INTEGRATION, the VLSI journal 43 (2010) 258–267

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Constant addition with flagged binary adder architectures

Vibhuti Dave [a,*], Erdal Oruklu [b], Jafar Saniie [b]

[a] Penn State Erie, The Behrend College, USA
[b] Illinois Institute of Technology, USA

ARTICLE INFO

Article history:
Received 14 July 2009
Received in revised form
10 March 2010
Accepted 18 March 2010

Keywords:
Constant addition
Prefix adders
Carry-save Adders

ABSTRACT

The goal of this paper is to present architectures that provide the flexibility within a regular adder to augment/decrement the sum of two numbers by a constant. This flexibility adds to the functionality of a regular adder, achieving a comparable performance to conventional designs, thereby eliminating the need of having a dedicated adder unit to perform similar tasks. This paper presents an adder design to accomplish three-input addition if the third operand is a constant. This is accomplished by the introduction of flag bits. Such designs are called Enhanced Flagged Binary Adders (EFBA). It also examines the effect on the performance of the adder when the operand size is expanded from 16 bits to 32 and 64 bits. A detailed analysis has been provided to compare the performance of the new designs with carry-save adders in terms of delay, area and power.

1. Introduction

Multi-operand addition is a part of many complex arithmetic algorithms, such as multiplication and certain DSP algorithms. One of the popular multi-operand adders is the carry-save adder [1] capable of adding more than two operands at a time. The objective of this paper is to introduce the flexibility of adding three-input operands to a regular adder, thereby eliminating the need of a special adder to do the same. This paper investigates the use of parallel prefix adders, carry-skip and carry-select adders [2] in order to achieve basic three-operand addition. It also compares the performance of this technique with carry-save addition (CSA) used as the benchmark for 16, 32 and 64-bit operands.

In order to increment or decrement a sum of two numbers, A and B by unity, parallel prefix adders can be easily modified to generate a new set of intermediate outputs called *flag bits*, making them Flagged Prefix Adders (FPA) [3]. These bits are used to invert selected bits from the sum, $A+B$ to generate the new result, $A+B+1$ or $A-B-1$. In addition to prefix adders, hardware can be incorporated within the carry-skip and carry-select adders to accomplish the same leading to Flagged Binary Adders (FBA) [4]. The same concept can be used to generate appropriate flag bits, to augment or decrement the sum of two numbers by an arbitrary operand, M. Enhanced Flagged Binary Adders (EFBA) [5] are designed and implemented with the assumption that M is a constant.

The advantage of using the proposed technique is that it introduces extra functionality to a regular adder, making it more flexible and convenient to use in applications such as image processing, DSP operations, decimal arithmetic, modulo arithmetic, and floating point multiplication. This paper starts with the design and implementation of flagged binary adders, followed by the design and implementation of enhanced flagged binary adders. Circuits have been developed for three different operand sizes, 16, 32, and 64 bits. A comparison has also been made between EFBA designs and carry-save adders since the latter is conventionally used for multi-operand addition.

Section 2 provides an overview of the different adder architectures that have been utilized to investigate the performance of the new technique. Section 3 has been dedicated to the derivation of the logic in order to compute flag bits correctly. Section 4 discusses the hardware implementation for the FBA, and EFBA architectures. Section 5 presents a theoretical analysis in order to understand what to expect after simulation. A gate count analysis has been provided to estimate the effect on area due to the extra hardware. The method of logical effort has been applied for delay estimations. Section 6 lists a range of applications for which constant addition is a necessity, and the utilization of EFBA designs could be a more efficient option over CSA. Section 7 investigates the performance of each adder design for three different wordlengths in terms of area, delay, and power based on results obtained from synthesis and simulation. The conclusion has been provided in Section 8.

* Corresponding author.
E-mail addresses: vbd1@psu.edu (V. Dave), erdal@ece.iit.edu (E. Oruklu), sansonic@ece.iit.edu (J. Saniie).

0167-9260/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.vlsi.2010.03.001

2. Background

The delay of an adder circuit often determines the clock cycle time of a processor, especially if it falls in the critical path of the design [24]. One of the primary causes, for the delay of an adder is the rippling nature of the carry. The key to fast addition is to compute carry bits for every bit position in parallel. The recurrence relationship presented in (1) achieves this conveniently by introducing the *generate* or *g* signal given by $g_i = a_i b_i$ and the *propagate* or *p* signal given by $p_i = a_i \wedge b_i$, where *i* represents the bit position [1]. C_{i+1} is the input carry for position *i*. In this paper "$\wedge$" refers to a logical OR operation to avoid confusion with addition.

$$c_{i+1} = g_i \wedge p_i\, c_i \tag{1}$$

The parallel prefix adders compute the carries in parallel efficiently by representing addition as a prefix computation [7]. Carry-skip and carry-select adders also employ similar techniques to reduce the computation times for the carries [8]. The following subsections briefly discuss each of these architectures.

2.1. Prefix adders

The parallel prefix adder (PPA) accomplishes the computation of the output carries in parallel by expressing binary carry-propagate addition as a prefix computation. Parallel prefix logic combines *n* inputs using an arbitrary associative dot operator, $\circ$ to get *n* outputs so that the outputs depend only on the input operands. The $\circ$ operator is shown in (2) where (g_1, p_1) and (g_2, p_2) are the inputs and, (G, P) are the outputs [11]. Upper case letters are used to denote outputs when the computation is performed on a range of bits over multiple positions.

$$\begin{aligned} G &= g_1 \wedge g_2 p_1 \\ P &= p_1 p_2 \end{aligned} \tag{2}$$

The parallel prefix adder computes the sum in three stages which comprise of the pre-processing, prefix computation and post-processing stages as shown in Fig. 1. The prefix trees selected for this paper are the Brent–Kung [9], Ladner–Fischer [7], and the Kogge-Stone [10] structures.

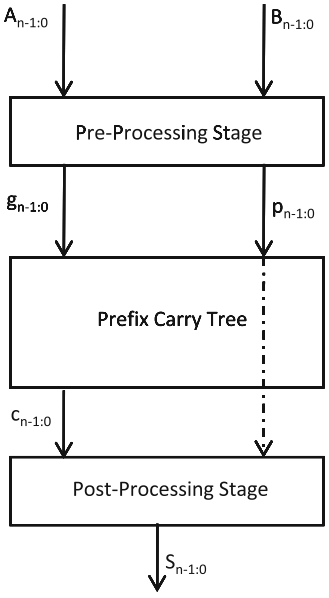


Fig. 1. Block diagram of a prefix adder.

