A Simple Circuit for Adding Complex Numbers

JOHNNY GOODE[†], TARIQ JAMIL[‡], AND DALE CALLAHAN[†]

[†]Department of Electrical and Computer Engineering,
University of Alabama-Birmingham (USA)
[‡]Department of Electrical and Computer Engineering,
Sultan Qaboos University (OMAN)

Abstract: - The important role of complex numbers in a wide range of engineering applications demands better and more efficient methods of handling arithmetic operations involving these numbers. Using base (-1+j), instead of base 2, to represent complex numbers in binary notation allows both real and imaginary parts of the number to be combined into single binary representation and facilitates reduction in the number of arithmetic operations. Design of a size-free adder circuit based on (-1+j)-base binary representation of complex numbers is presented in this paper.

Key-Words: - Complex number, Complex binary number, Arithmetic, Adder, State machine, Token

1 Introduction

Complex numbers play a very important role in engineering applications such as digital signal processing and image processing. Thus design of an efficient approach to handle complex arithmetic will result in better performance in such applications. These days, complex number operations involve, to a large extent, application of a "divide-and-conquer" technique, whereby a complex number is broken up into its real and imaginary parts. Operations are then carried out on each part as if it were a real part of the 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 (one for the real parts and one for the imaginary parts) while multiplication of the same two complex numbers requires four multiplications, one subtraction, and one addition. 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. Efforts in defining binary numbers with bases other 2, which facilitates one-unit representation of complex numbers, includes work by Knuth [1], Penney [2], and Stepanenko [3]. Jamil *et. al.* [4] have presented a detailed analysis of (-1+j)-base complex binary number system and elaborated on how addition, subtraction, multiplication, and division of two such complex binary numbers can be accomplished. Designs of nibble-size adders for complex binary numbers, based on traditional minimum-delay and carry look-ahead principles, and the performance of these circuits compared to base 2 adders have been presented in Ref.[5]. In continuation of these efforts, in this paper, we are presenting a novel design of a complex binary adder which does not impose any limit on the size of the operands.

This paper is organized as follows: In Section 2, we provide a review of the complex binary number system and describe how the addition of these complex binary numbers is accomplished. This is followed by our design of the complex binary adder in Section 3. In Section 4, we'll present our conclusion and outline future work in this area of computer arithmetic.

2 (-1+j)-Base Complex Binary Number

The value of an n-bit binary number with base (–1+j) can be written in the form of a power series as follows: $a_{n-1}(-1+j)^{n-1}+a_{n-2}(-1+j)^{n-2}+...+a_1(-1+j)^1+a_0(-1+j)^0$ 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). This is analogous to the ordinary binary number system power series of: $a_{n-1}(2)^{n-1}+a_{n-2}(2)^{n-2}+\ldots+a_1(2)^1+a_0(2)^0$ except that the bases are different. Using the algorithms, given in Ref.[4], we are able to represent a given complex number with single complex binary number as shown below:

2004 + j2004
= $11101000000011101110011000000_{base(-1+j)}$
This can be verified by computing the power series
$(-1+j)^{27} + (-1+j)^{26} + (-1+j)^{25} + (-1+j)^{23} + (-1+j)^{15} + (-1+j)^{14} + (-1+j)^{13} + (-1+j)^{11} + (-1+j)^{10} + (-1+j)^9 + (-1+j)^6 + (-1+j)^5 =$ 2004 + j2004

The binary addition of two complex binary numbers follows these rules: 0 + 0 = 0;0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 1100. 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. Furthermore, 11 + 111 = 0 [Zero Rule].

3 Complex Binary Adder Design

The design of the adder is based on using a state machine to store the logic details rather than designing the addition and carry operations with discrete components. This approach results in a very simple circuit implementation. The entire adder consists of a few gates to add single bits from the input numbers, memory to hold the state and output information, and a register to store the current state (in effect the carry to the next addition). Since operations are done bit by bit the adder itself imposes no limitations on the sizes of the numbers to be added. The state machine is not aware of the number of bits in the input numbers. The only requirement is to make sure that the inputs are sufficiently padded with high order zeros to allow for the carry from the addition of the high order bits of the input numbers. Depending on the carry-in from the previous addition and the values of the two current bits to be added, a carry of up to 8 bits may result. Consequently 8 bits of padding of high order zeros would be required to correctly complete the addition. An implementation of the adder was done in SystemView, a software modeling package.

