gauss-Jordan elimination method extends the original n-by-n matrix A to a n-by-$2n$ matrix:

$$[A|I] = \left[\begin{array}{ccc|cccc} a_{11} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 \\ a_{21} & \cdots & a_{2n} & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} & 0 & 0 & \cdots & 1 \end{array}\right] \quad (1)$$

Gaussian elimination is used to "zero-off" non-diagonal elements on the left half of the matrix, and the diagonal elements of the left half are scaled such that the resultant n-by-$2n$ matrix is:

$$[I|A^{-1}] = \left[\begin{array}{cccc|ccc} 1 & 0 & \cdots & 0 & a_{11}^{-1} & \cdots & a_{1n}^{-1} \\ 0 & 1 & \cdots & 0 & a_{21}^{-1} & \cdots & a_{2n}^{-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & a_{n1}^{-1} & \cdots & a_{nn}^{-1} \end{array}\right] \quad (2)$$

where A^{-1} is the matrix inverse of A.

II. Complex number on-line floating-point arithmetic

On-line arithmetic [4] is a class of arithmetic operations in which all operations are performed digit serially, in a most significant digit first (MSDF) manner. Several advantages, compared to conventional parallel arithmetic, include: (i) 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; (ii) low-bandwidth communication, since intermediate results pass to and from modules digit-serially, so connections need only be one digit wide; and (iii) support for variable precision, since once a desired precision is obtained, successive outputs can be ignored. 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, with on-line delay $\delta = 4$. The latency of an on-line arithmetic operator, assuming m-digit precision is then $\delta + m - 1$.

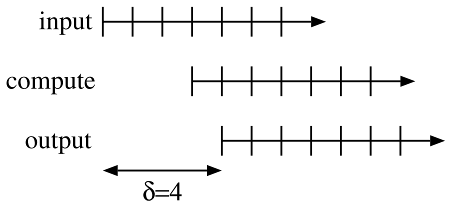

Fig. 1. On-line delay of a function

Complex number on-line arithmetic [6] uses a class of on-line arithmetic operators on complex number operands. For efficient representation, a *Redundant Complex Number System* (RCNS) [1] is adopted. A RCNS a radix rj system, in which digits are in the set $\{-a, \ldots, 0, \ldots, a\}$, where $r \geq 2$ and $\lceil r^2/2 \rceil \leq a \leq r^2 - 1$. Such a number system can be denoted $RCNS_{rj,a}$. A Redundant Complex Number System with $r = 2$, $a = 3$ 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.

This number system was introduced as Quarter-imaginary Number System in [5]. For implementation of the complex matrix inversion unit, in order to permit a relatively wide range of input values, we assume floating-point arithmetic. Three on-line floating-point arithmetic operations are used: (i) $RCNS_{2j,3}$ on-line floating-point addition; (ii) $RCNS_{2j,3}$ on-line floating-point multiplication; and (iii) $RCNS_{2j,3}$ on-line floating-point division. The recurrence algorithms and implementation parameters when mapped to a Xilinx Virtex FPGA are discussed in detail.

Using $RCNS_{2j,3}$, a floating-point 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.

A $RCNS_{2j,3}$ fraction x is considered normalized if $2^{-1} \leq \max(|X_R|, |X_I|) < 1$. The output of a complex number operation can be undernormalized for several reasons:

1. The range of an output determined by the on-line algorithm allows it to be undernormalized.
2. Digit cancellation resulting from the addition/subtraction of numbers with the same exponent value.

In this paper, we assume operands of an $RCNS_{2j,3}$ on-line algorithm have non-zero most significant digits and are normalized. When the result Z exceeds the range of a 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 normalized fraction (i.e. $\max(|Z_R, Z_I| < \frac{1}{2}$), 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 [3] for radix-2 on-line rotation.

