High Performance Signed-Digit Decimal Adders

Jeff Rebacz, Erdal Oruklu, and Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, Illinois 60616-3793
Email: jrebacz@iit.edu, erdal@ece.iit.edu, sansonic@ece.iit.edu

***Abstract*—Decimal arithmetic is desirable for high precision requirements of many financial, industrial and scientific applications. Furthermore, hardware support for decimal arithmetic has gained momentum with the revision of the IEEE 754 standard. This paper presents a new scheme for carry-free decimal addition using a signed-digit representation. In order to simplify the hardware requirements, each signed decimal digit uses a two's complement representation instead of complex representations found in other signed-digit decimal arithmetic implementations. Flagged adder speculation is used for fast addition of the constants required in the correction step. The proposed scheme is compared to existing signed-digit decimal adders. Each architecture is synthesized on 0.18μm technology for comparison in area, delay and power. The results show that both operation speed and area usage can be significantly improved with respect to existing signed-digit decimal adders.**

I. Introduction

The translations between decimal and finite binary numbers are lossy by their very nature. These losses impact the accuracy of any arithmetic result if they cause a rounding error. For example, if a 5% tax on a $0.70 telephone call is computed with IEEE 754 double precision floating-point numbers, the result would be $0.73499999999999999 and will be $0.73 after rounding to the hundreths place [1]. If a decimal floating-point system is used, the correct result of $0.735 would be obtained and could be correctly rounded to $0.74. If such decimal systems are not used, these tiny losses might accrue or multiply into a loss more substantial than $0.01.

Software packages have been available for most programming languages so that decimal numbers could be evaluated with decimal arithmetic to avoid error [2] [3]. Until recently, this has been the de facto solution; hardware based decimal arithmetic implementations can now be found in some general purpose processors. In particular, IBM recently incorporated a decimal floating-point arithmetic unit in the Power6 processor [4]. A compelling reason to do such is a report [5] showing that 55% of the numbers stored in the databases of 51 major organizations are decimal. If software implemented decimal arithmetic takes 100 to 1000 times longer than a hardware version [1], then decimal floating-point units can clearly impact performance.

Several architectures have been proposed for efficient hardware realization of decimal arithmetic operations. Decimal renditions of binary carry-save [6] [7] and carry-lookahead [8] [9] adders have been proposed. New decimal multipliers [10] [11] and dividers [4] have also been proposed. The main motivation behind this paper is to design a signed-digit decimal adder with improved throughput and area characteristics for rapid adoption into general purpose processors and embedded devices.

II. Theory

The decimal signed-digit set is -9 to 9 inclusive. In conventional signed-digit implementations, to add two signed-digit decimals x_i and y_i, three quantities must be found: the intermediate sum u_i, the carry c_i and the correction $-10 * c_i$. The intermediate sum u_i is $x_i + y_i$ and ranges from -18 to 18 inclusive. The carry c_i is generated using a rule set such that $-8 \leq u_i - 10 * c_i \leq 8$. A rule set that satisfies this restriction is to equate c_i with -1 when u_i is less than -1; equate c_i with 1 when u_i is greater than 1; equate c_i with zero otherwise [12]. The corrected sum $u_i - 10 * c_i$ can then be added with the previous digit's carry. Note that the corrected sum is between -8 and 8 inclusive, so an input carry of -1 or 1 will not cause a carry to propagate to the next decimal digit. The above is expressed in Equations (1) - (3).

$$u_i = x_i + y_i \tag{1}$$

$$s_i = u_i - 10 * c_i + c_{i-1} \tag{2}$$

$$c_i = \begin{cases} -1 & \text{if } u_i < -1 \\ 1 & \text{if } u_i > 1 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

It is important to see the hardware implications of these equations. Carry-free addition is gained with the expense of two or three levels of adders. One adder level is needed to find the intermediate sums u_i. The remaining number of adders, one or two, are dependent on whether or not $c_{i-1} - 10 * c_i$ is precalculated for all possible combinations of c_i and c_{i-1}.