3.1 The State Machine

The logic of the adder is stored in a state table. Each entry of the table contains the next state of the state machine and the output from the last addition operation. The table is organized into three sections, one for each state transition. A transition is determined by the result of the addition of the next two input bits. There are three results – (1) 0+0, (2) 0+1 or 1+0, and (3) 1+1. The result selects the particular section of the state table to use for determining the next state and output.

The input to the state machine is the sum of the current two binary bits to be added. The current state is composed of the carry out of the previous operation. The next state (and single bit output for the current addition) is found in the memory location formed by the concatenation of the input and current state bits as described above.

The state table is shown in Table 1 and the state diagram is given in Fig.1. The state table was constructed by: (i) Starting with a sum of 0 and a carry in (current state) of 0. (ii) Adding 0, 1 or 2. (The sum of input bits A and B). (iii) Shifting out the low order sum bit (The sum will be a single binary bit, 0 or 1 and a carry. The result of 0 plus 0 is 0 with no carry, 0 plus 1 is 1 with no carry, and 1 plus 1 is 0 with a carry of 110.). (iv) Repeating the above until all possible results were produced.

It can be seen that there are 15 states (representing carries from a previous operation) and 3 inputs per state resulting in 45 state transitions. For each transition the output is 1 or 0. For the purposes of this implementation, an additional redundant state was added simply to fill the 16 memory locations that were available. This additional state serves no other purpose and could be deleted with no effect on the operation of the adder.

It should be noted that the adder does not actually add inputs A and B to a previous carry operation. It only changes state based on the current input and current state. The states represent the result of addition operations. The actual result (0 or 1) of the addition of the two current input bits is stored in the state memory and output as the state changes.

Following is a description of the columns of the State Table (Table 1):

Column Number

(1) Memory Location (Current State). Memory is arranged in three banks. Each bank contains 16 locations, 00-15 representing the current state. In a physical implementation various combinations of memory sizes could be used. For example a single 48-location memory or 3 16-location memories could be used. Column (1) is the memory location number in decimal. The location is shown only for the first bank of memory. There are three rows of data for each location. Each row corresponds to a bank of memory. For example for location ‘00’, the first row corresponds to location ‘00’ for bank 1, the second row to location ‘00’ for bank 2 and the third row to location ‘00’ for bank 3.

(2) Contents (Hex). The contents of each memory location of each bank are shown in hexadecimal. For example, for memory location ‘06’, bank 1 contains x’04’ or binary 00000100. Bank 2 contains x’84’ or binary 10000100. These values represent the output and next state. The output bit is in the high order location (leftmost bit). The next state is in the 4 low order bits (4 rightmost bits). For example, memory location 13, bank1 contains x’8F’ or binary 10001111. The high order bit is 1 and is the output bit when the current state is 13 and 0000 (see column 3 and 4 explanation) is added via input bits A and B (both A and B are 0). The four low order

