Performance Evaluation of Flagged Prefix Adders for Constant Addition

Vibhuti Dave, Erdal Oruklu, and Jafar Saniie
Department of Electrical and
Computer Engineering
Illinois Institute of Technology
Chicago, Illinois 60616
Telephone: (312) 567–3400
Fax: (312) 567–8976
Email: davevib@iit.edu, erdal@ece.iit.edu, sansonic@ece.iit.edu

***Abstract*— The speed of the addition operation can play an important and complicated role in various signal processing algorithms. Parallel prefix adders have been one of the most notable among several designs proposed in the past. The advantage of utilizing these adders is the flexibility in implementing the tree structures based upon on the throughput requirements. Recently, a new technique has been proposed that utilizes the parallel prefix adder but modifies it to yield a new adder capable of performing simple increment and decrement operations. An extension to this adder has also been proposed enhancing the functionality of the same to allow the addition of any arbitrary number. This paper compares the performance of this unique adder design in terms of power, area, and delay by using the Brent-Kung, Kogge-Stone and the Ladner-Fischer tree structures. It also presents the advantage of using these kinds of adders over conventional adder designs to perform the same operation.**

I. Introduction

Addition circuits are utilized in a variety of applications ranging from cryptography, digital signal processing to the design of a simple ALU. Adder delays can determine the minimum clock cycle time in a processor. A small improvement in the design of an adder can result in significant improvements in the performance of an entire processor [12].

Many applications require the addition of two numbers followed by augmentation of the result by a constant. One example is changing the brightness of an image. In many cases, integrating this constant within a given architecture is not easy, resulting in utilizing two or more carry propagate adders in the design of the data path. The conditional sum adder provides a solution to the problem by generating a pair of sum and carry bits for each position. One pair assumes a carry-in of '1' and the other assumes a carry-in of '0'. Subsequently, a tree of multiplexors are used to select the appropriate sum and carry values. However, the conditional sum adder suffers from large fan-out and large area constraints [11] [13] [15].

Parallel prefix adders [8] have proven to be particularly attractive due to their regular structures and efficient design, compared with the prohibitive structures of carry lookahead and conditional sum adders. The underlying technique for these adders is to express addition as a prefix computation [10]. Using prefix computations offers the flexibility of having more than one implementation for intermediate structures within the adder, allowing trade-offs between the amount of internal wiring and the fan-out of intermediate nodes, therefore, resulting in a more advantageous combination of speed, area and power. The parallel prefix adders can be modified to yield a new design called the flagged prefix adder [3], [4]. The flagged prefix adder uses a simple technique [5] of inverting only selected sum bits to derive increment *(A+B+1)* and decrement *(A+B-1)* results in addition to normal addition *(A+B)* and subtraction *(A-B)* outcomes. This architecture can be further extended with small modifications in hardware [14] to invert the required sum bits and obtain the result of adding any arbitrary constant *(A+B+M)* following the addition/subtraction of two numbers.

This paper investigates the effects of the hardware modifications to the flagged prefix adder, for realizing arbitrary constant addition and subtraction with different prefix tree architectures [8]. The concept of the flagged prefix adder and the required modifications are reviewed in Section II. The three different prefix tree adder architectures considered in this paper are presented in Section III. In Section IV, the trade-offs in area, speed and power are evaluated among the enhanced prefix adder architectures. Area, delay and power numbers are also provided for a carry save adder [17] and a dual adder [3] to verify the advantage of the proposed architecture. The final conclusions are presented in Section V.

II. Background

The key to fast addition is to calculate the carry signals for all bit positions in parallel [16]. The recurrence relationship presented in Eq. 1 achieves this conveniently by introducing the *generate* or *g* signal given by $g_i = a_i \cdot b_i$, and the *propagate* or *p* given by $p_i = a_i + b_i$, where *i* represents the bit position.

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

The parallel prefix adder accomplishes the same by expressing binary carry propagate addition as a prefix computation. Prefix adders perform an n-bit addition in time *O(logn)*, using area

O(nlogn) [2]. Parallel prefix logic combines *n* inputs using an arbitrary associative operator ∘ to *n* outputs, so that the outputs Sum_i depend only on the input operands [6]. The parallel prefix adder computes the sum in three stages. This is illustrated in the block diagram in Fig. 1 [14].