A. Addition Example

An example demonstrates how this technique implements carry-free addition. In the addition of 313528 and 3478, a carry ripples through 3 digits.

$$\begin{array}{cccccccc} & & & 1 & 1 & 1 & & \\ & 3 & 1 & 3 & 5 & 2 & 8 & \\ + & & & 3 & 4 & 7 & 8 & \\ \hline & 3 & 1 & 7 & 0 & 0 & 6 & \end{array}$$

Using the signed-digit technique, carries are still produced, but they never propagate. In other words, a digit's carry out is not a function of its carry in.

978-1-4244-3355-1/09/$25.00©2009 IEEE

	3	1	3	5	2	8	
+			3	4	7	8	
	3	1	6	9	9	16	u vector
	−10	0	−10	−10	−10	−10	correction vector
1	0	1	1	1	1		c vector
1	−7	2	−3	0	0	6	sum vector

The general algorithm is as follows:

1) For each digit, add the two operands to get the intermediate sum, u_i. The range for u_i is -18 to 18 inclusive.
2) If $u_i > 1$, set $correction_i = -10$ and $c_i = 1$. If $u_i < -1$, set $correction_i = 10$ and $c_i = -1$. In the example, the c vector is shifted one digit position to the left for addition.
3) Find the sum of the three vectors, u, $correction$ and c. This operation, as those before, will not propagate carries.

III. Prior Work in Signed-Digit Decimal Addition

In the past, several architectures based off of signed-digit (SD) concepts have been proposed. The Svoboda adder [13] was an early design that added digits from -6 to 6 inclusive. The Svoboda adder used an SD code where 5-bits represented numbers as multiples of three. This code helped simplify addition. However, the architecture uses two binary adders with end-around-carries. Also, converting BCD operands into the Svoboda code requires significant overhead. Although the Svoboda adder is quite old, it has been used in a recent, novel decimal multiplier using signed-digit partial products [10].

The Redundant Binary Coded Decimal adder (RBCD) [14] [15] is faster and more efficient than the Svoboda. This implementation adds 4-bit wide signed-digits between 7 and -7 inclusive. The reduced digit set dramatically simplifies carry detection at the expense of overhead required to conform BCD operands to the adder's digit range.

Another SD architecture is the speculative SD in [12]. This architecture is rather complex as it implements a clever speculation technique that greatly facilitates the addition of input carries. The input operands are two SD decimal numbers with each digit represented by one positive 4-bit vector x^+ and one negative 4-bit vector x^-. For example, the addition of -3 and 9 may be performed like this (let the two vectors be expressed as (x^+, x^-)):

$$\underbrace{(0010, 0101)}_{(2-5=-3)} + \underbrace{(1010, 0001)}_{10-1=9} = \underbrace{(00101000, 00011100)}_{28-22=6}$$

The SD decimal adder proposed in this paper is most alike to the RBCD adder. The key differences lie in the digit set, the carry detection circuit, and the correction method, all of which will be presented and explained in the next section.

IV. Proposed Signed-Digit Decimal Adder

In the proposed signed-digit architecture, the digit set used is -9 to 9 inclusive and is represented using a conventional 5-bit, two's complement vector. For the digit at position i, x_i and y_i are added to yield a 6-bit wide intermediate sum u_i (the carry-propagate adder (CPA) chosen in these designs uses carry lookahead logic). Then, two levels of simple logic determine the carry c_i, which uses positive and negative magnitude components to represent {-1, 0, 1}: c_i^- and c_i^+.

TABLE I
Correction Vector Generation

C_i^+	C_i^-	C_{i-1}^+	C_{i-1}^-	Correction Vector
0	0	0	0	00000_2 (0_{10})
0	0	0	1	11111_2 (-1_{10})
0	0	1	0	00001_2 (1_{10})
0	1	0	0	01010_2 (10_{10})
0	1	0	1	01001_2 (9_{10})
0	1	1	0	01011_2 (11_{10})
1	0	0	0	10110_2 (-10_{10})
1	0	0	1	10101_2 (-11_{10})
1	0	1	0	10111_2 (-9_{10})

$$c_i = c_i^+ - c_i^- \tag{4}$$

Simplicity is owed to a rule set for c_i that differs from (3) by ensuring that all intermediate sums over 7 will generate a positive carry and all intermediate sums below -8 will generate a negative carry. The main advantage is the reduced logic complexity. The new rule set can be implemented as a boolean function of 3 variables with 4 minterms as opposed to a boolean function of 6 variables with 9 minterms (for the ruleset in Equation 3). This ruleset is defined in Equation (5).