Table 1. Adder State Table

(1) Memory Location (Current State)	(2) Contents (Hex)	(3) Previous Carry	(4) Add	(5) Result	(6) After Shift	(7) Next State	(8) Output
00	00	0000	0000	0000	000	00	0
	80		0001	0001	000	00	1
	01		1100	1100	110	01	0
01	02	0110	0000	0110	011	02	0
	82		0001	0111	011	02	1
	05		1100	111010	11101	05	0
02	83	0011	0000	0011	001	03	1
	04		0001	1110	111	04	0
	84		1100	1111	111	04	1
03	80	0001	0000	0001	000	00	1
	01		0001	1100	110	01	0
	81		1100	1101	110	01	1
04	82	0111	0000	0111	011	02	1
	05		0001	111010	11101	05	0
	85		1100	111011	11101	05	1
05	86	11101	0000	11101	1110	06	1
	00		0001	0000	000	00	0
	80		1100	0001	000	00	1
06	04	1110	0000	1110	111	04	0
	84		0001	1111	111	04	1
	07		1100	111010010	11101001	07	0
07	88	11101001	0000	11101001	1110100	08	1
	09		0001	0100	010	09	0
	89		1100	0101	010	09	1
08	0A	1110100	0000	1110100	111010	10	0
	8A		0001	1110101	111010	10	1
	0B		1100	1000	100	11	0
09	03	0010	0000	0010	001	03	0
	83		0001	0011	001	03	1
	04		1100	1110	111	04	0
10	05	111010	0000	111010	11101	05	0
	85		0001	111011	11101	05	1
	0D		1100	111010110	11101011	13	0
11	0C	100	0000	0100	010	12	0
	8C		0001	0101	010	12	1
	0E		1100	111000	11100	14	0
12	03	0010	0000	0010	001	03	0
	83		0001	0011	001	03	1
	04		1100	1110	111	04	0
13	8F	11101011	0000	11101011	1110101	15	1
	02		0001	0110	011	02	0
	82		1100	0111	011	02	1
14	06	11100	0000	11100	1110	06	0
	86		0001	11101	1110	06	1
	00		1100	0000	000	00	0
15	8A	1110101	0000	1110101	111010	10	1
	0B		0001	1000	100	11	0
	8B		1100	1001	100	11	1