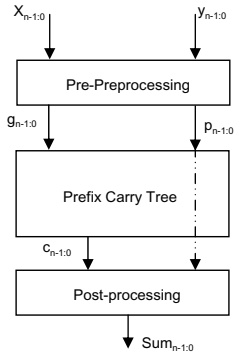

Fig. 1. Block Diagram of Parallel Prefix Adder

In the block diagram, illustrated in Fig. 1 *x* and *y* represent the *n* bit operands. *p* and *g* represent the *generate* and *propagate* signals described earlier. These signals are utilized to compute the *carries* through the recurrence equation given in Eq. 1. The prefix carry tree is an interconnection of a number of *black*, *gray* and *buffer* cells where the logic for each cell is illustrated in Fig. 2.

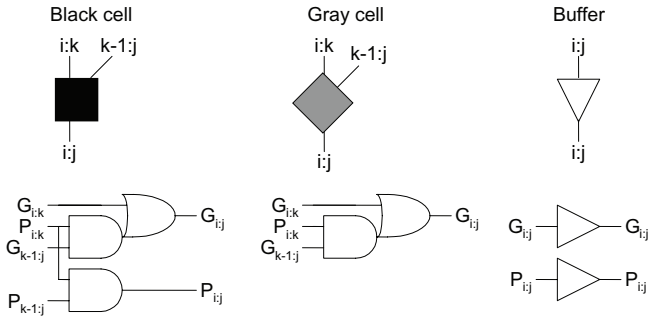

Fig. 2. Logic gates within the Prefix Carry Tree

The *black* cell is a complex logic gate that performs the associative ∘ dot operation. $G_{m:n}$and $P_{m:n}$ represent the *Group Generate* and *Group Propagate* signals across the bits from significance *m* up to and including significance *n* respectively [4]. The inputs to these cells come from the pre-processing stage of the adder. The *gray* cell is similar to the *black* cell, except that it does not output the *Group Propagate* signal. These cells are connected to form a multilevel tree structure. The output of the tree is then passed on to the post processing stage to produce the final sum.

A. Flagged Prefix Addition

The flagged prefix adder [3] [4] is a slightly modified version of a conventional prefix adder . The prefix carry tree in the modified version computes the so called *flag* bits in addition to the carry signals. These flag bits are utilized to invert the required sum bits. This can be better understood with the following example. Here, operand *x=9* and operand *y = 78* [14].

x	=	0	0	0	0	1	0	0	1
y	=	0	1	0	0	1	1	1	0
Sum	=	0	1	0	1	0	1	1	1
F	=	0	0	0	0	1	1	1	1
Sum+1	=	0	1	0	1	1	0	0	0

The result *Sum+1* is a result of XORing the flag bits, *F* and the *Sum* bits. The flag bits are shown to be easily generated from the prefix carry tree [1]. The flag bits are related to the *Group Propagate* signals [3] and therefore they can be obtained at the output of the prefix carry tree, while the carry is being computed. Consequently, computation of the flag bits does not affect the critical path. The block diagram of a flagged prefix adder is shown in Fig. 3 [14].

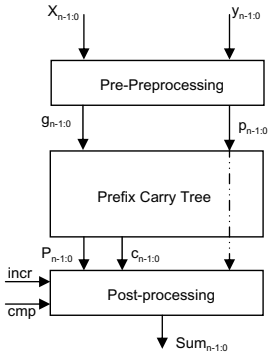

Fig. 3. Block Diagram of a Flagged Prefix Adder

The difference between the prefix adder shown in Fig. 1 and the flagged prefix adder illustrated in Fig. 3 is the output of the prefix carry tree. In the second case, the prefix carry tree is modified to output, $P_{n-1:0}$, the *Group Propagate* signal as shown in the figure. Also, two new inputs are introduced named as, *incr* and *cmp*. These are two control bits to select and invert the appropriate sum bits to achieve the desired result. A straightforward implementation of the flagged inversion logic is shown in Fig. 4 [3]. Therefore, the modifications required to achieve the design of the flagged prefix adder are as follows [3]:

- Convert the *gray* cells of a conventional prefix carry tree to *black* cells, in order to obtain the *Group Propagate* signals at the output
- A minimal amount of additional logic is required in order to compute the flag bits from the *Group Propagate* signals
- The necessary Flagged Inversion Logic will also be required to invert the appropriate sum bits depending on the control bits, *incr* and *cmp*.