2.2. Carry-skip adders

A carry-skip adder uses the concept of generating the group propagate signal in order to determine if the *carry out* of a set of bits is identical to the *carry in* [11]. The carry-skip adder uses a regular full adder for every bit position which also generates the bit propagate signal, *p*. The adder structure is divided into blocks of consecutive stages with the full adder scheme modified to output the bit propagate signal. Every block generates a group propagate signal represented as $P_{i:k}$. This signal determines, whether the carry out, c'_{i+1} of the block is propagated to the next block, or if it is skipped and instead, the input carry, c_k is directly selected as the carry out, c_{i+1}. This is expressed according to (3):

$$c_{i+1} = \overline{P_{i:k}}\, c'_{i+1} \wedge P_{i:k} c_k \tag{3}$$

2.3. Carry-select adders

The underlying strategy of the carry-select adder is to generate two results in parallel [12,21]. One result assumes the input carry to be a zero and the other assumes the input carry of one. The carry-select adder is divided into blocks of *m*-bit vectors. Each block generates two outputs according to the equations presented in (4).

$$\begin{aligned} (c_m^0, S^0) &= ADD(A, B, c_0 = 0) \\ (c_m^1, S^1) &= ADD(A, B, c_0 = 1) \end{aligned} \tag{4}$$

Here, *A*, *B* and *S* are *m*-bit vectors. Once the input carry for a particular stage has been computed and assigned, the final result is selected from the two pre-computed sets.

2.4. Carry-save adders

When adding three numbers, the result can be represented redundantly as two numbers [1]. One number consists of all the sum bits and the other number is comprised of all the carry bits. The carry propagation is saved for a later stage within the design. Observe in Fig. 2, three bits in each column are operated upon with (3,2) counters, generating the sum (*S*) and the carry (*C*) bits as two separate outputs. The carry-save adders can be utilized as efficient multi-operand adder circuits. They tend to be very fast due to the absence of carry propagation until the last stage. An added advantage is their simple structure. However, the potential for further improvement is minimal [20].

Having discussed each adder, the following section discusses the technique and hardware required to compute a new set of intermediate outputs called flag bits, which eventually contributes to three-input addition.

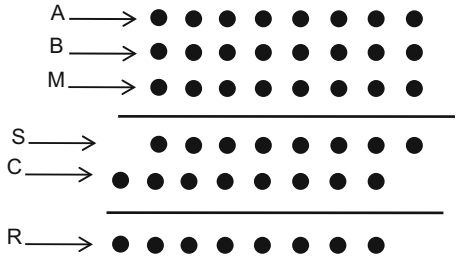


Fig. 2. 8-bit carry-save adder.

3. Flag bit computation

A variety of applications require certain arithmetic operations such as incrementing the sum of two numbers by unity, finding the absolute difference between two numbers, or augmenting the sum of two numbers by a constant. One approach to perform these operations is to utilize dual adders or use multi-operand adders such as the carry-save adders. This paper presents area-efficient hardware modifications whereby, only one kind of regular adder is made capable of performing the above mentioned operations by introducing a new set of bits, called the *flag* bits. This set of intermediate outputs flag the inversion of the *sum* bits (the result from the addition of two numbers) to generate appropriate results. In other words, when the sum bits are XORed with the flag bits, the output is another function result. The technique first proposed, resulted in a new set of architectures called the flagged prefix adders [3]. This paper generalizes the technique to other binary adders and also derives the Boolean expressions to generate flag bits for constant addition. The next subsections describe these concepts in detail.

3.1. Increment and decrement sum by unity

Basic increment and decrement operations can be performed by introducing flag bits, f_i that are related to the group propagate, $P_{i-1:0}$ signals according to [3,18]:

$$f_i = P_{i-1:0} \tag{5}$$

For parallel prefix adders, the required signals to compute flag bits can be easily generated at the output of the prefix tree which is the second stage of the adder. For the carry-skip adder, these signals are generated within the adder itself as can be seen in (3). However, the carry-select adder does not generate the group propagate signals and hence, it is infeasible to introduce the extra flexibility to this family of adders [13]. A simple 8-bit example is as follows. The final result ($A+B+1$) is obtained by XORing the sum, S and flag bits, F.

A	0	0	0	0	1	0	0	1
B	0	1	0	0	1	1	1	0
$S=A+B$	0	1	0	1	0	1	1	1
F	0	0	0	0	1	1	1	1
$R=A+B+1$	0	1	0	1	1	0	0	0

3.2. Increment and decrement the sum by a constant, M

The computation of flag bits for three-input flagged binary adders is slightly more complicated than that for the basic version [14]. The flag bits as proven in this section are related to the third operand and the carry outputs at every bit position. The traditional logic equations for a full adder (FA) are [25]:

$$\begin{aligned} S_k &= A_k \oplus B_k \oplus c_k \\ c_{k+1} &= A_k B_k \wedge A_k c_k \wedge B_k c_k \end{aligned} \tag{6}$$

Here, the adder takes in two input operands A and B, producing a result, S and a carry out c_{k+1} at every bit position k. Introducing the third operand, M, which is the value by which the result S, needs to be augmented or decremented, (6) can be rewritten as:

$$\begin{aligned} R_k &= S_k \oplus M_k \oplus d_k \\ d_{k+1} &= S_k M_k \wedge S_k d_k \wedge M_k d_k \end{aligned} \tag{7}$$

Here R represents the final sum of $A+B+M$, and d_{k+1} represents the corresponding carry-output at every bit position k. Utilizing the new set of equations, the new sum needs to be computed such that, $S+M=S\oplus F$, where F is the flag function. The flag bits can be seen as bits that indicate whether the current value is flagged to change. Consequently, the flag bits can be computed based on speculative elements of the constant.

Table 1
Flag bit logic using carry bits from the CPA.

M_k	M_{k-1}	F_k,
0	0	$S_{k-1} d_{k-1}$
0	1	$S_{k-1} + d_{k-1}$
1	0	$\overline{S_{k-1}} + \overline{d_{k-1}}$
1	1	$\overline{S_{k-1} d_{k-1}}$

$$d_{k+1} = \begin{cases} S_k d_k & M_k = 0 \\ S_k \wedge d_k & M_k = 1 \end{cases} \quad F_k = \begin{cases} d_k & M_k = 0 \\ \overline{d_k} & M_k = 1 \end{cases} \tag{8}$$

Two bits from the third operand can be examined to determine whether or not the carry-bit affects the flag bit in the current position. For example, assume that $M_k=0$ and $M_{k-1}=1$ utilize the relationship in (8) to achieve (9).

$$\begin{aligned} &d_{k+1} = S_k d_k M_k = 0 \\ &d_k = S_{k-1} \wedge d_{k-1} \quad M_{k-1} = 1 \end{aligned} \tag{9}$$

The computation of the flag bits, therefore relies on the carry bits that are generated after adding three operands, *A, B,* and *M.* (8) and (9) can be utilized to derive Table 1. In order for the equations to be computed properly, some initial conditions need to be assumed and are given in (10).

$$\begin{aligned} &S_{-1} = M_{-1} = 0 \\ &F_0 = M_0 \end{aligned} \tag{10}$$

The following example can further clarify this operation. Having derived the logic required to compute flag bits for different operations, the next section describes the hardware implementation.