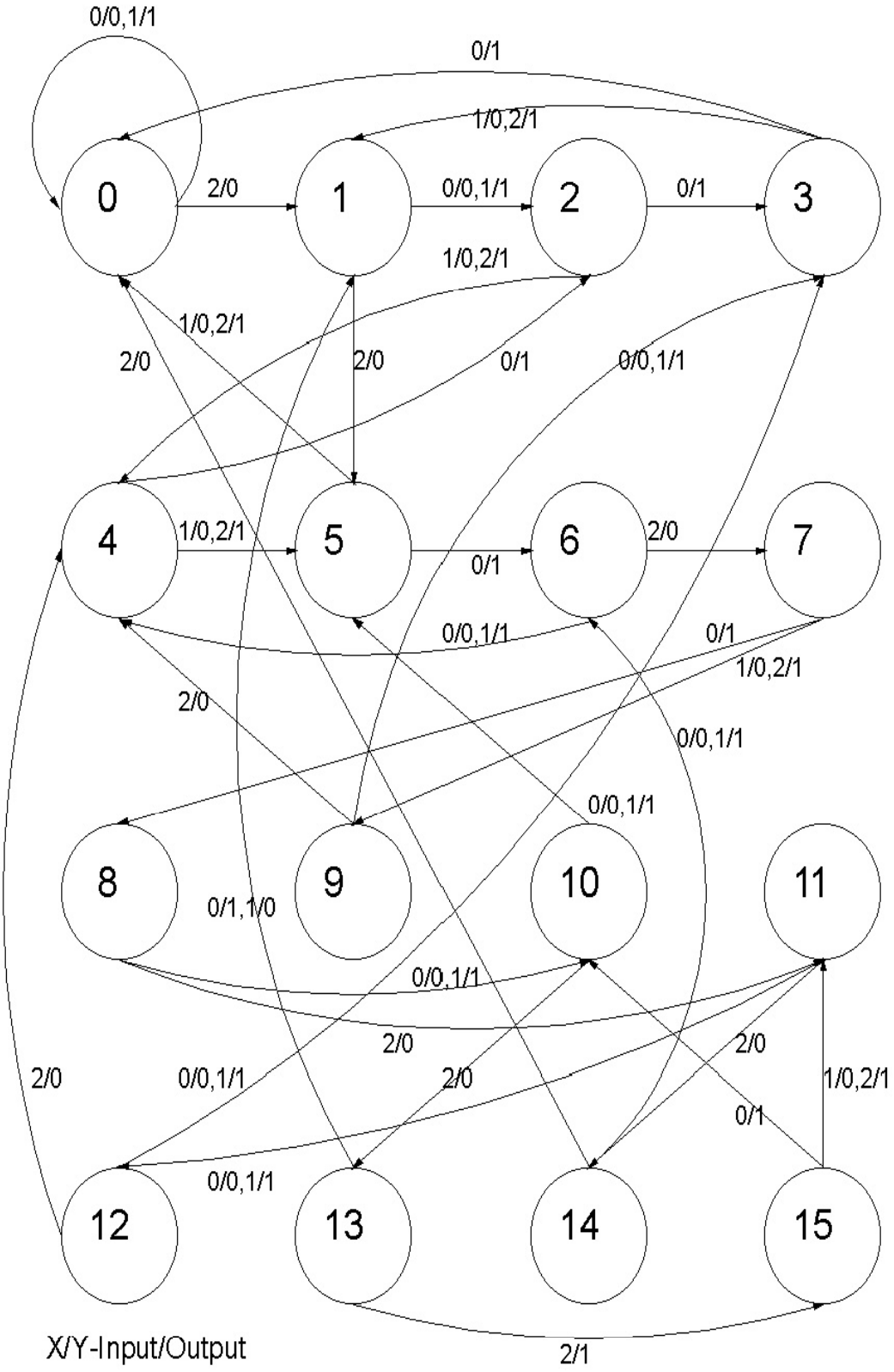

Fig.1. Adder State Diagram

bits ‘1111’ are the next state. The next state is shown in decimal in column 7. It corresponds to the four low order bits of column 2.
(3) Previous Carry. This column shows the carry from the result of the addition of the previous two bits. The memory location in column 1 is the representation of this value. The addition of input numbers A and B and the previous carry is done in the adder by considering the previous carry as the current state and the sum of A and B as selecting the particular state transition from the current state.
(4) Add. This column shows the sum of input bits A and B that is added to the previous carry. The sum actually results in a selection of the memory bank to use for finding the next state. If the sum is 0000 (both A and B are 0), bank 1 is used. If the sum is 0001 (either A or B is one), bank 2 is used. If the sum is 1100 (both A and B are 1), bank 3 is used. The value of the previous state is used to access the memory location of the selected bank to get the next state and output bit.
(5) Result. The result of adding columns (4) and (3) is shown. This addition is not actually performed by the adder. This result after shifting right one bit is assigned a state number to be used for the next operation. The result after shifting is in column (6). The state number is shown in column (7).
(6) After Shift. This column shows the result of shifting column 5 right one bit. This is the operation that would be performed to get ready to add the next two values of input bits A and B. The value is assigned a state number (column 7) that is the next state (column 1).
(7) Next State. This number represents the value in column (6). It is the memory location that is used in the next decoding operation. The sum of input bits A and B select the memory bank to use. Column (6) addresses the location within the bank.
(8) Output. This is the bit that is output as a result of adding A, B and the carry operation from the previous addition operation.

Using the diagram to add 1 and 1 will be instructive. Assuming the initial state is 0, the addition of input bit A=1 and B=1 results in an input of $2_{base\ 10}$, output of 0 and next state of 1. The next values of A and B are 0, so the next input is 0, output is 0 and next state is 2. Again A and B are 0, the input is 0, the output now is 1 and the next state is 3. Again A and B are 0, the input is 0, the output is again 1 and the next state is 0. (The output to this point is $1100_{base\ -1+j}$ which is $2_{base\ 10}$ as it should be). A and B are 0 for the rest of the 32 bit sequence. The input is 0, the output is 0 and the next state is 0 for all remaining input bits. The same sequence can be followed in the state table.

3.2 Implementation and Performance

A functional diagram of the adder is shown in Fig. 2. Data are input by storing two complex binary base numbers in the input memories. The numbers are shifted serially, least significant bit first into the single bit adder section. As discussed previously the adder merely selects a memory bank to use based on the values of the input bits (00, 01 or 10, or 11). The 'state and output' memory holds the state values and output for a given state transition. The result is shifted into the output shift registers and the carry is saved in a register for addition to the next two input bits. A master clock synchronizes all operations. An implementation (summarized below) allows addition of two 32-bit numbers. No provision was made for carries beyond 32 bits. In the earlier discussion of the adder state machine it was noted that a carry out value could be up to 8 bits. So, if the input numbers are limited to 24 bits, a maximum size carry could always be handled.