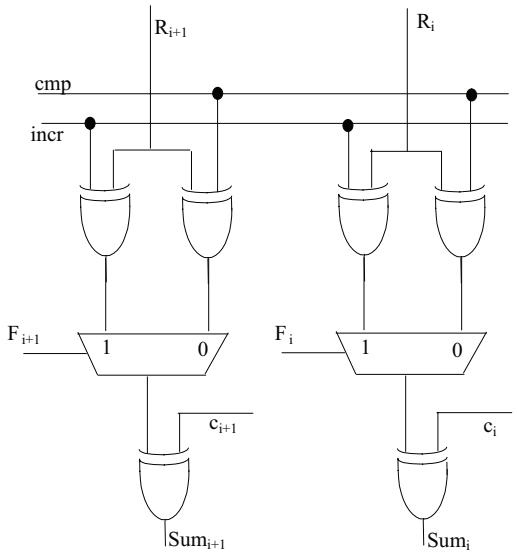

Fig. 4. Flagged Inversion Logic

B. Enhanced Flagged Prefix Adder

As presented in [14], the flagged prefix adder can be further extended to include another function of adding an arbitrary constant to the sum of two input operands. Assume that *R* is the result of adding two arbitrary inputs *A* and *B,* and *R* needs to be augmented/decremented by a value, *M*. The full adder equations can be written as

$$Sum_k = R_k \oplus M_k \oplus c_k$$

$$c_{k+1} = R_k \cdot M_k + R_k \cdot c_k + M_k \cdot c_k \tag{2}$$

Here, *k* represents the bit position.

Utilizing these new equations, the new sum needs to be computed such that $R + M = R \oplus F$ 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. The concept behind computation of the flag bits is similar to the one used in a conditional sum adder. Two bits of the constant are examined to determine whether or not the carry bit from the constant affects the current position.

$$c_{k+1} = \begin{cases} R_k \cdot c_k, & if\ M_k = 0 \\ R_k + c_k, & if\ M_k = 1 \end{cases} \tag{3}$$

$$F_k = \begin{cases} c_k, & if\ M_k = 0 \\ \overline{c_k}, & if\ M_k = 1 \end{cases} \tag{4}$$

Assuming M_k= *0* and M_{k-1}= *1*, utilizing the relationships in equations 2, 3 and 4, we obtain

$$c_{k+1} = R_k \cdot c_k \quad if\ M_k = 0$$

$$c_k = R_{k-1} + c_{k-1} \quad if\ M_{k-1} = 1$$

$$\therefore c_{k+1} = R_k \cdot F_k$$

$$\therefore c_k = R_{k-1} + \overline{F_k - 1} \tag{5}$$

Table I shows the complete set of equations for both, carry and the flag

TABLE I
OUTPUT LOGIC FOR SELECTION OF REQUIRED RESULT [14]

M_k	M_{k-1}	c_k	F_k
0	0	$R_{k-1} \cdot F_{k-1}$	$R_{k-1} \cdot F_{k-1}$
0	1	$R_{k-1} + F_{k-1}$	$R_{k-1} + F_{k-1}$
1	0	$R_{k-1} \cdot F_{k-1}$	$\overline{R_{k-1} \cdot F_{k-1}}$
1	1	$R_{k-1} + F_{k-1}$	$\overline{R_{k-1} \cdot F_{k-1}}$

However, the computation can be simplified by utilizing the output carry signals from the prefix tree and selecting the appropriate logic function based on the output carry. The flag equations can be rewritten as shown in Table II.

