Science and Technology, 7 (2002) 199-209

Towards Formulation of a Complex Binary Number System

Tariq Jamil*, David Blest and Amer Al-Habsi***

**Department of Information Engineering, College of Engineering, Sultan Qaboos University, P.O. Box 33, Al Khod 123, Muscat, Sultanate of Oman. **School of Mathematics and Physics, University of Tasmania, Launceston, TAS 7250, Australia.*

نحو نظام للأرقام المركبة الثنائية

طارق جميل ، ديفيد بلست وعامر الحبسي

خلاصة : لسنوات عديدة ما زالت الأرقام المركبة (المعقدة) تعامل بشي من عدم الالفة نسبة الى الأرقام الحقيقة بغض النظر عن انتشار استعمالاتها في تطبيقات عديدة في مجالات الهندسة الكهربائية والحاسوبية. في هذه الأيام عمليات الحاسوب التي تشمل أرقام مركبة (معقدة) تنفذ في كثير من الأحيان بواسطة تطبيق طريقة تقسيم أجزاء الرقم المركب إلى مكوناته وتنفيذ العمليات الحسابية على كل جزء على حدى. فالجزء الحقيقي والخيالي من الرقم المركب يفصلوا عن بعض وتنفذ العمليات الحسابية على كل جزء ثم تجمع النتائج لكل جزء لتركيب النتيجة النهائية للعملية المركبة. هذه الطريقة في تعقيد العمليات الحسابية على الأرقام المركبة تحث على إيجاد طريقة لمعاملة الأرقام المركبة تحث على إيجاد طريقة لمعاملة الأرقام المركبة كوحدة واحدة وتنفيذ العمليات الحسابية عليها بطريقة مباشرة. في هذا البحث قمنا بتحليل واقتراح النظام الرقمي الثنائي المركب بأساس (j-1-) . ناقشنا العمليات الحسابية على عددين في هذا النظام واوضحنا الدراسات القائمة حاليا في مجال الحاسوب.

ABSTRACT: For years complex numbers have been treated as distant relatives of real numbers despite their widespread applications in the fields of electrical and computer engineering. These days computer operations involving complex numbers are most commonly performed by applying divide-and-conquer technique whereby each complex number is separated into its real and imaginary parts, operations are carried out on each group of real and imaginary components, and then the final result of the operation is obtained by accumulating the individual results of the real and imaginary components. This technique forsakes the advantages of using complex numbers in computer arithmetic and there exists a need, at least for some problems, to treat a complex number as one unit and to carry out all operations in this form. In this paper, we have analyzed and proposed a (–1–j)-base binary number system for complex numbers. We have discussed the arithmetic operations of two such binary numbers and outlined work which is currently underway in this area of computer arithmetic.

KEYWORDS: Complex Binary Number, Addition, Subtraction, Multiplication, Division.

1. Introduction

The use of complex numbers in mathematics can be traced as far back as 1545 when Cardano used the notation $\sqrt{-1}$ during investigation of the roots of polynomials. Later, Euler in 1777 introduced the abbreviation i for $\sqrt{-1}$ and originated the $a + ib$ notation to represent complex numbers (in electrical and computer engineering, we tend to replace the symbol i with j because it is easier to distinguish between the number 1 and j than 1 and i). Since then, complex numbers have played a truly unique role in the development and research of modern science and engineering. In the fields of electrical and computer engineering, the application of Fast Fourier Transform in most digital signal processing algorithms, and the geometric analysis of pixels in graphics and image processing owe their advantage to the use of complex numbers. Despite their widespread applications, complex number operations have, to a large extent, been treated as just an add-on patch to the basic operations of real arithmetic. Today, even with the availability of over

100-million transistors on a single IC-chip (Geppert, 1999), virtually the entire complex arithmetic involves the application of "divide-and-conquer" technique, whereby a complex number is broken-up into its real and imaginary parts and then operations are carried out on each part as if it were a part of the real arithmetic. Finally, the overall result of the complex operation is obtained by accumulation of the individual results. For instance, addition of two complex numbers (a + jb) and (c + jd) requires two separate additions (a + c) and (b + d) while multiplication of the same two complex numbers requires four multiplications (ac), (ad), (bc), (bd), one subtraction ($j^2bd = – bd$), and one addition (ac + j(ad + bc) + (– bd)). This can be effectively reduced to just one complex addition or only one multiplication and addition respectively for the given cases if each complex number is represented as one unit instead of two individual units.