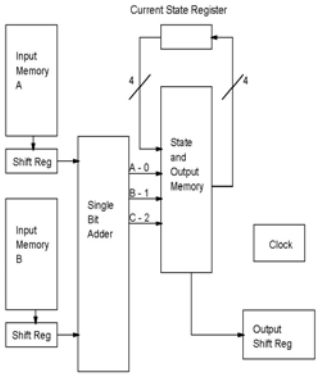

Fig.2. Adder Functional Diagram

Addition of numbers of any length could be accommodated by expanding the input memory and output shift register length. The adder itself is a single bit adder. An actual implementation in a digital computing system would use some other form of input and output such as general-purpose registers. Also, provision for carry beyond the maximum register length would be made. Other specialized subsystems for doing multiplication, division and floating point operations could of course be designed. In short there is no limitation to the length of numbers that can be added other than that imposed by the input and output devices.

An implementation of the adder was accomplished using SystemView[6], a modeling and simulation tool that provides tokens representing memories, gates and other devices from which systems may be constructed.

The results of output operations can be plotted using 'sink' tokens. For example the display of the results of an addition shifted into the output shift register is shown in Fig. 3. On the plot of Token 57 data, the results are read right to left (least significant bit of the sum on the left). The time scale of course is arbitrary.

In a physical implementation of the circuit, the main component of delay and thus a limit on performance would typically be the time required to address the memory with the results of the addition operation and store the result in the 'current state' register. Otherwise the addition speed would be limited only by the speed of the logic gates at input and output.

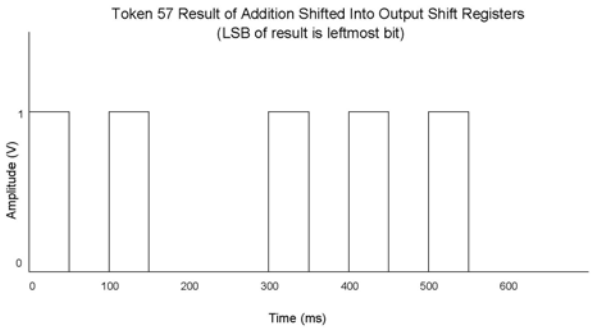

Fig. 3. Addition result shifted into output shift registers

3.3 Operation

In the SystemView implementation numbers to be added (for example 2 and -j $_{base\ 10}$) are placed in the input memories. In base(-1+j), 2 is 1100 or xC in hexadecimal. In base(-1+j), -j is 111 or x7 in hexadecimal. So, xC is entered in one memory and x7 is entered in the other. The result can be viewed from the plot of Token 57 (Fig.3). Reading from right to left the result is $111011_{base\ -1+j}$. An expansion of this number as coefficients of powers of (-1+j) shows it to be 2-j.

4 Conclusion and Future Research

Despite their usefulness in a wide range of engineering applications, operations involving complex numbers have been carried out inefficiently for a long time. A number of schemes have been proposed to allow a one-unit representation of complex numbers but design of a size-free adder unit based on any of those schemes has not yet been implemented. In this paper, we have presented design and implementation of a size-free adder based on (-1+j)-base complex binary number system. Work needs to be continued in this arena to compare the performance and speed of this adder with the base-2 adder designs so that, based on the application desired for a given processor, a suitable type of adder unit can be designed within the system.

References:

[1] D. Knuth: 'An Imaginary Number System', *Communications of the ACM*, pp. 245-247,1960.

[2] W. Penney:, 'A Binary System for Complex Numbers', *Journal of the ACM*, pp. 247-248, April 1965.

[3] V. Stepanenko: 'Computer Arithmetic of Complex Numbers', *Cybernetics and System Analysis*, Vol. 32, No. 4, pp. 585-591, 1996.

[4] T. Jamil, N. Holmes, and D. Blest:`Towards Implementation of a Binary Number System for Complex Numbers', *Proceedings of the IEEE SoutheastCon 2000*, Nashville, Tennessee (USA), pp. 268-274, April 2000.

[5] T.Jamil, B. Arafeh, and A. AlHabsi: 'Hardware Implementation and Performance Evaluation of Complex Binary Adder Designs," *Proceedings of the 7th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2003*), Orlando, Florida (USA), Vol. II, pp. 68-73, July 2003.

[6] http://www.elanix.com