TABLE II
MODIFIED OUTPUT LOGIC FOR SELECTION OF REQUIRED RESULT UTILIZING CARRY PRODUCED FROM THE PREFIX CPA [14]

M_k	M_{k-1}	$F_k, (c_k = 0)$	$F_k, (c_k = 1)$
0	0	$R_{k-1} \cdot F_{k-1}$	$\overline{R_{k-1} \cdot F_{k-1}}$
0	1	$R_{k-1} + F_{k-1}$	$\overline{R_{k-1} \cdot F_{k-1}}$
1	0	$\overline{R_{k-1} \cdot F_{k-1}}$	$R_{k-1} + F_{k-1}$
1	1	$\overline{R_{k-1} \cdot F_{k-1}}$	$R_{k-1} \cdot F_{k-1}$

In order for the equations, to be computed correctly, the following initial conditions are assumed

$$R_{-1} = M_{-1} = F_{-1} = 0 \tag{6}$$

$$F_0 = M_0$$

Continuing with the same example as presented before, with *x*=9, *y*=78 and *M*=0011_1001$_2$= 57$_{10}$, we get

M	=	0	0	1	1	1	0	0	1
x	=	0	0	0	0	1	0	0	1
y	=	0	1	0	0	1	1	1	0
Sum	=	0	1	0	1	0	1	1	1
F	=	1	1	0	0	0	1	1	1
Sum+57	=	1	0	0	1	0	0	0	0

In order to accomplish the extra functionality as a part of the flagged prefix adder, minimal extra logic depending on the specific constant needs to be included. The flag equations rely on the carry produced from the prefix carry tree, thus not significantly affecting the critical path. If another constant is chosen, the logic for the flag changes according to Table II. Fortunately, the logic functions implementing the flag can be pre-computed. It is however possible that specific designs incorporating a particular constant may incur more delay than other constants.

This paper investigates the performance of the approach described above implemented in three different prefix structures

and the trade-offs in area, power and delay. The implementation will determine the tree that would be the most appropriate to use in order to exploit the usefulness of the design. The different tree structures considered in this paper are described in Section III and the implementation results in Section IV.

III. Prefix Carry Tree Implementations

Prefix adders have become popular due to their regular structures and the fast computation of carry values compared to the carry lookahead adders. All prefix structures exploit the associativity of the dot operator [17], which is defined according to the following equation

$$(g,p) \circ (g',p') = (g + (p \cdot g'),\ p \cdot p') \tag{7}$$

Here, *g,g',p,* and *p'* represent pairs of bit *generate* and *propagate* signals. This in turn leads to regular and easily implementable structures as described in [2], [10], [9]. All prefix trees, use the same *black* and *gray* cells as shown in Fig. 2 to produce the carry values for each bit position at the same time. However, the performance of each tree differs due to the difference in interconnectivity of these gates, amount of fanout required at each level, and the wiring density.

Addition was successfully expressed as a prefix computation by Ladner and Fischer [10]. The corresponding prefix carry tree is shown in Fig. 5 . The Ladner-Fischer tree aims at

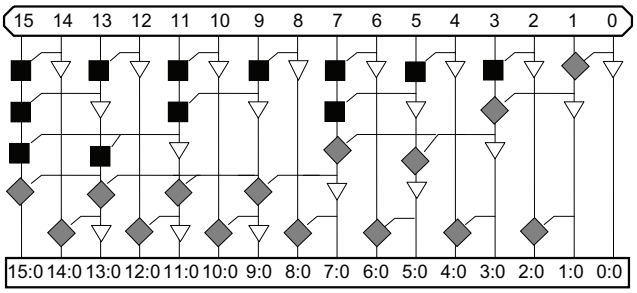

Fig. 5. Ladner-Fischer Tree

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 structure that addresses the fan out restrictions, but the logical depth of the tree is increased as seen in Fig. 6. Therefore the Brent-Kung tree [2] has a more regular structure, which is easy to implement in terms of chip design and wiring density. Another approach was proposed to target the fan out issue and that led to the Kogge-Stone tree [9], shown in Fig. 7. The Kogge-Stone structure limits the lateral logic fanout to unity [8] at each node, but increases the number of lateral wires at each level. This leads to an increase in area and also in the complexity of interconnections. However, in terms of speed, the adder performs better than the previous two designs.