$$c_i = \begin{cases} -1 & \text{if } u_i < -8 \\ 1 & \text{if } u_i > 7 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

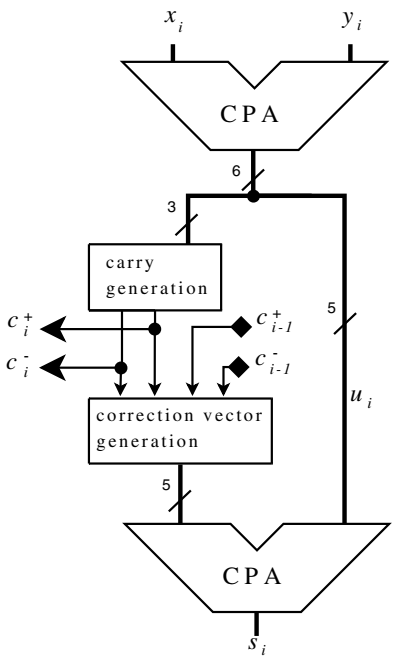
Each digit's positive and negative carry signals are easily calculated with Equations (6) - (7). With these, Equation (5) translates to the hardware implementation shown in Figure 1:

$$c_i^+ = \overline{u_i[5]}\ AND\ (u_i[4]\ OR\ u_i[3]) \tag{6}$$

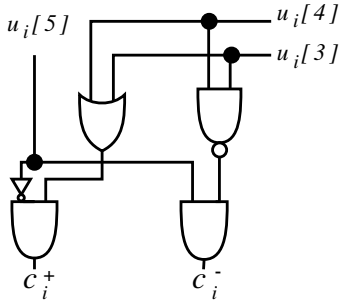
$$c_i^- = u_i[5]\ AND\ (u_i[4]\ NAND\ u_i[3]) \tag{7}$$

After u_i and c_i are found, the correction vector must be found and added. The correction vector corresponds to $-10{*}c_i$ from Equation (2). However, since the incoming carry from the previous digit, c_{i-1}, must eventually be added to u_i as well, it is convenient to think of the correction vector as $-10{*}c_i{+}c_{i-1}$. The correction vector (which is simply the constant term to be added) is tabulated for every possible input combination in Table I. Addition between u_i and this correction vector will result in s_i.

The proposed SD decimal adder is shown in Figure 1a for one SD digit. The carry generation block of the proposed adder is shown in Figure 1b. It is apparent in this figure that carries only propagate to the next digit. There is no ripple effect since carry generation does not depend on the carry in. This design is a necessary precursor to the improved version in the next section.



(a)



(b)

Fig. 1. In (a), a high-level diagram depicts the 2-operand SD adder. The circuit in (b) elaborates the carry generation block in (a). Once u_i is obtained from the carry propagate adder, c_i^+ and c_i^- are calculated with a few gates.

A. Parallel Correction Vector Addition

The previous proposed adder's correction generation block and correction addition CPA can be replaced by a parallel speculative structure to reduce delay since the lower bits of u_i are available before the carries. In this parallel adder, fast addition of the eight constants (see Table I) is performed by two stages of (*fic*) sequences [16]. The method in [16] describes adding two variables and a constant. For this architecture, the method has been simplified for adding one variable and a constant.

The addition of $r + m$ (r is a variable and m is a constant) is performed by inverting the 'flagged' bits in r. The flag at position i, $f[i]$, is a function of $f[i-1]$ and $r[i-1]$. A ripple effect is created but is tolerated for two reasons: the logic being rippled through is small and the ripple only travels through a few bits.

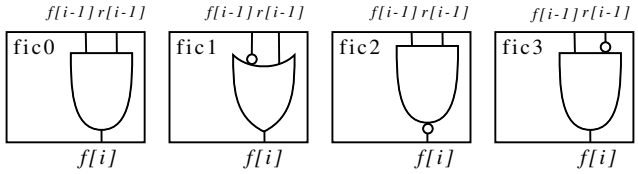


Fig. 2. The four flag inversion cells.