$A=9$	0	0	0	0	1	0	0	1
$B=78$	0	1	0	0	1	1	1	0
S	0	1	0	1	0	1	1	1
$M=57$	0	0	1	1	1	0	0	1
F	1	1	0	0	0	1	1	1
$R=S+57$	1	0	0	1	0	0	0	0

4. Implementation

This section starts with the description of flagged binary adders. Parallel prefix adders and the carry-skip adder are incorporated with extra hardware in order to obtain useful results such as $A+B+1$ and $A-B-1$. This is followed by a discussion of changes that need to be made in order to enable a binary adder to add three operands, with the limitation that the third operand is a constant (EFBA).

4.1. Flagged binary adders

This section describes the implementation of parallel prefix and carry-skip adders incorporated with the technique of flag bit generation. According to (5), the flag bits are directly computed from the group propagate signals. This is accomplished within the parallel prefix adders by modifying the prefix tree to output the flag bits in addition to the carry outputs. The block diagram in Fig. 1 is therefore modified to Fig. 3 as shown below.

The flag bits at the output of the prefix carry tree can be easily obtained by changing all the gray cells to black cells [4]. This change is feasible due to the very simple relationship between the

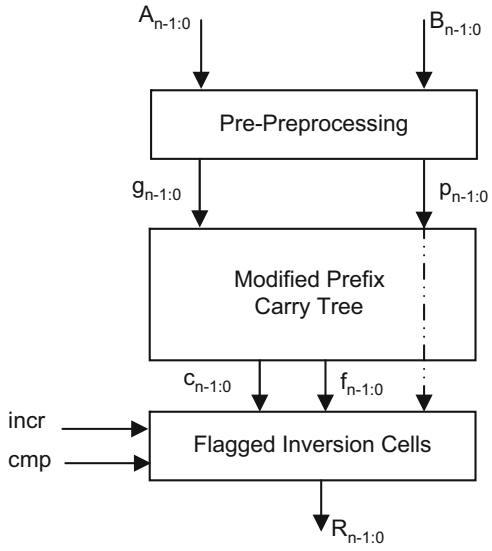


Fig. 3. Modified PPA block diagram.

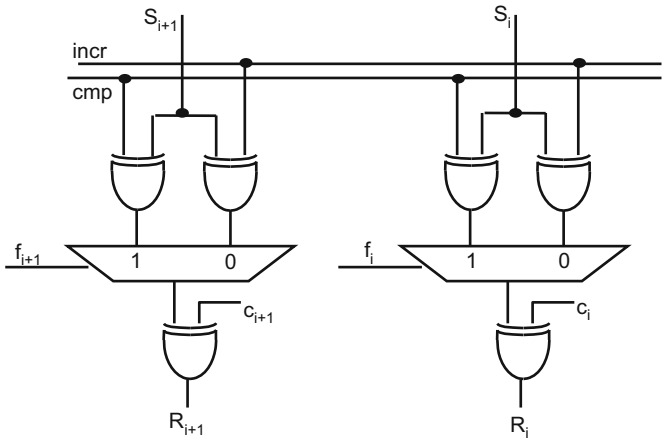


Fig. 4. Flagged inversion cells.

flag bits and the group propagate signals in (5). The prefix tree is designed to provide the necessary group generate signals to compute the carry bits and by including these cells in the last stage of the tree, the group propagate signals are also readily available. Notice the inclusion of a new block called Flagged Inversion Cells (FIC) [3]. The FIC is the extra hardware required in the final stage of the adder to compute the final output. A detailed view of the FIC is shown in Fig. 4.

The signals, *incr* and *cmp* are used to select the appropriate result from all the different possible results as listed in Table 2. The minimal amount of additional hardware therefore comprises of two levels of XOR gates and a multiplexer, thereby affecting the critical path minimally. The same can be accomplished using the carry-skip adders. The carry-skip adder can conveniently incorporate the computation of flag bits since, it generates the bit propagate signals for every bit position.

Therefore the computation of the flag bit for position i will require an AND gate according to the following equation:

$$P_{i-1:0} = AND_{k=0}^{k=i-1} p_k \tag{11}$$

The block diagram of the carry-skip adder is shown in Fig. 5. Minimal modifications need to be made in order to accomplish the same results. The idea of flag bit generation is utilized in the following subsections to obtain constant addition.

Table 2
Results obtained from flagged binary adders.

incr	cmp	Result(Add)	Result(Sub)
0	0	$A+B$	$A-B-1$
0	1	$A+B+1$	$A-B$
1	0	$-(A+B+2)$	$B-A$
1	1	$-(A+B+1)$	$(B-A-1)$

4.2. Enhanced flagged binary adders

Enhanced flagged binary adders (EFBA) incorporate extra hardware allowing the third operand to be an arbitrary constant [4]. This implies that it is a requirement to know what the third operand is going to be ahead of time before hardware modifications can be made to any adder design. The underlying technique is to generate flag bits with the computation of flag bits getting more complicated due to the dependence of the bits on the third operand. Computation of flag bits for this new adder design also depends on the carry outputs, readily computed according to (9).

Following the same example of 9+78+57, where $(57)_{10}=(00111001)_2$ is the constant, M, Fig. 6 shows the necessary gates required to compute the carry outputs, d_k. This figure is a direct implementation of (9).

The carry signal will ripple through only one gate for every bit position. Once the carry bits are obtained, the flag bits can be computed in parallel according to Table 1 after a delay of one gate level. Fig. 7 shows the logic gates required to compute the flag bits.

Notice that $F_1=d_1$, $F_2=d_2$, $F_6=d_6$, and $F_7=d_7$. This implies that the gates used to compute flag bits in these bit positions can be eliminated in order to optimize the hardware. Similar optimizations can be applied for other constants. The flag bits, F will be XORed with the sum bits, S, to get the final result, R. It is important to note that the critical delay of the circuit will not change if the constant is changed assuming that a 2-input AND and a 2-input OR gate have equal delay. However, the types and order of the 2-input gates utilized to compute d_k will change. This design provides two separate useful results with minor modifications and minimal impact on the critical delay. Flag logic can be incorporated in parallel prefix, carry-skip, and carry-select adders in a straightforward fashion. This family of adders was modified to incorporate the flag logic shown in Fig. 7 for 16, 32, and 64-bit operand sizes. The value of the constant chosen is equal to 57.

Typical applications requiring constant addition include rounding algorithms and decimal arithmetic. Another useful application for such an adder would be within a floating point multiplier. This adder design can be also be used if reconfigurable hardware is being utilized thereby making the change of hardware easy.

5. Gate count and delay analysis

A theoretical analysis has been provided in this section to estimate area and delay numbers for the proposed architectures. The first subsection provides an approximate gate count for every adder design to estimate area. This is followed by an analysis based on the method of logical effort to estimate the delay along the critical path of the circuits.