Parallel prefix Adders utilizing each of these prefix trees are implemented in the $0.18\mu m$ System-on-Chip design flow

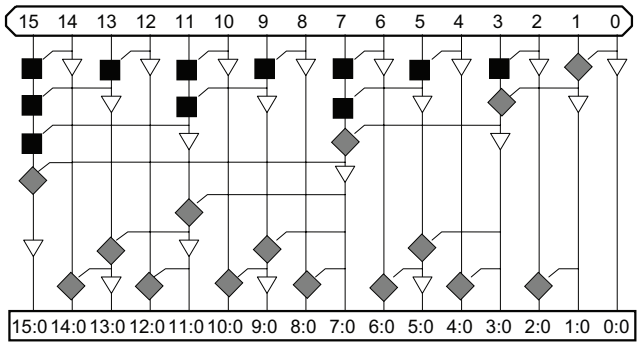

Fig. 6. Brent-Kung Prefix Tree

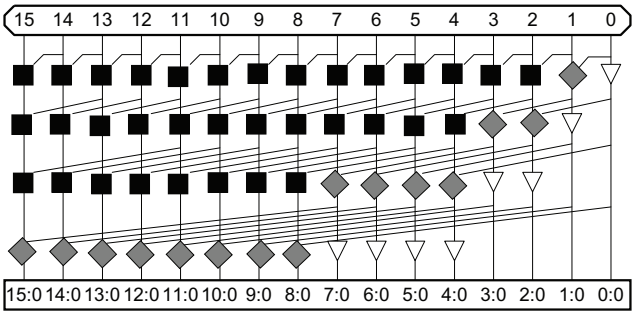

Fig. 7. Kogge-Stone tree

to investigate the power, area and delay trade-offs. Synthesis is performed with Cadence Build Gates and Encounter [7]. Layouts are generated for the adders and parasitically extracted to obtain accurate numbers for area, delay and power. The results presented are for 16-bit designs and represent worst-case input/output transitions.

TABLE III
Post-Layout estimates for Brent-Kung, Kogge-Stone and Ladner-Fischer Adders

Adder/Parameters	Area (mm^2)	Delay(ns)	Power(mW)
Brent-Kung	0.2756	16.825	5.63E-04
Ladner-Fischer	0.2763	14.025	5.81E-04
Kogge-Stone	0.4961	11.412	7.78E-04

As listed in Table III, each tree has an interesting power, delay, area trade-off relationship. The Brent-Kung adder sacrifices speed in order to get minimal area. However, the regularity of the structure and minimal fan out result in attractive power dissipation results. The Ladner-Fischer adder due to the least number of logic levels performs better in terms of speed by approximately 17% compared to the Brent-Kung adder. However, it lags behind in area and power dissipation, but the difference is not very significant since the number of *black* and *gray* cells required by each tree are approximately the same. A noticeable difference in performance, is observed in the design of the Kogge-Stone adder. The Kogge-Stone

structure due to the increased wire density required to connect the cells and the increased number of *black* cells, sacrifices area and power in order to achieve the best speed among all three designs. The latter is 32% times faster than the Brent-Kung design but dissipates 38% more power.

IV. Implementation of the Enhanced Flagged Prefix Adder

The prefix carry trees described in the previous section are implemented using the enhanced version of the flagged prefix adder. Each tree is modified in order to incorporate the flag logic according to Table II. In each prefix tree, the *gray* cells are converted to *black* cells in order to obtain the *Group Propagate* signals in addition to the *Group Generate* signals, since these signals are utilized to generate the flag bits for each position. Furthermore, the post processing stage is modified to include XOR gates that invert the appropriate sum bits based on the flag bits generated. Multiplexors can also be used instead. In order to better understand the advantages of this adder design, two additional adder structures were designed and implemented to perform the same operation. The first one is a multi- operand adder, the carry save adder [17] which avoids carry propagation by treating the intermediate carries as outputs instead of advancing them to the next higher bit position. This adder accepts three binary input operands,(*A*, *B*, and *M*) and produces a redundant result consisting of two binary numbers, *S*(sum bits) and *C*(carry bits). The final result is produced by utilizing a carry propagate adder to add *S* and *C*. The second option is the use of two consecutive stages of an adder. The first stage computes *A+B*. The second stage augments this result by *M*. This scheme is called the dual adder design due to the use of two adders. A Brent-Kung adder has been utilized for this architecture.

The results presented are for 16-bit designs and represent worst case input/output transitions. The value of *M* has been chosen as 57. The designs were implemented using TSMC $0.18\mu m$ technology and the synthesis was performed using the same tools as mentioned in Section III. The nominal operating voltage is 1.8V and simulation is performed at $T = 25°C$.