The units that find f are known as flag inversion cells (*fics*). There are only 4 types since there are 4 combinations of $m[i]$ and $m[i-1]$. These *fics* are illustrated in Figure 2.

Consider the addition of $m = 10$ to a 5-bit vector, r. First, the *fic* sequence must be found. This is done for each bit position i by examining $m[i]$ and $m[i-1]$ ($m[-1]$=0).

$$\textit{fic} \text{ at i} = \begin{cases} \text{fic0} & \text{if } m[i]=0 \text{ and } m[i-1]=0 \\ \text{fic1} & \text{if } m[i]=0 \text{ and } m[i-1]=1 \\ \text{fic2} & \text{if } m[i]=1 \text{ and } m[i-1]=0 \\ \text{fic3} & \text{if } m[i]=1 \text{ and } m[i-1]=1 \end{cases}$$

So, $m = 10_{10} = 01010_2$ will result in this *fic* sequence:

fic[0]=fic0
fic[1]=fic2
fic[2]=fic1
fic[3]=fic2
fic[4]=fic1

The flag bits produced by the fic sequences must be XORed with r to invert the flagged positions and yield $r+m$. However, the same flags can be produced with less cells. Notice that the upper 3 *fic* cells are not necessary. Adding an even constant to a number means the LSB will be unchanged ($f[0] = 0$). If bit 0 is unaffected, adding 1 to $r[1]$ will always invert $r[1]$ ($f[1] = 1$). Since $f[2]$ is a function of $f[1]$ and $r[1]$, and since $f[1]$=1, a full *fic* is not needed. After a quick examination of fic1 in Figure 2, $f[2]$ is simply $r[1]$ when $f[1] = 1$. Figure 3 shows the reduced version for finding the sum s.

In the parallel adder, the intermediate sum, u_i, is fed to two reduced *fic* sequences for adding and subtracting 10. However, only the flags are calculated. A 3-to-1 multiplexer will select the flag vector for adding -10, 0 or 10 according to the selects: c_i^+ and c_i^-. The multiplexer's output is XORed with u_i to invert the flagged bits. This inversion yields $u_i - 10 * c_i$.

The remaining step is to add the incoming carry. Two more reduced *fic* sequences are used to add and subtract 1 from $u_i - 10 * c_i$. Another 3-to-1 multiplexer is used with XOR gates after it to invert another flagged set of bits and yield the sum.

The new parts of the parallel adder are shown in Figure 4. At first glance, breaking the addition up into two levels seems to sacrifice speed and area. However, the two levels involve a very small amount of logic. The fic sequences for adding 10 and -10 involve 2 cells; adding 1 and -1 involve 3 cells. Thus, in both levels this method is very fast and efficient for constant addition.

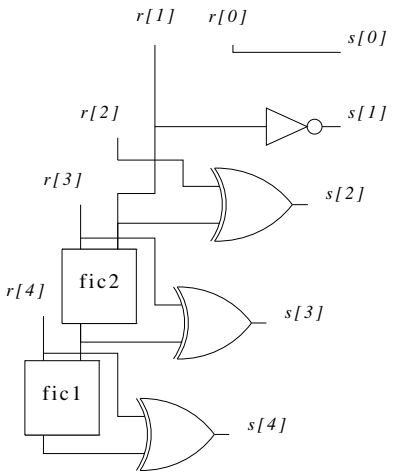


Fig. 3. The reduced flag inversion cell sequence for the addition of the constant 10 to a variable 5-bit vector r.

TABLE II
RANGES FOR MULTI-OPERAND ADDITION

Number of Operands	Range of u_i	Bounds of $u_i - 10 * c_i$
2 operands	[-18,18]	[-8,8]
3 operands	[-27,27]	[-7,7]
4 operands	[-36,36]	[-6,6]
5 operands	[-45,45]	[-5,5]
6 operands	[-54,54]	[-4,4]

V. MULTI-OPERAND ADDITION

The 2-operand signed-digit decimal adders can be used to add multiple numbers if arranged in a tree. In this way, corrections are made after every addition. However, it is apparent that immediate correction is not necessary. The correction step can be postponed in a similar manner as shown in [6] with the addition of new constraints to the problem (see Table II).