5.1. Gate count

In order to understand the tradeoffs between the extra hardware and the added flexibility incorporated within the adder,

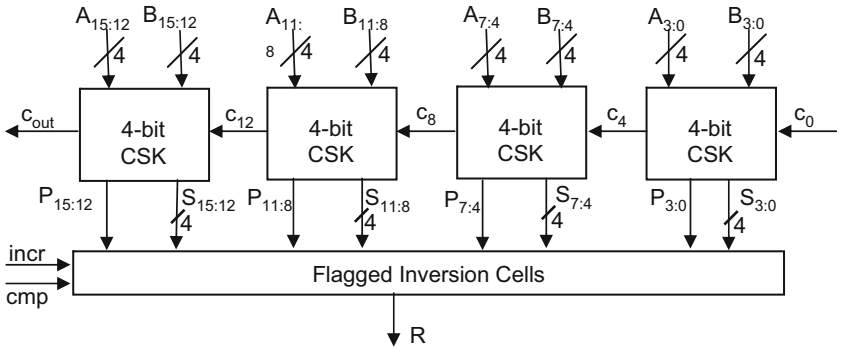


Fig. 5. Flagged carry-skip adder.

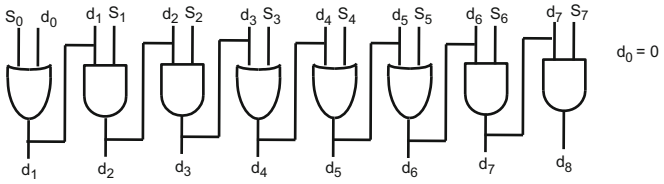


Fig. 6. Carry computation for constant addition.

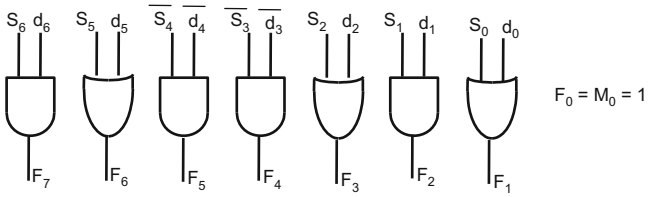


Fig. 7. Flag bit computation for constant addition.

an area analysis in terms of gate count has been provided. This analysis gives an idea of the impact that can be expected on the area of a regular adder. The XOR, AND, OR, and INVERTER are all considered as single gates for the presented analysis. The multiplexer is considered as a separate complex gate entity.

For the normal prefix adder, consider, n as the wordlength and k as the multiplying factor of the prefix adder architecture used. k is a factor that determines the number of prefix cells for a given architecture. The gate count (gc) used in an n-bit parallel prefix adder (gc_{pp}) can be summarized as in (12) [5]:

$$gc_{pp} = n(p \text{ and } g \text{ cells}) + k(\text{black prefix cells}) + (n-1)(\text{gray prefix cells}) + n(\text{XOR}) \quad (12)$$

The p and g cells are utilized to calculate the bit propagate and the bit generate signals in the pre-processing stage and therefore contribute to a total of $2n$ AND/OR gates. The black and the gray prefix cells are a part of the prefix tree, each comprising of 3 and 2 AND/OR gates, respectively. The post-processing stage consists of a group of XOR gates to calculate the final sum. For a Brent–Kung adder, $k=n$ [3], for the Ladner–Fischer prefix adder, $k=n/2\ log_2 n/4$ [3], and for the Kogge-Stone architecture, it has been proved that $k=n\ log_2 n/4$ [3].

The gate count will increase for a flagged prefix adder (gc_{fpp}) capable of performing basic increment decrement operations and hence (12) is modified to give (13),

$$gc_{fpp} = n(p \text{ and } g \text{ cells}) + k(\text{black prefix cells}) + (n-1)(\text{black prefix cells}) + n(\text{MUX}) + 3n(\text{XOR}) \quad (13)$$

The multiplexers are a part of the flag inversion cell logic as seen in Fig. 4. The number of XOR gates increases from n to $3n$ due to their presence in the flag inversion cells as well. Following the same concept, the gate count for prefix adders capable of constant addition (gc_{epa}) is given in (14):

$$gc_{epa} = n(p \text{ and } g \text{ cells}) + k(\text{black prefix cells}) + (n-1)(\text{gray prefix cells}) + 2n(\text{AND/OR}) + 2n(\text{XOR}) \quad (14)$$

No modifications are made to the prefix tree in this case and therefore, the prefix cells stay the same as in the original prefix adder. However, the stages following the prefix tree comprise of the cells that compute the new set of carry-signals and the flag bits. 2-input AND/OR gates are utilized in every bit position to compute d_k. The same holds true for the flag bits according to Table 1. In addition to the n XOR gates required to compute the sum of two input operands, an additional set of n XOR gates are required to compute the final result, i.e., the sum of three-input operands.

In order to get optimum performance from a carry-skip adder, it is recommended to divide the operand into groups of varying lengths [11]. Example, it is advantageous to implement a 16-bit carry-skip adder with 4 blocks. However instead of dividing the 16-bit operand into 4 blocks of 4 bits each, the blocks are 3, 5, 5, and 3 bits long starting from the least significant block. For a carry-skip adder, it is infeasible to express the gate count as an equation since it depends on the number of groups that the adder is divided into and the size of each group [11]. In this paper, the following group sizes have been utilized.

Wordlength	Group Size
16	3-5-5-3
32	3-5-8-8-5-3
64	2-4-5-6-7-8-8-7-6-5-4-2

Consider m_i as the group size which implies that each group comprises of m_i full adder (FA) modules. Each FA in turn consists of 2 XOR gates, 2 AND gates and 1 OR gate, a total of 5 basic gates. Also, each group includes AND gates in order to compute the group propagate output. Each carry-skip module needs a multiplexer to select the output carry based on the group propagate signal according to (3) and therefore each adder will consist of the same number of multiplexers as there are groups. It is also important to realize that when the group size exceeds 4 bits, it is necessary to compute the group propagate utilizing multi-level AND gates, in order to ensure, that the fan-in of any of the gates does not exceed four. For a flagged carry-skip adder, an additional set of AND gates are a part of the carry-skip module to enable the correct computation of flag bits according to (5). On the other

Table 3
Gate count for all adder implementation.

Adder designs	16 bits		32 bits		64 bits	
	Gate count	Mux count	Gate count	Mux count	Gate count	Mux count
Brent–Kung	123	0	254	0	510	0
Ladner–Fischer	129	0	305	0	705	0
Kogge-stone	179	0	449	0	1089	0
Carry-skip	86	3	170	5	329	10
Carry-select	160	3	320	7	640	15
Flagged BK	141	16	285	32	573	64
Flagged LF	144	16	336	32	768	64
Flagged KS	192	16	480	32	1152	64
Flagged CSK	148	19	298	37	583	74
Enhanced BK	169	0	348	0	700	0
Enhanced LF	175	0	499	0	895	0
Enhanced KS	225	0	543	0	1279	0
Enhnaced CSK	164	3	264	5	519	10
Enhanced COS	206	3	414	7	830	15

hand, for an enhanced flagged carry-skip adder, the gate count is only increased by $2n$ AND/OR gates to compute carry and flag bits.