The pursuance of providing equal opportunity representation to complex numbers has resulted in some efforts of defining binary numbers with bases other than 2. In 1960, Donald E. Knuth described a "quater-imaginary" number system with base 2j and analyzed the arithmetic operations of numbers using this imaginary base (Knuth, 1960). However, he was unsuccessful in providing a division algorithm and considered it as a main obstacle towards hardware implementation of any imaginary-base number system.

Walter Penney, in 1964, attempted to define a complex number system, first by using a negative base of – 4 (Penney, 1964) and then by using a complex number (–1+j) as the base (Penney, 1965). However, the main problem encountered with using these bases was again the inability to formulate an efficient divison process. Stepanenko (1996) utilizes the base $j\sqrt{2}$ in which the even powers of the base yield real numbers and the odd powers of the base result in imaginary numbers. Although partly successful in resolving the division problem as an "all-in-one" operation, in his algorithm "…everything…reduces to good choice of an initial approximation…" in a Newton-Raphson iteration which may or may not converge.

In an earlier paper (Jamil *et al* 2000), we revisited Penney's number system of base (–1+j) and extended his work by providing algorithms for converting integers, imaginary, fractional, and floating point numbers into (–1+j)-base binary number system, including description of the basic arithmetic operations based on this new number system.

In this paper, we concentrate our efforts on providing algorithms and arithmetic operations for (–1–j)-base binary number system. In addition to this, we have provided algorithms for obtaining conjugate and magnitude of the given (–1–j)-base complex binary number. This will help conclude the fact that both (–1+j) and (–1–j) are excellent bases for facilitation of complex numbers' representation as a single entity. This paper is organized as follows: In Section 2 we present an analysis of (–1–j)-base binary number system. In Section 3, we present algorithms for converting various types of numbers into the proposed (–1–j)-base binary number system. This is followed by an analysis of arithmetic operations in Section 4. In Section 5 we present algorithms for obtaining conjugate and magnitude of a given (–1–j)-base complex binary number. Finally, in Section 6 we present conclusion and a synopsis of the ongoing work being done by us in this area.

2. The Base –1–j

The value of an n-bit binary number with base (–1–j) can be written in the form of a power series as follows:

$$a_{n-1}(-1-j)^{n-1} + a_{n-2}(-1-j)^{n-2} + a_{n-3}(-1-j)^{n-3} + \ldots + a_2(-1-j)^2 + a_1(-1-j)^1 + a_0(-1-j)^0 \quad (1)$$

where the coefficients $a_{n-1}, a_{n-2}, a_{n-3}, \ldots, a_2, a_1, a_0$ are binary (either 0 or 1). Table 1 gives some real and imaginary numbers along with their complex binary representations (base –1–j).

Table 1: Binary representations for some real and imaginary numbers (base –1–j).

Real No.	Complex Binary Number	Imaginary No.	Complex Binary Number
–5	1100 1101	–j5	111 0011
–4	1 0000	–j4	111 0000
–3	1 0001	–j3	111 0111
–2	1 1100	–j2	111 0100
–1	1 1101	–j1	0011
0	0000	j0	0000
1	0001	j1	0111
2	1100	j2	0100
3	1101	j3	11 0011
4	1 1101 0000	j4	11 0000
5	1 1101 0001	j5	11 0111

3. Binary representation for complex numbers

3.1 Conversion algorithm for real integers[1]

Let's first begin with the case of positive integers *N*. To represent *N* in the proposed (–1–j)-base binary number system, we express *N* in terms of powers of 4 using the division process. Thus

$$N_{base\ 4} = \sum q_i\, 4^i \tag{2}$$

This "normalized" representation is unique when $0 \leq q_i < 4$. In that case the non-zero 'digits' …, $q_5, q_4, q_3, q_2, q_1 q_0$ are called the base 4 representation of *N*. If the constraint on the q_i is removed, then we call it an un-normalized base 4 representation of N, which is not unique. Now convert the base 4 number …, $q_5, q_4, q_3, q_2, q_1 q_0$ to base – 4 by replacing each digit in odd location $q_1, q_3, q_5, \ldots$ with its negative to get

$$(\ldots, q_5, q_4, q_3, q_2, q_1, q_0)_{base\ 4} = (\ldots, -q_5, q_4, -q_3, q_2, -q_1, q_0)_{base-4} \text{ (un-normalized)}$$