Note that 6 or more operands cannot be added without one or more intermediate corrections since it presents the hazard of accumulating 5 or more carries. To accumulate 5 carries in 6-operand addition without propagating to the next digit, the range of $u_i - 10 * c_i$ must be within -4 and 4 inclusive. This range cannot represent all decimal numbers. Therefore, only 2, 3, 4 and 5 operand signed-digit adders are possible with the present scheme.

VI. SYNTHESIS

In addition to the proposed decimal adders mentioned in this paper, the Svoboda adder, the RBCD adder, the speculative SD adder in [12], and a conventional BCD adder are implemented, tested, synthesized and compared against.

The conventional BCD adder uses a standard carry-propagate scheme with two binary adders. To generate a carry, simple combinational logic detects when the result of the first binary adder is over 9. When carries are generated, 6 is added to the first result using the second binary adder.

All designs were written in Verilog HDL. Synthesis results for area, timing and power were obtained from Synopsys Design Compiler using TSMC 0.18μm standard cell technology. Each design was compiled as an 8-digit (decimal) adder. The adders' input operands and output operands were latched, and the highest possible frequency was found through several aggressive t-optimizing compilation iterations. The period of this frequency is accurate to the hundredths place.

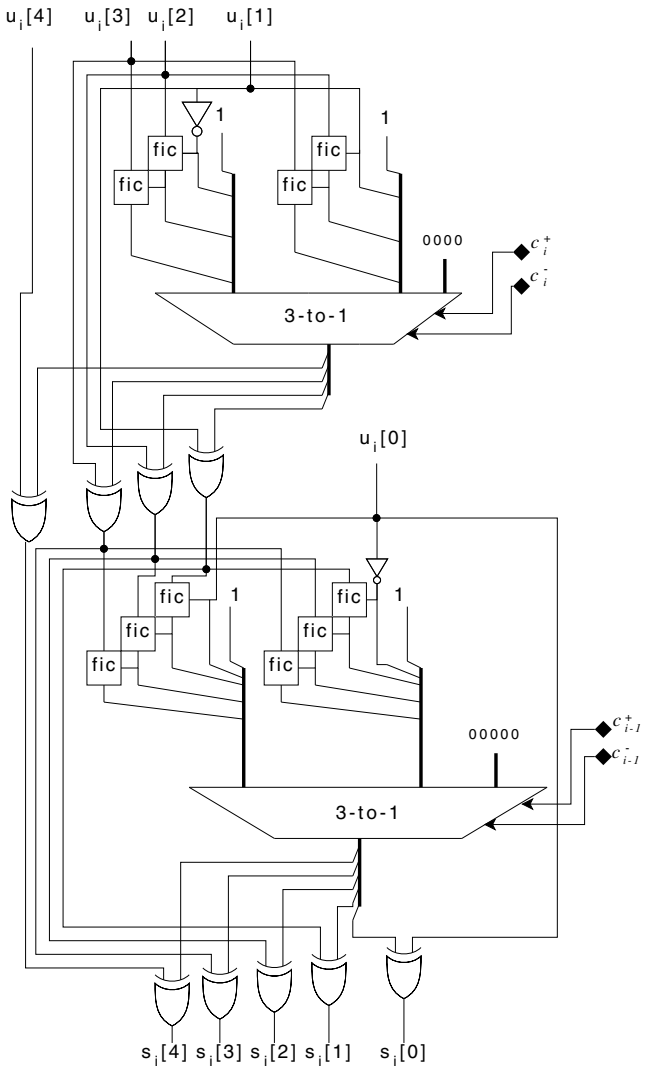


Fig. 4. Parallel correction adder. Not shown: the CPA that generates u_i and the simple circuit that generates c_i^+ and c_i^-. These were eliminated to save room. The first mux is 4 bits wide, and the second is 5 bits wide.