Similar to the carry-skip adder, the carry-select adder is also divided into groups. Assume, the n-bit adder is divided into groups of size, m bits. Each group consists of an m-bit conditional adder and $(n/m)-1$ MUXs. The conditional adder module in turn consists of two m-bit FAs, thereby leading to a total of 10 m gates in each module [10]. The total number of gates therefore equals

$$gc_{COS} = (10m)(n/m)(\text{AND/OR}) + (n/m) - 1(\text{MUX}) \quad (15)$$

Once the adder is modified to incorporate the flag logic to allow constant addition, the gate count is modified to:

$$gce_{COS} = (10m)(n/m)(\text{AND/OR}) + (n/m) - 1(\text{MUX}) + 2n(\text{AND/OR}) + n(\text{XOR}) \quad (16)$$

Table 3 summarizes the gate count for each adder design. These designs comprise of the conventional, flagged, and enhanced versions for three different operand sizes.

5.2. Delay analysis with logical effort

Logical Effort (LE) is a method described in Ref. [6] to calculate the critical delay of a circuit based on fan-out and gate size. This section applies the method of logical effort to conventional prefix adders, Enhanced Flagged Prefix Adders (EFPA), and the carry-save adder. The focus is on prefix architectures, because they are expected to give minimum delay. The carry-save adder is utilized as a benchmark for the three-input designs since it is the most widely used design for multi-operand addition.

The delay along the critical path, also called the *path delay* is composed of two components, *logical effort* (LE) and *parasitic delay* (PD) [15]. LE is used to estimate the load at the output of a cell and PD is the delay of the cell itself. The path delay is obtained by summing the LE and PD of all the cells that fall on the critical path of the circuit [15].

In order to simplify the analysis, the prefix circuits have been broken down into various cells as shown in Table 4. The function of each cell has been well described in Ref. [15]. For conventional parallel prefix adders with no extra hardware, the FO4 delay based on the method of logical effort has been computed in Ref. [15] and is presented in Table 5. Delays have been presented for 16, 32, and 64-bit operand sizes.

The critical path will change once the extra hardware is incorporated within the design to enable constant addition. For convenience, the critical path is marked in Fig. 8. The path ends at F_7 since, once all the flag bits, F_0 – F_7 are computed, the sum can be easily obtained from the flag bits. Fig. 8 specifically applies to the example used in the paper where the constant is 57. It is important to recall that although the gates in the circuit will change based on the constant, the critical path and therefore, the critical delay will remain the same. This can be emphasized looking at Table 1 and Fig. 7. Table 1 suggests that flag bits for all bit positions can be computed using 2-input AND or OR gates with complemented or uncomplemented inputs. Fig. 7 corresponds to $M=(57)_{10}$. If the constant changes to $(49)_{10}=(00110001)_2$, then the order of the logic gates required to compute the flag bits would change to OR, AND, AND, OR, AND, OR, and AND gates starting from the least significant position. Therefore, it is just the order of the gates that changes, not the critical delay. Note that it is assumed that a 2-input OR gate and a 2-input AND gate will have the same LE and PD values.

According to Fig. 8, the LE and PD will increase as given in (17) and (18). LE_{OR2} and LE_{XOR2} represent the LE values for a 2-input

Table 4
LE and PD for prefix adders.

Cell	Term	Values
Bitwise cell	LE_{bit}	9/3
	PD_{bit}	6/3
Black cell	$LE_{blackgu}$	4.5/3
	$LE_{blackgl}$	6/3
	$LE_{blackpl}$	10.5/3
	LE_{blackg}	4.5/3
	PD_{blackg}	7.5/3
	PD_{blackp}	6/3
Gray cell	LE_{graygu}	4.5/3
	LE_{graygl}	6/3
	LE_{graypu}	6/3
	PD_{gray}	7.5/3
Buffer	LE_{buf}	9/3
Sum XOR	LE_{xor}	9/3
	PD_{xor}	9/3+12/3

Table 5
FO4 delay estimates for parallel prefix adders.

Adder design	Logical effort		
	16 bits	32 bits	64 bits
Brent–Kung	9.4	11.4	15.4
Ladner–Fischer	9	11	14
Kogge-Stone	7.6	9	11.8

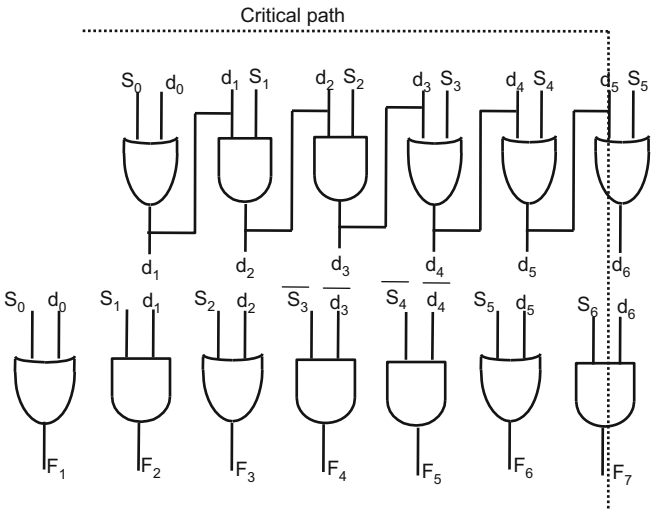


Fig. 8. Flag logic for constant addition.

Table 6
FO4 delay estimates for EFPA.

Adder design	Logical effort		
	16 bits	32 bits	64 bits
Brent–Kung	55.4	99.06	188.4
Ladner–Fischer	55	98.66	188
Kogge-Stone	53.6	96.67	185.8
CSA	57.38	99.76	195.6

OR gate and a 2-input XOR gate (to compute the final result). These values are 5/3 and 4, respectively [6]. The PD values are 2 and 4 for the two gates [6]. Table 6 provides a delay estimate for EFPA based on logical effort.

$$LE_{EFPA} = (n-2)LE_{OR2} + LE_{OR2} + LE_{XOR2} \tag{17}$$

$$PD_{EFPA} = (n-2)PD_{OR2} + PD_{OR2} + PD_{XOR2} \tag{18}$$

In order to compare the performance of the new adder designs with a traditional multi-operand adder, the delay calculations based on logical effort have also been done for a CSA [15]. The LE and PD for a CSA has been represented in (19) and (20).

$$LE_{CSA} = LE_{XOR} + LE_{AND2} + LE_{XOR} + LE_{AND2} + LE_{OR2} + LE_{XOR} + LE_{AND2} + LE_{PPA} \tag{19}$$

$$PD_{CSA} = PD_{XOR} + PD_{AND2} + PD_{OR2} + PD_{XOR} + PD_{PPA} \tag{20}$$

Here LE_{PPA} represents the LE of a parallel prefix adder whose operand size is n. The CSA has been implemented to utilize a Brent–Kung adder in the final stage. Certain assumptions such as, all inputs arrive at the same time with equal drive, and uniform gate size simplify the computations. The logical effort analysis highlights the difference in delay, and the synthesis results confirm the calculations. In the following section, a few example case studies have been presented for the proposed constant adder implementations.