We normalize the new number (i.e. get each digit in the range 0 to 3) by repeatedly using the operation of adding four to the negative digits and adding a one to the digit on its left. This operation will get rid of the negative numbers, but might create some digits with a value of 4 after the addition of a 1. To normalize this, we replace the four by a zero and subtract a one from the digit on its left. Of course this subtraction might once again introduce negative digits which will be normalized by the previous method, but this process will terminate! What is interesting is that with negative bases, all integers, positive or negative have a unique positive representation. As an example

$$55_{base10} = (3,1,3)_{base\ 4} = (3,-1,3)_{base-4} = (4,3,3)_{base-4} = (-1,0,3,3)_{base-4}$$
$$= (1,3,0,3,3)_{base-4} \quad \text{(normalized)}$$

To represent the given number in the base (–1–j), we replace each digit in base – 4 representation with a four bit sequence according to Table 2, which yields:

$$55_{base10} = \quad (1,3,0,3,3)_{base-4} = 0001\ 1101\ 0000\ 1101\ 1101_{base-1-j}$$

[1] See Figure 1 for a program in C language to convert real integers into (–1–j)-base complex binary number system.

To convert a negative integer into (–1–j)-base representation, we simply multiply the representation of the corresponding positive integer with 11101 (equivalent to $-1_{base-1-j}$) according to the multiplication algorithm given in Section 4.3.

Thus -55_{base10} = (0001 1101 0000 1101 1101) x (11101)
= 0000 0001 1100 1101 1101 0001 $_{base\ -1-j}$

Figure 1: A program in C language for conversion of real and imaginary integers to (–1–j)-base complex binary number system.

```
/* Program for Real and
Imaginary       */
/* Integers Conversion
to (-1-j)-base  */
/* Complex Binary Number
System        */
/* Author: Amer Al-Habsi
*/
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include "base1j.h"
int clear(int *a, int n)
{      int i;
       for(i=0; i<n-1;
i++)
       a[i]=0;
       a[n-1]=1;
/*Length = 1*/
       return 0;}
int print(int *a)
{      int i;
       for(i=MAX-1; i>=0;
i--)
       printf("%i
",a[i]);
       return 0;}
int print_bits(int *a,
int n)
{      int i, digits;
       i=n-1;
       while(a[--i]==0) ;
digits=i;
       for(i=digits;
i>=0; i--)
       printf("%i",a[i]);
       return 0; }
int sprint_bits(int *a,
int n, char *s)
{      int i, digits;
       i=n-1;
       while(a[--i]==0 &&
i>=0);
       digits=i;
       if(digits==-1){

sprintf(s++,"0");
            return 0; }
       for(i=digits;
i>=0; i--)

sprintf(s++,"%i",a[i]);
       return 0;}
int int2basen(int x, int
n, int a[])
{      int i, remainder,
dividend;
       i=0;
       do {
        dividend=x/n;
        remainder=x%n;
        a[i++]=remainder;
        if(i==MAX-1)
return -1;
        x=dividend;}
       while(dividend!=0)
;
       a[MAX-1]=i; return
0;}
int basenegative(int
a[])
{      int i;
       for(i=1; i<MAX-1;
i+=2)
       a[i]=-a[i];
       return 0;}
int done(int a[], int n,
int size)
{      int i;
       for(i=0; i<size-1;
i++)
       if((a[i]>=n) ||
(a[i]<0))
       return 0;
       return 1;}
int normalize(int a[],
int n)
{      int i;
       do{for(i=0; i<MAX-
1; i++)
       if(a[i]<0){
        a[i]+=n;
++a[i+1];}
       for(i=0; i<MAX-1;
i++)
        if(a[i]==n){
        a[i]=0; --
a[i+1];}
       while(!done(a,4,MA
X));
        return 0;}
int normalized2bits(int
bits[], int a[])
{      int i, digits;
       i=MAX-1;
       while(a[--i]==0) ;
digits=++i;
       for(i=0;
i<digits;i++){
        if(a[i]==0){
          bits[i*4]=0;
bits[i*4+1]=0;
          bits[i*4+2]=0;
          bits[i*4+3]=0;}
        else if(a[i]==1){
          bits[i*4]=1;
bits[i*4+1]=0;
          bits[i*4+2]=0;
          bits[i*4+3]=0;}
        else if(a[i]==2){
          bits[i*4]=0;
bits[i*4+1]=0;
          bits[i*4+2]=1;
          bits[i*4+3]=1;}
        else if(a[i]==3){
          bits[i*4]=1;
bits[i*4+1]=0;
          bits[i*4+2]=1;

bits[i*4+3]=1;}}
        i=4*MAX-1;
        while(bits[--
i]==0) ;     digits=++i;
        bits[MAX*4-
1]=digits;
             return
(digits);}
int
normalized2string(char
*s, int a[])
{      int i, digits;
       char  *lut[]=
       {"0000","1000","00
       11","1011"};
       char *temp, c;
temp=s;
       i=MAX-1;
       while(a[--i]==0) ;
digits=++i;
       for(i=0; i<digits;
i++){

sprintf(s,"%s",lut[a[i]]
);
        s+=4;}
       *s='\0';
       s=temp;
       for(i=0;i<digits*2
;i++) {
        c=s[i];
        s[i]=s[digits*4-
i-1];
        s[digits*4-i-
1]=c; }
        return digits;}
int int2base1j(int
bits[], int r)
{      int temp[MAX*4];
       int i;
       int
minus1[]={1,0,1,1,1};
       int a[MAX];
       clear(a, MAX);
       if(r==0){
clear(bits,MAX*4);
            return 0; }
       if(r>0){int2basen(
r, 4, a);

       basenegative(a);
```