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

/* Initialization */
   $done = 0$

/* Computation */
  if $k = (\delta - 2)$ and $z_k \neq 0$ then
      $e_z = e_z + 2$
     $done = 1$
  if $k = (\delta - 1)$ and $z_k \neq 0$ and not($done$) then
     $e_z = e_z + 1$
     $done = 1$
  else if $k \geq \delta$ and $z_k = 0$ and not($done$) then
     $e_z = e_z - 1$
  else if ($k \geq \delta$ and $z_k \neq 0$) then
     $done = 1$
  end if
```

III. 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 $-3X$, requiring an extra addition step.
- A significand X with fractional real and imaginary components X_R and X_I can have integer digits, such as $(11.32\overline{1}\overline{2})_{2j} = \frac{3}{8} + \frac{3}{8}j$, 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.

A. 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})

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

$\chi_k = t_k + w_k$

B. 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 next.

MSREC(w_{k-2}, w_k)

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

IV. $RCNS_{2j,3}$ ON-LINE FLOATING POINT ARITHMETIC OPERATIONS

An overview and implementation description for the operators used in the design of the complex matrix inversion unit are shown next. The implementation parameters when mapped to a Xilinx Virtex FPGA are discussed in detail.

A. $RCNS_{2j,3}$ *on-line floating-point addition*

$RCNS_{2j,3}$ 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

$$\begin{aligned} Z_R &= X_R + Y_R \\ Z_I &= X_I + Y_I \\ e_z &= \max(e_x, e_y) \end{aligned} \tag{3}$$

Each output digit at step $k - \delta$, namely $z_{k-\delta}$ is generated based on input digits x_k and y_k. The design of a m-digit significand and e-bit exponent $RCNS_{2j,3}$ on-line floating point adder is shown in Figure 2. The SUBE unit computes the difference of the exponents. The ALIGN unit performs alignment of operand y'_k to synchronize the arrival of the input digits. The SWAP unit exchanges the operands if necessary. The PPM and MMP modules are simple full-adders that appropriately negate (indicated by "-" on the port) inputs and outputs to perform borrow-save addition. The NORM unit normalizes the result by updating the output exponent e_z. A summary of cost of individual modules is shown in Table I. Assuming $m = 24$ and $e = 8$, the cost is 108 CLB slices. The on-line delay is $\delta = 3$.

B. $RCNS_{2j,3}$ *on-line floating-point multiplication*

$RCNS_{2j,3}$ 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

$$\begin{aligned} Z_R &= X_R Y_R - X_I Y_I \\ Z_I &= X_R Y_I + X_I Y_R \\ e_z &= e_x + e_y \end{aligned} \tag{4}$$

TABLE I
COST OF $RCNS_{2j,3}$ ON-LINE FLOATING-POINT ADDER

Module	Count	CLB slices
SUBE	1	e
ALIGN	1	$3m$
SWAP	1	e
PPM/MMP	4	4
NORM	1	$2e$
Total cost		$3m + 4e + 4$

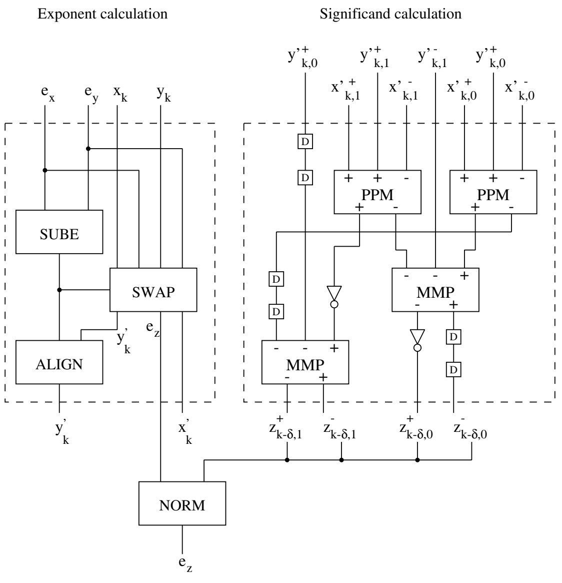

Fig. 2. $RCNS_{2j,3}$ on-line floating-point adder

An $RCNS_{2j,3}$ on-line floating-point multiplier can be designed as a series of modular slices M_i, 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 each cycle until the output digit $z_{k-\delta}$ is non-zero. The design of a m-digit significand and e-bit exponent on-line floating-point multiplier is shown in Figure 3. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table II. Assuming $m = 24$ and e=8, the total cost is 452 CLB slices. The on-line delay is $\delta = 9$.

TABLE II
COST OF $RCNS_{2j,3}$ ON-LINE FLOATING-POINT MULTIPLIER

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		$16m + 3e + 44$

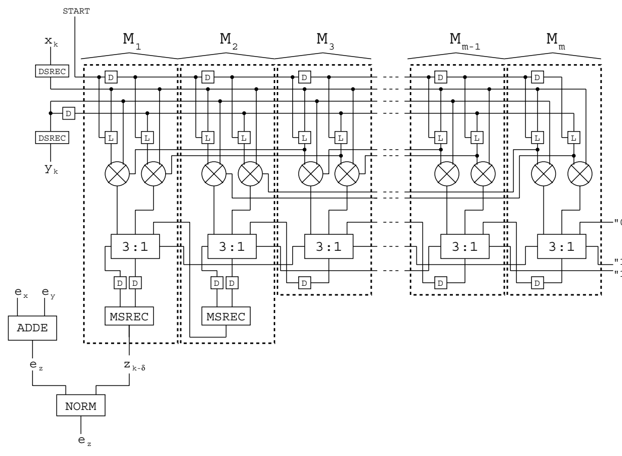

Fig. 3. $RCNS_{2j,3}$ on-line floating-point multiplier