6. Applications

Constant addition with flagged prefix adders can be applied to wide variety of computer arithmetic implementations including floating point operations, decimal arithmetic and image processing.

6.1. Floating point arithmetic

A floating number is represented as

$$F = (-1)^S M \beta^E \tag{21}$$

where S is the sign bit, M is the unsigned fraction (significand), β is the base of the exponent and E is the exponent. Most common floating point representations use a *biased exponent* [22] as expressed in (22):

$$E = E^{true} + bias \tag{22}$$

where *bias* is a constant and E^{true} is the true value of the exponent represented in two's complement.

Given two floating numbers: $F_1 = (-1)^{S_1} M_1 \beta^{E_1 - bias}$ and $F_2 = (-1)^{S_2} \cdot M_2 \beta E_2 - bias$, the result of multiplication is given in (23)

$$F_3 = F_1 \times F_2 = (-1)^{S_3} M_3 \beta^{E_3 - bias} \tag{23}$$

New exponent E_3 will require addition of two exponents $E_1 = E_1^{true} + bias$ and $E_2 = E_2^{true} + bias$. However, *bias* should be subtracted once to obtain the correct exponent (24)

$$E_3 = E_1 + E_2 - bias \tag{24}$$

Similarly, for floating point division, the bias value should be added to the difference $E_1 - E_2$:

$$E_3 = E_1 - E_2 + bias \tag{25}$$

It is important to note that bias value is fixed (constant) for the floating point standard. Therefore, in both cases, two operand additions (or subtractions) and a constant addition are required. This proves that floating point multiplication and division can be implemented optimally using EFBA described in Section 4.2.

6.2. Image addition with contrast enhancement

Adding a constant offset to all pixels in an image is a very common, fundamental operation to brighten or darken images. With the proposed flagged constant addition hardware, addition of two images (for superimposed images) I_1 *and* I_2 with a constant C can be performed simultaneously in a single pass. This operation can be defined as:

$$I_3(i,j) = I_1(i,j) + I_2(i,j) \pm C \tag{26}$$

6.3. BCD addition

A simple application of the proposed adder would be for BCD addition/subtraction. For BCD operations, when the intermediate digit sum needs correction because it is greater than 9, a correction vector 6 is added to the sum. The correction vector never changes and is a constant. Many processors such as those from Intel support this operation as a special instruction, *Decimal Adjust Accumulator (DAA)*. This instruction can be implemented in hardware as two operand addition + constant addition as realized by EFBA.

6.4. Signed-digit decimal addition

A signed-digit two-operand decimal adder with the benefit of carry-free addition is presented in Ref. [23]. 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 $u_i - 10\,c_i$. The operations to obtain these quantities are expressed in (27)–(29).

$$u_i = x_i + y_i \tag{27}$$

$$s_i = u_i - 10 * c_i + c_{i-1} \tag{28}$$

$$c_i = \begin{cases} -1 & \text{if } u_i < -8 \\ 1 & \text{if } u_i\, i > 7 \\ 0 & \text{otherwise} \end{cases} \quad (29)$$

After u_i and c_i are computed, the correction vector must be found and added. The correction vector corresponds to $-10\,c_i$. However, since the incoming carry from the previous digit, c_{i-1}, must eventually be added to u_i, it is convenient to think of the correction vector as $-10c_i+c_{i-1}$. There are 8 possible values (± 1, ± 9, ± 10, ± 11) for the correction vector. The decimal adder described in Ref. [23] is designed to be able to apply any of these correction vectors based on the carry-input using the technique of constant addition with flag bits. Therefore, the proposed EFBA designs can be part of the signed-digit decimal adder hardware as described above.

7. Simulation results

The binary adders described in the previous sections are implemented and modified to be able to perform increment and decrement operations. The adders are also modified to incorporate the flag logic according to Fig. 8 enhancing the functionality of flagged adders to perform constant addition. Each adder was designed for 16, 32 and 64-bit operand sizes. An analysis was performed on all adders with regards to area, delay, and power. The designs are implemented in the TSMC 0.18 μm technology System-on-Chip design flow to investigate the power, area, and delay tradeoffs. Synthesis is performed with Cadence Build Gates and Encounter [16]. The nominal operating voltage is 1.8 V and simulation is performed at $T=25$ °C. Layouts are generated for each adder design and parasitically extracted to obtain numbers for area, delay and power.

7.1. Conventional adder architectures

The results for regular adders, without any modifications are tabulated as shown in Table 7 for 16, 32 and 64-bit designs.

The Ladner–Fischer prefix tree aims at reducing the depth of the tree in order to compute the carry-signals. The complexity measure for this case is the gate count and speed. However it does not perform well in terms of the capacitive fan out load. Contrary to this is the Brent–Kung design that addresses the fan-out restrictions, but the logical depth of the tree is increased. Therefore, the Brent–Kung tree has a more regular structure, which is easy to implement in terms of chip design and wiring density. Another approach is the Kogge-Stone design that limits the lateral logic fan-out to unity at each node, but increases the number of lateral wires at each level. This accounts for the fact that the Kogge-stone adder has the highest value for the multiplying factor k among the three prefix designs selected. Maximum increase in area is observed with the Kogge-Stone adders with increase in operand size. However, it should also be noted that the Kogge-Stone adder has the best performance in terms of speed.

The objective of the carry-skip adder is to reduce the worst-case delay by reducing the number of FA modules through which the carry has to propagate [17]. Therefore, the adder consists of small groups of ripple carry adders modified to include the skip network. This leads to attractive regular structures, but unlike the prefix adders, the carry-signals are not generated in parallel, leading to a less attractive performance in terms of speed. The carry-select adder consists of a pair of carry-propagate adders for each group, leading to significant area consumption.

7.2. Flagged binary adder architectures

The regular adder architectures are modified to perform operations presented in Section 4. Area, delay and power estimates are also obtained for these designs. A comparison has also been made between conventional designs and the flagged architectures to study the impact of the additional hardware on the critical path delay and area consumption.

For the prefix adders, the gray cells are converted to black prefix cells, increasing the gate count by unity for each cell. Also, the inclusion of the flagged inversion cells will account for the rise in area consumption. As the bit count increases, the number of prefix cells within the prefix tree also increases. For a carry-skip adder, the increase in area can be attributed to the fact that the maximum fan-in has been limited to 4. As the number of bits increases, the number of AND gates required in order to obtain all the necessary Group Propagate signals also increases.