Table 2: Equivalence between base –4 and base (–1–j) representations.

base –4	base (–1–j)
0	0000
1	0001
2	1100
3	1101

3.2 Conversion algorithm for imaginary integers[2]

To obtain binary representation of a given positive or negative imaginary number, we simply multiply (according to algorithm in Section 4.3) the corresponding (–1–j)-base representation of positive or negative integer with 111 (equivalent to j_{base10}) or 11 (equivalent to $-j_{base\ 10}$), as required.

Thus

$j55_{base10}$ = (0001 1101 0000 1101 1101) x (111)
= 0000 0001 0001 0000 0100 0011 $_{base\ -1-j}$
$-j55_{base10}$ = (0001 1101 0000 1101 1101) x (11)
= 0000 0111 0111 0100 0111 $_{base\ -1-j}$

3.3 Conversion algorithm for decimal fractions[3]

The procedure for finding the binary equivalent for fractions in base (–1–j) is based on the usual approach to obtaining ordinary binary representations. Any fraction F can be expressed uniquely in terms of powers of $½ = 2^{-1}$ such that $F = r_0 = f_1.2^{-1} + f_2.\ 2^{-2} + f_3.\ 2^{-3} + f_4.\ 2^{-4} + \ldots$ up to machine limit. Then the coefficients f_i and remainders r_i are given as follows:
Initially if $2r_0 - 1 < 0$ then $f_1 = 0$ and set $r_1 = 2r_0$ or if $2r_0 - 1 \geq 0$ then $f_1 = 1$ and set $r_1 = 2r_0 - 1$.
Then if $2r_i - 1 < 0$ then $f_{i+1} = 0$ and $r_{i+1} = 2r_i$ or if $2r_i - 1 \geq 0$ then $f_{i+1} = 1$ and $r_{i+1} = 2r_i - 1$

We continue this process until $r_i = 0$ or the machine limit has been reached. Then, for $\forall f_i = 1$, we replace its associated 2^{-i} according to Table 3 (only the first four values of i are listed in this table; for i>4, refer to Table 4).

Table 3: Equivalence between fractional coefficients and base (–1–j) representations.

i	2^{-i}	base (–1–j)
1	2^{-1}	1.11
2	2^{-2}	1.1101
3	2^{-3}	0.000011
4	2^{-4}	0.00000001

As an example, let $F = r_0 = 0.6875_{base10}$
Initially $2r_0 - 1 = 2(0.6875) - 1 = 0.375 > 0 \Rightarrow f_1 = 1$, $r_1 = 2r_0 - 1 = 2(0.6875) - 1 = 0.375$.
Then $2r_1 - 1 = 2(0.375) - 1 = -0.250 < 0 \Rightarrow f_2 = 0$, $r_2 = 2r_1 = 2(0.375) = 0.750$

[2] See Figure 1 for a program in C language to convert imaginary integers into (–1–j)-base complex binary number system.
[3] See Figure 2 for a program in C language to convert real/imaginary fractions into (–1–j)-base complex binary number system.