C. $RCNS_{2j,3}$ on-line floating-point division

$RCNS_{2j,3}$ 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

$$\begin{aligned} Z_R &= \frac{X_R Y_R + X_I Y_I}{Y_R^2 + Y_I^2} \\ Z_I &= \frac{X_I Y_R - X_R Y_I}{Y_R^2 + Y_I^2} \\ e_z &= e_x - e_y \end{aligned} \tag{5}$$

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 $z_{k-\delta}$ 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 $z_{k-\delta}$. The SUBE unit subtracts the two exponents to produce the exponent of the output, not considering exponent overflow/underflow. The digit x_k is appended to the most significant end of the vector product $Z[k-1]y_k$. 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 divider is shown in Figure 4. The number of individual module types utilized, the cost per module type, and the total overall cost are summarized in Table III. Assuming $m = 24$ and e=8, the total cost is 545 CLB slices. The on-line delay is $\delta = 9$.

TABLE III
COST OF $RCNS_{2j,3}$ ON-LINE FLOATING-POINT DIVIDER

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		$16m + 3e + 137$

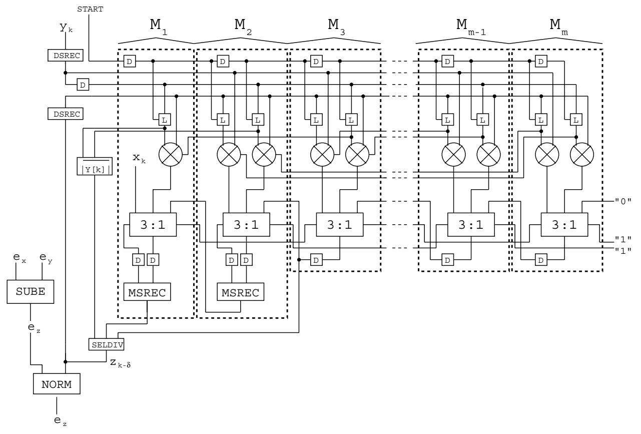

Fig. 4. $RCNS_{2j,3}$ on-line floating-point divider

V. IMPLEMENTATION

Each "sweep" of the Gauss-Jordan elimination method consists of zeroing-off" a non-diagonal element $a_{i,j}$ where $i \neq j$. The other elements $a_{i,k}$ in the same row, where $k \neq j$ are scaled as element $a'_{i,k}$ where

$$a'_{i,k} = a_{i-1,k} - \left(\frac{a_{i-1,j}}{a_{i,j}}\right) a_{i,k} \tag{6}$$

Since this operation is applied toward $(n-1)$ elements within each of $(n-1)$ rows successively n times, until finally only diagonal elements remain in the left half of the matrix, at which point the diagonal elements are scaled to produce 1's along the diagonal requiring n^2 divisions, the total number of arithmetic operations is: $n(n-1)^2$ additions/subtractions, $n(n-1)^2$ multiplications, and $2n(n-1)$ divisions. The total delay is: n additions/subtractions, n multiplications, and $(n+1)$ divisions. We compare a parallel radix 2 approach, an on-line radix 2 approach, and the on-line $RCNS_{2j,3}$ approach.

A. $RCNS_{2j,3}$ on-line network

An n-by-n complex matrix inversion unit can be designed as a network of $RCNS_{2j,3}$ floating-point arithmetic operators. For 24-bit significands and 8-bit exponents, the cost and on-line delay of a complex floating-point adder is 108 CLB slices and 3 cycles, respectively, the cost and on-line delay of an on-line complex floating-point multiplier is 452 CLB slices and 9 cycles, respectively, and the cost and delay of an on-line complex floating-point divider is 545 CLB slices and 9 cycles, respectively. The total cost is $560n^3 - 30n^2 - 530n$ CLB slices. The total latency, summing the on-line delays to produce the first digit, after which the remaining 23 digits are produced one per cycle, is $21n + 32$ cycles.

B. Radix 2 on-line network

An n-by-n complex matrix inversion unit can be alternatively designed as a network of radix 2 floating-point arithmetic operators, as described in [6]. For 24-bit significands and 8-bit exponents, the cost and on-line delay of an equivalent complex floating-point adder is 140 CLB slices and 3 cycles, respectively, the cost and on-line delay of an equivalent on-line complex floating-point multiplier is 868 CLB slices and 7 cycles, respectively, and the cost and delay of an equivalent on-line complex floating-point divider is 1502 CLB slices and 12 cycles, respectively. The total cost is $1008n^3 + 988n^2 - 1996n$ CLB slices. The total latency, summing the on-line delays to produce the first digit, after which the remaining 23 digits are produced one per cycle, is $22n + 35$ cycles.