The delay of the flagged prefix adders increases almost linearly with the increase in operand size. For, a carry-skip adder, this does not hold true. Again, the multi-level AND gate inclusion accounts for a higher increase in delay compared to the prefix tree architectures. The carry-select adder has not been incorporated with the flag logic since it proves to be an inefficient design. Table 8 summarizes area, delay and power measurements for the flagged binary adders for 3 different operand sizes.

The impact of the additional hardware is analyzed by looking at the difference in area and delay between a regular adder and flagged binary adder. The increase in area is more noticeable for the Kogge-Stone adder than it is for the Brent–Kung or Ladner–Fischer designs. Also notice the significant impact on area for the carry-skip design. Although the gate count increase is higher for the prefix trees, the layout shows a major difference for the carry-skip adder. The variable group sizes utilized within the design is responsible for the reduction in power consumption, but leads to a negative impact on area consumption. The impact on delay proves to be insignificant after inclusion of the flagged inversion cells within each design. This particularly holds true for the prefix adder designs. The critical path is affected minimally, compared to the flexibility incorporated within the design. The Kogge-Stone structures prove to be the fastest, but the trade-off is the area consumption. It can also be observed, that the power consumption for these structures is the highest probably due to the high switching speed within the circuit.

Table 7
Simulation results for conventional adder designs.

Adders/parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ps)	Power (mW)	Area (mm^2)	Delay (ps)	Power (mW)	Area (mm^2)	Delay (ps)	Power (mW)
Brent–Kung	0.276	815	5.63E−04	0.8551	973	2.35E−03	1.8843	1255	3.97E−03
Ladner–Fischer	0.276	780	5.81E−04	0.8583	937	2.30E−03	1.8871	1163	4.11E−03
Kogge-Stone	0.496	682	7.78E−04	1.6146	770	3.31E−03	3.701	993	6.21E−03
Carry-skip	0.259	900	4.27E−04	0.6914	1184	9.05E−04	1.6432	1495	2.64E−103
Carry-select	0.327	684	6.04E−04	1.3367	900	9.47E−04	2.707	1245	2.81E−03

Table 8
Simulation results for flagged binary adders.

Adders/parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ps)	Power (mW)	Area (mm^2)	Delay (ps)	Power (mW)	Area (mm^2)	Delay (ps)	Power (mW)
Brent–Kung	0.2803	1118	7.06E−04	0.8917	1270	2.94E−03	2.0871	1553	4.27E−03
Ladner–Fischer	0.2821	1081	7.27E−04	0.9004	1238	3.10E−03	2.1691	1466	4.66E−03
Kogge-Stone	0.5362	987	9.92E−04	1.9146	1078	4.02E−03	3.9163	1300	6.47E−03
Carry-skip	0.3746	1200	6.13E−04	0.8499	1486	1.08E−03	1.9247	1797	3.12E−03

Table 9
Simulation results for enhanced flagged binary adders.

Adders/parameters	16 bits			32 bits			64 bits		
	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)	Area (mm^2)	Delay (ns)	Power (mW)
Brent–Kung	0.29	5.27	8.08E−04	0.9073	8.55	3.05E−03	2.1083	15.34	4.35E−03
Ladner–Fischer	0.2915	5.19	8.34E−04	0.9236	8.49	3.14E−03	2.2339	15.25	4.82E−03
Kogge-Stone	0.5456	5.12	1.12E−03	2.033	8.34	4.10E−03	4.1127	15.1	7.14E−03
Carry-skip	0.39	5.34	7.72E−04	0.8695	8.72	1.17E−03	2.0611	15.57	3.48E−03
Carry-select	0.4953	5.13	9.03E−04	1.5315	8.5	1.28E−03	3.2522	15.33	3.15E−03
Carry-save	0.3455	5.83	1.17E-03	1.2142	10.02	2.06E−03	2.4013	17.49	2.77E−03

7.3. Enhanced flagged binary adders

A similar analysis is performed to study the impact of the flag logic cells for constant addition. Numerical results have been provided in Table 9 for each operand size. The constant utilized for simulation is $M=57$.

Traditionally, carry-save adders can be used to achieve three-operand addition. This technique has been implemented for different operand sizes in order to compare the performance of the proposed architecture with the traditional approach. This can be seen in Table 9 as well. The hardware can be optimized to include fewer gates in the circuit when the carry-bit is the same as the flag bit. Table 9 presents the worst-case scenario when that circuit has not been optimized. Optimization will lead to fewer gates and therefore better area result.

Area increases by approximately 2–4 times for all architectures when the operand size is doubled. Increase in power with change in operand size is significant for prefix structures compared to the carry-save adder. This can be due to the high fan-in and significant wiring complexity associated with the prefix trees. The depth of the Kogge-Stone adder is the least among all three adder architectures resulting in it being the fastest design. The Ladner–Fischer adder has fewer levels compared to the Brent–Kung adder, leading to lower delay numbers. Brent–Kung adders however, have lower numbers of area making it a good choice where area is a constraint. A trade-off between area and delay can be observed for the Kogge-Stone flagged prefix adder, which is observed to be the fastest design among all adder architectures and consumes maximum area. The Ladner–Fischer tree provides a good compromise in terms of area and speed.

Doubling the operand size, causes less than double the increase in delay for all designs including the carry-select adder. This is due to the inherent qualities of the basic adders themselves. This is one of the reasons why these adders were selected to implement constant addition. The increase in delay is seen to be highest for the carry-skip and the carry-select adder. This is due to the fact that unlike the prefix structures, the carries are not obtained in parallel. The adders are divided into groups and hence, each group waits for the preceding group to generate the carry. The carry computation for constant addition falls in the critical path of the design as explained in the previous section. However, the added flexibility is an advantage.

It is important to notice the difference in delay between the adders designed for constant addition and the carry-save adder. EFBA architectures have a favorable performance compared to the CSA. The delay incurred due to the additional hardware incorporated to implement constant addition is less than the delay incurred by the prefix adder in the last stage of a carry-save adder. The delay results concur with the logical effort FO4 delay estimates from Section 5. It can be concluded, that with minor modifications to the hardware, more than one useful functional results can be obtained utilizing prefix structures. In applications such as decimal arithmetic, this flexibility can be exploited. Constant addition can also be utilized with applications that use floating point numbers where the exponent is always stored as a biased number.

8. Conclusion

Prefix adder architectures capable of constant addition and three-operand addition [19] for cell based design and their synthesis have been implemented and investigated in this paper. Prefix adders have been incorporated with minimal hardware to compute a new set of bits called flag bits. The flag bits can be used to obtain two different results, $A+B$ and $A+B+M$.