Figure 2: A program in C language for conversion of real and imaginary fractions to (–1–j)-base complex binary number system.

```
/* Program for Real
and Imaginary
*/
/* Fractions
Conversion to
*/
/* (-1-j)-base Complex
Binary Number    */ /*
System
*/
/* Author: Amer Al-
Habsi
*/

#include <stdio.h>
#include <stdlib.h>
#include "base1j.h"
#define FRACPART 80
#define INTPART (MAX-
FRACPART)
#define ARBIT 0

void print_frac(int
bits[], int left, int
right)
{  int i, j;

for(i=FRACPART+left-1;
i>=FRACPART;i--)
    printf("%i",
bits[i]);
    printf(".");
 for(i=FRACPART-1,
j=0; j<right; i--,
j++)
      printf("%i",
bits[i]);}
void neg_frac(int
bits[])
{  int temp[MAX*4];
   int i;
int
minus1[]={1,0,1,1,1,0,
0,0,0,0,0,0};
   clear(temp,MAX*4);

mult(minus1,bits,12,MA
X,temp);
   for(i=0;
i<MAX*4;i++)
    bits[i]=temp[i];}
int frac(double F, int
accumulator[])
{  int i, j;
   double r,
doublerm1;
   double save_F;
   int temp[MAX*4],
tempsum[MAX*4];
   int t, s, k;
   int f[FRACPART/4];
   save_F=F;
   if(F<0) F=-F;

clear(accumulator,4*MA
X);
      clear(temp,4*MAX
);

clear(tempsum,4*MAX);
   for(i=0;
i<FRACPART/4; i++)
f[i]=0;
   r=F;
   for(i=1;
i<FRACPART/4 &&
r!=0.0; i++){
      doublerm1=2.0*r-
1.0;

      if(doublerm1<0.0
){f[i]=0;
      r=doublerm1+1;}
      else{ f[i]=1;
r=doublerm1;}}

  clear(accumulator,
MAX);

if(f[1]==1){clear(temp
, MAX);

temp[FRACPART]=temp[FR
ACPART-
1]=temp[FRACPART-2]=1;
  add(accumulator,
temp, tempsum, MAX-
ARBIT);
  for(j=0; j<MAX; j++)

accumulator[j]=tempsum
[j];}

if(f[2]==1){clear(temp
, MAX);

temp[FRACPART]=temp[FR
ACPART-
1]=temp[FRACPART-
2]=temp[FRACPART-4]=1;
    add(accumulator,
temp, tempsum, 100);
  for(j=0; j<MAX; j++)

accumulator[j]=tempsum
[j];}

if(f[3]==1){clear(temp
, MAX);
    temp[FRACPART-
5]=temp[FRACPART-6]=1;
    add(accumulator,
temp, tempsum, MAX-
ARBIT);
  for(j=0; j<MAX; j++)

accumulator[j]=tempsum
[j];}
  for(i=4;
i<FRACPART/4; i++){
    if(f[i]==1){t=i%4;
s=i/4;
         k=8*s;
clear(temp, MAX);
         switch (t) {
    case 0:
temp[FRACPART-k]=1;
           break;
    case
1:temp[FRACPART-
k]=temp[FRACPART-k-
1]=temp[FRACPART-k-
2]=1; break;
    case 2:
temp[FRACPART-
k]=temp[FRACPART-k-
1]=temp[FRACPART-k-
2]=temp[FRACPART-k-
4]=1; break;
    case
3:temp[FRACPART-k-
5]=temp[FRACPART-k-
6]=1; break;}
     add(accumulator,
temp, tempsum, MAX-
ARBIT);
     for(j=0; j<MAX;
j++)

accumulator[j]=tempsum
[j];}}
```

Continuing according to the algorithm, we have

$2r_2 - 1 = 2(0.750) - 1 = 0.5 > 0 \Rightarrow f_3 = 1, r_3 = 2r_2 - 1 = 2(0.750) - 1 = 0.5$

$2r_3 - 1 = 2(0.5) - 1 = 0$ (STOP) $\Rightarrow f_4 = 1, r_4 = 0$

Thus $0.6875_{base\ 10}$ $= 1.2^{-1} + 0.\ 2^{-2} + 1.\ 2^{-3} + 1.\ 2^{-4}$
$= 1(1.11) + 0(1.1101) + 1(0.000011) + 1(0.00000001)$
$= 1.11001101_{base\ (-1-j)}$
(addition according to algorithm in Section 4.1)

It is likely that most fractions will not terminate (as our example) until the machine limit is reached. For example $0.351_{base\ 10} = 1.11011100110011000001110011 0\ldots_{base\ -1-j}$

In that case, it is up to the user to terminate the process when certain degree of accuracy has been achieved.

In general, to find binary representation of any 2^{-i}, express i as $4s + t$ where s is an integer and $0 \le t < 4$. Then, depending upon value of t, 2^{-i} can be expressed as given in Table 4. All rules for obtaining negative integer and positive/negative imaginary number representations in base (–1–j), as discussed in previous sections, are equally applicable for obtaining negative fractional and positive/negative imaginary fractional representations in the proposed new base.