C. Radix 2 parallel network

An n-by-n complex matrix inversion unit can be alternatively designed as a network of radix-2 parallel arithmetic operators. The library of Xilinx CORE floating-point arithmetic modules [9], which can be scaled in terms of precision is used. For 24-bit significands and 8-bit exponents, the cost and latency of an equivalent parallel complex floating-point adder is 672 CLB slices and 11 cycles, respectively, the cost and delay of an equivalent parallel complex floating-point multiplier is 1292 CLB slices and 17 cycles, and the cost and delay of an equivalent parallel complex floating-point divider is 3492 CLB slices and 44 cycles. The total cost is $1964n^3 + 3056n^2 - 5020n$ CLB slices. The total latency is $72n + 44$ cycles.

D. Cost comparison

The cost of the proposed $RCNS_{2j,3}$ on-line network, and the alternative radix 2 on-line network and the radix 2 parallel network are compared for the implementation of an n-by-n complex matrix inversion unit for various values of n. In each case, we assume floating-point operands consisting of 24-digit (or bit) significands and 8-bit exponents, as shown in Table IV.

E. Delay comparison

The delay of the proposed $RCNS_{2j,3}$ on-line network, and the alternative radix 2 on-line network and the radix 2 parallel network are compared for the implementation of an n-by-n complex matrix inversion unit for various values of n. In each case, we assume floating-point operands consisting of 24-digit (or bit) significands and 8-bit exponents, as shown in Table V.

TABLE IV
COMPARISON OF CLB COSTS FOR COMPLEX MATRIX INVERSION

n	$RCNS_{2j,3}$ on-line	Radix-2 on-line	Radix-2 parallel
2	3192	7884	15932
3	13260	30120	65472
4	33240	72236	154512
5	66600	140720	296800
6	116700	241320	504120

TABLE V
COMPARISON OF CYCLE LATENCIES FOR COMPLEX MATRIX INVERSION

n	$RCNS_{2j,3}$ on-line	Radix-2 on-line	Radix-2 parallel
2	44	45	72
3	95	101	260
4	116	123	332
5	137	145	404
6	158	167	476

VI. CONCLUSION

We have demonstrated a new approach for implementating an n-by-n complex matrix inversion unit, based on using complex number on-line arithmetic modules which adopt a redundant complex number system (RCNS) for efficient representation. Significant improvement in cost in comparison to a radix 2 on-line approach and a radix 2 parallel approach, as well as a significant reduction in latency in comparison to a radix 2 parallel appraoch have been shown. This offers motivation for further research into other applications utilizing complex number operations.

REFERENCES

[1] 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.

[2] A. Ben-Israel and T.N.E. Greville, ”Generalized Inverses: Theory and Applications”, 1977.

[3] M.D. Ercegovac and T. Lang, ”On-line scheme for computing rotation factors,” *Journal of paralle and distributed computing*, 1988. pp. 209-227.

[4] M.D. Ercegovac and T. Lang, “Digital Arithmetic,” Morgan Kaufmann Publishers, 2004.

[5] D.E. Knuth, “The art of computer programming,” Vol. 2, 1973.

[6] R. McIlhenny, “Complex number on-line arithmetic for reconfigurable hardware: algorithms, implementations, and applications,” *Ph.D. Dissertation, University of California, Los Angeles*, 2002.

[7] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Gauss-Jordan elimination,” *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 1992, pp. 27-32.

[8] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “LU Decomposition and Its Applications,” *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, 1992, pp. 34-42.

[9] Xilinx Corporation, “Xilinx Data Book,” 2004.