TABLE III
AREA, DELAY AND POWER COMPARISONS

Decimal Adder	Delay (ns)	Area (μm^2)	Cells	Power (mW)	Area-Delay (ns$*\mu$m^2)
RBCD adder	1.55	33100	898	33.6	51300
BCD-RBCD adder	2.16	43100	1261	30.5	93100
Svoboda SD [13]	2.10	41100	1193	30.9	86300
Speculative SD [12]	2.11	59600	1663	50.4	126000
Proposed SD	1.70	38900	1001	39.8	66100
Proposed Parallel SD	1.61	39500	992	40.4	63600

VII. Results

Table III shows that the new SD adder designs proposed in this work outperform existing SD designs in delay and area. With respect to the Svoboda adder, delay can be reduced by 23% with a 4% decrease in area. Furthermore, the proposed SD adders achieve better area and delay using the more common two's complement number representation instead of the Svoboda adder's unique number representation.

Against the Speculative SD in [12], there is a 24% reduction of delay. More appreciable is the improvement in area, which is about 34%. Storage-wise, the new adders (as well as the Svoboda adder) represent a digit with 5 bits instead of 8.

The RBCD adder is fast, however it requires extra logic for conversion from BCD to SD. A BCD-RBCD adder that performs this conversion was synthesized along with the other SD adders. Against the BCD-RBCD adder, the proposed parallel adder is 25% faster and 9% smaller.

These results show that decimal addition can be performed quickly and efficiently with a signed-digit representation. Best of all, delay will not increase for more digits. The only drawback for the proposed SD adders is a 30% increase in power with respect to existing schemes.

VIII. Conclusion

With the growing desire to incorporate decimal arithmetic into microprocessors, carry-free signed-digit decimal addition using a two's complement representation and flagged adders presents a competitive solution. The proposed decimal adder schemes provide short critical path delays and are efficient with area. They can also be integrated into multi-operand decimal addition, multiplication units and more.

References

[1] M. Cowlishaw, "Decimal floating-point: algorism for computers," in *Computer Arithmetic, 2003. Proceedings. 16th IEEE Symposium on*, June 2003, pp. 104–111.

[2] (2008) BigDecimal. [Online]. Available: http://java.sun.com/products

[3] (2008) alphaworks: decnumber. [Online]. Available: http://www.alphaworks.ibm.com/tech/decnumber

[4] E. Schwarz and S. Carlough, "Power6 decimal divide," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, July 2007, pp. 128–133.

[5] A. Tsang and M. Olschanowsky, "A study of database 2 customer queries," IBM Santa Teresa Laboratory, San Jose, CA, USA, Tech. Rep. TR-03.413, Apr. 1991.

[6] R. Kenney and M. Schulte, "High-speed multioperand decimal adders," *Computers, IEEE Transactions on*, vol. 54, no. 8, pp. 953–963, Aug. 2005.

[7] I. D. Castellanos and J. E. Stine, "Compressor trees for decimal partial product reduction," in *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2008, pp. 107–110.

[8] I.S. Hwang, "High speed binary and decimal arithmetic unit," *United States Patent*, no. 4,866,656, September 1989.

[9] A. Bayrakci and A. Akkas, "Reduced delay BCD adder," in *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, July 2007, pp. 266–271.

[10] M. Erle, E. Schwarz, and M. Schulte, "Decimal multiplication with efficient partial product generation," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*, June 2005, pp. 21–28.

[11] M. Erle, M. Schulte, and B. Hickmann, "Decimal floating-point multiplication via carry-save addition," in *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, June 2007, pp. 46–55.

[12] J. Moskal, E. Oruklu, and J. Saniie, "Design and synthesis of a carry-free signed-digit decimal adder," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, May 2007, pp. 1089–1092.

[13] A. Svoboda, "Decimal adder with signed digit arithmetic," *Computers, IEEE Transactions on*, vol. C-18, no. 3, pp. 212–215, March 1969.

[14] B. Shirazi, D. Yun, and C. Zhang, "RBCD: redundant binary coded decimal adder," in *Computers and Digital Techniques, IEE Proceedings E*, vol. 136, no. 2, Mar 1989, pp. 156–160.

[15] ——, "VLSI designs for redundant binary-coded decimal addition," in *Computers and Communications, 1988. Conference Proceedings., Seventh Annual International Phoenix Conference on*, Mar 1988, pp. 52–56.

[16] V. Dave, E. Oruklu, and J. Saniie, "Design and synthesis of a three input flagged prefix adder," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, May 2007, pp. 1081–1084.