Table 4: Equivalence between value of "t" and base (–1–j) representations.

t	base (–1–j)
0	0.0…(8s–1)zeroes followed by 1
1	0.0…(8s–1)zeroes followed by 111
2	0.0…(8s–1)zeroes followed by 11101
3	0.0…(8s+4)zeroes followed by 11

3.4 Conversion algorithm for floating point numbers

To represent a floating point positive number in the new base, we add the integer and fractional representations according to the addition rules given in Section 4.1. Once again, all rules for obtaining negative integer and positive/negative imaginary number representations in base (–1–j), as discussed in previous sections, are equally applicable for obtaining negative floating point and positive/negative imaginary floating point representations in the proposed new base.

For example

$55.6875_{base\ 10}$ $= 0001\ 1101\ 0000\ 1101\ 1101 + 1.11001101$
$= 0001\ 1101\ 0000\ 1100\ 0000.1100\ 1101_{base\ -1-j}$

And

$j55.6875_{base\ 10}$ $= 0001\ 1101\ 0000\ 1100\ 0000.1100\ 1101 \times 111$
$= 11000001000000.01110011_{base\ -1-j}$

Knowing the conversion algorithms, as described in the previous sections, the binary representation for any given complex number can be easily obtained, as shown by the following example:

$(55.6875 + j55.6875)_{base10}$ $= 0001\ 1101\ 0000\ 1100\ 0000.1100\ 1101_{base\ -1-j}$
$+ 11000001000000.01110011_{base\ -1-j}$
$= 0010\ 0011\ 1000\ 0011.1010\ 0010_{base\ -1-j}$

This can be verified to be equivalent to the given complex number by calculating:

$$(-1-j)^{13} + (-1-j)^{9} + (-1-j)^{8} + (-1-j)^{7} + (-1-j)^{1} + (-1-j)^{0} + (-1-j)^{-1} + (-1-j)^{-3} + (-1-j)^{-7}$$

4. Arithmetic operations for complex numbers

4.1 Addition

The binary addition of two complex numbers follows these rules: 0 + 0 = 0; 0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 1100. These rules are very similar to the traditional binary arithmetic except for the last case where when two 1s are added, the sum is zero and (instead of just one carry) two carries are

generated which propagate towards the two adjoining positions after skipping the immediate neighbor of the sum column. That is, if two numbers with 1s in position n are added, this will result in 1s in positions n+3 and n+2 and 0s in positions n+1 and n in the sum. Similar to the ordinary computer rule where 1+111 … (to limit of machine) =0, we have 11 + 111 = 0 [zero rule]. (See Section 4.3 for an example of addition).

4.2 Subtraction

The binary subtraction of two complex numbers follows these rules: 0 - 0 = 0 ; 0 - 1 = * ; 1 - 0 = 1; 1 - 1 = 0. Three of the four conditions listed in these rules are the same as for subtraction in ordinary binary system. For the case where 1 is subtracted from 0 (* case in the rules), the following algorithm applies:

Assuming our minuend is $a_n a_{n-1} a_{n-2} \ldots a_{k+4} a_{k+3} a_{k+2} a_{k+1} a_k \mathbf{0} a_{k-1} \ldots a_3 a_2 a_1 a_0$ and subtrahend is $b_n b_{n-1} b_{n-2} \ldots b_{k+4} b_{k+3} b_{k+2} b_{k+1} \mathbf{1} b_{k-1} \ldots b_3 b_2 b_1 b_0$. Then, the result of subtracting 1 from 0 is obtained by changing $a_k \rightarrow a_k + 1$, $a_{k+1} \rightarrow a_{k+1}$ (unchanged) , $a_{k+2} \rightarrow a_{k+2} + 1$, $a_{k+3} \rightarrow a_{k+3} + 1$, $a_{k+4} \rightarrow a_{k+4} + 1$ and $b_k \rightarrow 0$.

Example: Subtract (2–3j) from 3

Solution: In base (–1–j) notation, we have

3 – (2–3j) = 1101 – (1100 – 0111 0111) ≡ 1101 – 1011 (by algorithm) = 0100 – 0010
= (0100 + 111010) – 0000 (by algorithm)
= 111110 = (1+3j)

4.3 Multiplication

The multiplication process of two complex binary numbers is similar to multiplication of two ordinary binary numbers except that while adding the intermediate results of multiplication, the new rules for addition, as given in Section 4.1, should be followed. The zero rule plays an important role in reducing the number of summands resulting from intermediate multiplications.