The proposed design can be used as a replacement to carry-save adders with the possibility of having the third operand as a constant. EFBA designs have a favorable performance compared to carry-save adders in terms of area and delay. The Kogge-Stone adder and the Ladner–Fischer adder provide the best results when incorporated with the extra hardware. One of the advantages is the availability of two useful results within one circuit. It eliminates the need to have dedicated adder units to perform the operation since the new logic is incorporated within a regular adder. For three-operand addition, the additional logic leads to a significant increase in delay limiting their use for general three-operand addition.

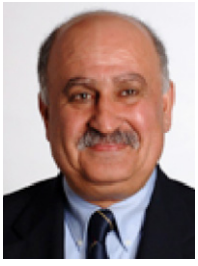
In terms of applications, EFBA designs can be used for decimal arithmetic, floating point multiplication, signal processing and image processing applications where a biasing constant needs to be added to improve signal quality and also floating point arithmetic.

References

[1] R. Zimmerman, "Binary adder architectures for cell based VLSI and their synthesis," Ph.D. dissertation, Swiss Federal Institute of Technology, Zurich, 1997.
[2] A. Tyagi, A reduced area scheme for carry-select adders, IEEE Transactions on Computers vol. 42 (1993) 1163–1170.
[3] N. Burgess, The Flagged prefix adder for dual additions, in: Franklin T. Luk (Ed.), Proceedings of SPIE, Advanced Signal Processing Algorithms, Architectures, and Implementations VIII, vol. 3461, 1994, pp. 567–575.
[4] V. Dave, E. Oruklu, and J. Saniie, "Analysis, design and synthesis of flagged binary adders with constant addition," in: Proceedings of the 49th IEEE International Midwest Symposium on Circuits and Systems, vol. 1, pp. 23–27, 2006.
[5] V. Dave, "High-speed multi-operand addition utilizing flag bits," Ph.D. dissertation, Illinois Institute of Technology, Chicago, 2007.
[6] I. Sutherland, Bob Sproull, David Harris, in: Logical Effort: Designing Fast CMOS Circuits, Morgan Kauffmann, 1999.
[7] R. Ladner, M. Fischer, Parallel prefix computation, Journal of the ACM vol. 27 (1980) 831–838.
[8] J. Rabaey, A. Chandrakasan, and B. Nikolic, Digital Integrated Circuits. A Design Perspective. 2nd ed., Prentice Hall.
[9] R. Brent, H Kung, A regular layout for parallel adders, IEEE Transactions on Computers vol. 31 (1982) 260–264.
[10] P. Kogge, H. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations, IEEE Transactions on Computers vol. 22 (1973) 783–791.
[11] M. Ercegovac, and T. Lang, Digital Arithmetic. Morgan Kaufmann.
[12] H. Lindkvist, and P. Anderson, "Techniques for fast CMOS based conditional sum adders," in: Proceedings of the 1994 International Conference on Computer Design, pp. 626–635, 1995.
[13] V. Dave, E. Oruklu, and J. Saniie, "Performance evaluation of flagged prefix adders for constant addition," in: Proceedings of the 6th IEEE International Conference on Electro/Information Technology, 2006, 2006.
[14] J. Stine, C. Babb, and V. Dave, "Constant addition utilizing flagged prefix structures," in: Proceedings of the International Sympoisum on Circuits and Systems, pp. 668–671, 2005.
[15] D. Harris, and I. Sutherland. "Logical effort of carry propagate adders." Conference Record of the Thirty-Seventh Asilomar Conference on Signals, Systems and Computers, vol. 1, pp. 873–78, 2003.
[16] J. Grad, and J. Stine, "A standard cell library for student projects," in: Proceedings of the International Conference on Microelectronics Systems Education, pp. 98–99, 2003.
[17] S. Winograd, On the time required to perform addition, Journal of the ACM vol. 12 (1965) 277–285.
[18] N. Burgess, The flagged prefix adder and its applications in Integer arithmetic, Journal of VLSI Signal Processing Systems vol. 31 (2002) 263–271.
[19] V. Dave, E. Oruklu, and J. Saniie, "Design and synthesis of a three input flagged prefix adder," in: Proceedings of the IEEE International Symposium on Circuits and Systems, ISCAS 2007, pp. 1081–1084, 2007.
[20] S. Knowles, "A family of adders," in: Proceedings of the 14th IEEE Symposium on Computer Arithmetic, pp. 30–34, 1999.
[21] Sklansky, Conditional sum addition logic, IRE Transactions on Electronic Computing vol. EC-9 (1960) 226–231.
[22] Israel Koren, in: Computer Arithmetic Algorithms, 2nd ed., A. K Peters, Ltd., Natick, MA, 2002.
[23] J. Rebacz, E. Oruklu, J. Saniie, "Performance evaluation of multi-operand fast decimal adders,", 2009. MWSCAS '09. In: Proceedings of the 52nd IEEE International Midwest Symposium on Circuits and Systems, Aug. 2009, pp:535–538.
[24] M. Alioto, G. Palumbo, Impact of supply voltage variations on full adder delay: analysis and comparison, IEEE Transactions On Very Large Scale Integration (VLSI) Systems 14 (12) (Dec. 2006) 1322–1335.
[25] M. Alioto, G. Palumbo, Analysis and comparison on full adder block in submicron technology, IEEE Transactions On Very Large Scale Integration (VLSI) Systems 10 (6) (Dec. 2002) 806–823.

Vibhuti Dave received a B.E. degree in Electronics and Communication from Nirma Institute of Technology, India, M.S. in Electrical Engineering, and Ph.D. in Computer Engineering from Illinois Institute of Technology, Chicago in 2000, 2002, and 2007, respectively. In Fall 2007, she joined Penn State Erie, The Behrend College as Assistant Professor. Her research interests lie in the field of High Speed Computer Arithmetic and Computer Architecture.

Erdal Oruklu received a B.Sc. degree in ECE from Technical University of Istanbul, Turkey, M.Sc. in Electrical Engineering from Bogazici University, Istanbul, Turkey, and Ph.D. in Computer Engineering from Illinois Institute of Technology, Chicago in 1995, 1999, and 2005, respectively. Dr. Oruklu joined the department of Electrical and Computer Engineering at IIT in May 2005. Dr. Oruklu's research interests are reconfigurable computing, advanced computer architectures, hardware/software co-design, and embedded systems. The main focus of his studies is the research and development of system-on-a-chip (SoC) frameworks for FPGA and VLSI implementations of signal processing algorithms.



Jafar Saniie received a B.S. degree in Electrical Engineering from University of Maryland, M.S. in BME from Case Western Reserve University, and Ph.D. in Electrical Engineering from Purdue University, Chicago in 1974, 1977, and 1981, respectively. Professor Jafar Saniie joined the Department of Electrical and Computer Engineering at Illinois Institute of Technology in 1983. He is currently a Professor, Senior Advisor to the ECE Chair, and Director of the Ultrasonic Information Processing Laboratory.