Example: Multiply (2–j3)(1+j3)

Solution: The binary representations of the given complex numbers in base (–1–j) are: (using Table 1)

$2–j3 = 1100 + (1110111) = 1011_{base\ –1–j}$
$1+j3 = 0001 + 110011 = 111110_{base–1–j}$

Now (2–j3)(1+j3) = 1011 x 111110

```
                111110
                  1011
==========================
                1~~111~~10
               ~~11111~~0
              000000
             111~~11~~0
============================
           111100010
```

Bold-faced **1**s help us in recognising the pattern 111 + 11 which results in 0 (zero rule).

For verificaton

$$111100010 = (–1–j)^8 + (–1–j)^7 + (–1–j)^6 + (–1–j)^5 + (–1–j)^1 = 11 + j3$$

4.4 Division

The division algorithm is based on determining the reciprocal of the divisor (denominator) and then multiplying it with the dividend (numerator) according to the multiplication algorithm given in Section 4.3.

Thus

$$(a + jb) \div (c + jd) \quad = \quad (a+jb)(c+jd)^{-1} \quad = \quad (a+jb)z \tag{3}$$

where $z = w^{-1}$ and $w = c + jd$

We start with our initial approximation of z setting $z_0 = (-1-j)^{-k}$ where k is obtained from the representation of w such that

$$w \equiv \sum_{i=-\infty}^{k} a_i \, (-1-j)^{-i} \tag{4}$$

in which $a_k \equiv 1$ and $a_i \equiv 0$ for $i>k$.

The successive approximations are then obtained by $z_{i+1} = z_i \, (2 - wz_i)$. If the values of z do not converge, we correct our intial approximation to $z_0 = -j(-1-j)^{-k}$ which will definitely converge (Blest and Jamil, 2001). Having calculated the value of z, we just multiply it with (a+jb) to obtain the result of the division. In the following examples, for the sake of clarity, we have used decimal numbers to explain the converging process of the division algorithm.

Let (a+jb) = 1 + j2, and w = 1+j3. Our calculations for approximation of $z = w^{-1}$ then begin as follows:

$1 + j3 = 0001 + 110011 = 111110_{base-1-j}$

$= 1.(-1-j)^5 + 1.(-1-j)^4 + 1.(-1-j)^3 + 1.(-1-j)^2 + 1.(-1-j)^1 + 0.(-1-j)^0 \Rightarrow k = 5$

Therefore

$z_0 = (-1-j)^{-5} = 0.125-j0.125$
$z_1 = 0.15625 -j0.21875$
$z_2 = 0.100208989 -j0.299802410$
$z_3 = 0.1000000024 -j0.3000000010$
$z_4 = 0.10000000 -j0.30000000$
$z_5 = 0.1 - j0.3$
$z_6 = 0.1 - j0.3$ (converging)

Now

$0.1 - j0.3 = 0.0111111111111111\ldots_{base-1-j}$

So $(1+j2) \div (1+j3) = (1+j2) \times (1+j3)^{-1}$

$= 0101_{base-1+j} \times 0.0111111111111111\ldots_{base-1-j}$

$= 1.0001001001001\ldots_{base-1-j} = 0.7 - j0.1$

As another example, let $w = -28-j15$, then

$-28-j15 = 111100010111_{base-1-j}$

$= 1.(-1-j)^{11} + 1.(-1-j)^{10} + 1.(-1-j)^9 + 1.(-1-j)^8 + 0.(-1-j)^7 + 0.(-1-j)^6$
$+ 0.(-1-j)^5 + 1.(-1-j)^4 + 0.(-1-j)^3 + 1.(-1-j)^2 + 1.(-1-j)^1 + 1.(-1-j)^0 \Rightarrow k = 11$

We begin by choosing

$z_0 = (-1-j)^{-11} = 0.15625 + j0.15625$
$z_1 := 0.239 + j0.0449$
$z_2 := -0.249 + j0.128$
$z_3 := -0.398 - j0.160$
$z_4 := 1.014 + j5.235$
$z_5 = -895 - j87.9$
(not converging)

So we correct our initial approximation to:
$z_0 = -j(-1-j)^{-11} = -0.015625 + j0.015625$
$z_1 := -0.02393 + j0.0176$
$z_2 := -0.0279 + j0.01556$
$z_3 := -0.0278 + j0.01486$
$z_4 := -0.02775 + j0.014866$
(converging)

The converging value of z_4 can be represented in base (–1–j) and then multiplied with any given complex number to obtain the result of dividing the given complex number by –28–j15, as in previous example.

5. Conjugate and magnitude of complex numbers

Beyond the rules for real arithmetic, complex numbers arithmetic may require the calculation of conjugates and magnitudes as well. Thus, if

$$w_{base\ -1-j} = (c + jd)_{base\ -1-j} = \sum_{i=-\infty}^{k} a_i (-1-j)^i \tag{5}$$

then, since –j(–1–j) = –1+j, the complex conjugate of w is given by

$$w^{\sim}_{base\ -1-j} = (c - jd)_{base\ -1-j} = \sum_{i=-\infty}^{k} a_i (-j)^i(-1+ j)^i \tag{6}$$

which gives $c = \sum_{i=-\infty}^{k} a_i^{\sim}(-1-j)^i$

where $a_i^{\sim} = \frac{1}{2}(1-j^i)\, a_i$ and for i = 0 (mod 4), $a_i^{\sim} = a_i$; i = 1 (mod 4), $a_i^{\sim} = 1110.1a_i$; i = 2 (mod 4), $a_i^{\sim} = 0$; and for i = 3 (mod 4), $a_i^{\sim} = 1.1a_i$

The imaginary part of the conjugate d can be calculated as j (c – w).
As an example, let w = 1 + j3, then its conjugate $w^{\sim}$ will be of the form c – jd, where c and d are calculated as follows:

In base (–1–j), $w = 1+j3 = a_5a_4a_3a_2a_1a_0 = 111110_{base\ -1-j}$ then
$a_0^{\sim} = a_0 = 0$
$a_1^{\sim} = 1110.1a_1 = 1110.1 \times 1 = 1110.1$
$a_2^{\sim} = 0$
$a_3^{\sim} = 1.1a_3 = 1.1 \times 1 = 1.1$
$a_4^{\sim} = a_4 = 1$
$a_5^{\sim} = 1110.1a_5 = 1110.1 \times 1 = 1110.1$

Thus
$c = a_5^{\sim}(-1-j)^5 + a_4^{\sim}(-1-j)^4 + a_3^{\sim}(-1-j)^3 + a_2^{\sim}(-1-j)^2 + a_1^{\sim}(-1-j)^1 + a_0^{\sim}(-1-j)^0$
$= 1110.1(-1-j)^5 + 1(-1-j)^4 + 1.1(-1-j)^3 + 0(-1-j)^2 + 1110.1(-1-j)^1 + 0(-1-j)^0$
$= 000000001_{base\ -1-j}$
And
$d = j(c - w) = 111(000000001 - 111110)$
$= 01101_{base\ -1-j}$

Now –jd = 11 x 1101 = 1110111
Thus $w^{\sim} = c - jd = 1 + 1110111 = 1010_{base\ -1-j} = 1 - j3$
To calculate the square of magnitude of w, we have
$|w|^2 = ww^{\sim} = 111110 \times 1010 = 111001100_{base\ -1-j}$
To verify $(1+j3)(1-j3) = 10_{base\ 10} = 111001100_{base\ -1-j}$

6. Summary and Conclusion

We have described conversion algorithms and arithmetic procedures for a (–1–j)-base binary number system which allows given complex numbers to be represented as one unit. This is expected to facilitate equal opportunity representation to complex numbers and, hence, simplify their operations in today's microprocessors. Currently, work is underway to design a hardware arithmetic unit based on algorithms presented in this paper and then it will be implemented using Field Programmable Gate Arrays.

References

BLEST, D. and JAMIL, T. Efficient division in a binary representation for complex numbers. *Proceedings of the IEEE Southeastcon 2001*. Clemson, South Carolina. 30 March-1 April 2001. USA.

GEPPERT, L. 1999. The 100-million Transistor IC. *IEEE Spectrum*. **36: 7**: 23-60.

JAMIL, T., HOLMES, N., and BLEST, D. 2000. Towards implementation of a binary number system for complex numbers. *Proceedings of the IEEE SoutheastCon 2000*. Nashville, Tennessee. 7-9 April 2000. USA

KNUTH, D.E. 1960. An Imaginary Number System. *Communications of the ACM*. **3**: 245-247.

PENNEY, W. 1964. A Numeral System with a Negative Base. *Mathematics Student Journal*. **11(4):** 1-2.

PENNEY, W. 1965. A Binary System for Complex Numbers. *Journal of the ACM*. **12(2):** 247-248.

STEPANENKO, V.N. 1996. Computer Arithmetic of Complex Numbers. *Cybernetics and System Analysis*. **32(4):** 585-591.

Received 8 June 2001
Accepted 7 November 2001