TABLE IV
Results for Flagged Prefix Adder

Adder/Parameters	Area(mm^2)	Delay(ns)	Power(mW)
Brent-Kung	0.2921	16.901	8.08E-04
Ladner-Fischer	0.2933	14.242	8.34E-04
Kogge-Stone	0.5462	11.526	1.12E-03
Carry-Save	0.3236	16.832	1.37E-03
Dual Adders	0.6011	28.344	1.84E-03

The extra logic required to generate the necessary flag bits depends on the constant chosen. The flag bit at every subsequent position is computed depending on the flag bit computed in the preceeding position. This results in a rippling structure at the last level of the design. The extra hardware, however will always comprise of *n* gates, where *n* is the size of the input. This can be easily concluded from the results presented in Table II. Therefore, although the logic changes depending on the constant, the amount of additional logic stays consistent for every prefix tree. The increase in area therefore, is not expected to vary significantly between different prefix structures. Internal wiring within the prefix trees could play a small role in increase in area. The critical delay of each adder depends on the prefix tree utilized and also the flag bit produced at each bit position. For a particular constant, the logic added to the critical path for each prefix architecture stays the same. For adding constants that have low amounts of entropy, this method achieves small amounts of additional hardware and high amounts of throughput without requiring an additional carry propagate adder.

For the presented simulation results the value of *M* was chosen as 57. It is observed that the performance of the flagged prefix adders is favorable compared to that of a carry-save adder or a dual adder to perform the same operation. The carry-save adder consumes more area compared to the Brent-Kung and Ladner-Fischer prefix architectures by approximately 11%. Although, it has a better number for area when compared with the Kogge-Stone adder, it lags behind in delay and power. The Kogge-Stone adder has the best performance in terms of speed among all adder designs. The dual adder design consumes approximately twice the area and has twice the delay due to the fact that two consecutive stages are used to perform the addition of three different numbers.

Comparing Tables III and IV, it can be noticed that there is a minimal increase in area and delay due to the changes made to the prefix tree and the extra logic required to generate the flag bits. However, the power dissipation rises by approximately 40% due to the increased fan out and added wire interconnectivity. The automated tools are used to optimize the circuit for area. However, custom layout implementations could reduce the required area since the flag logic could be implemented more efficiently at physical level. Due to the similarity in the structures of the Brent-Kung and Ladner-Fischer, the increase in area due to extra hardware is approximately 6% for both designs. A significant increase in area, by about 10% is observed upon changing the Kogge-Stone structure. The wiring becomes progressively complex once the *gray* cells are changed into *black* cells, increasing the wire density between the cells. The critical delay in all three structures is only slightly increased compared to the delay of plain prefix adder designs. The advantage of the proposed design lies in the fact that additional logic stays consistent with each prefix design, thereby allowing flexibility in the choice of which prefix tree to utilize. The choice of the prefix tree can be made based on what area, power, and delay requirements for the targeted application.

V. Conclusion

This paper describes the implementation of the enhanced version of flagged prefix adder using three different prefix carry tree structures. The impact of each prefix tree structure is examined for the enhanced flagged prefix adder design with

respect to area, speed and power. It is noticed, that the Brent-Kung adder performs the best in terms of power dissipation due to the restricted fan-out and the structural regularity that is not affected by the additional hardware. Also, the Kogge-Stone adder is a good compromise between speed and power since it provides the best results in terms of delay and the increase in power is approximately the same as for the Ladner-Fischer and Brent-Kung trees.

REFERENCES

[1] A. Beaumont-Smith, N. Burgess, S. Lefrere, and C. C. Lim. Reduced Latency IEEE floating point standard adder architectures. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 35–42, 1999.

[2] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transaction on Computers*, C-31:260–264, 1982.

[3] N. Burgess. The flagged prefix adder for dual addition. In *Proceedings of SPIE-the international society for Optical Engineering*, volume 3461, pages 567–575, 1998.

[4] N. Burgess. The Flagged Prefix Adder and its Applications in Integer Arithmetic. *Journal of VLSI Signal Processing*, 31(3):263–271, 2002.

[5] S. Cui, N. Burgess, M.J Liebelt, and K.Eshraghian. A GaAs IEEE Floating Point Standard Single Precision Multiplier. In *Proceedings of the 12th IEEE Symposium on Computer Arithmetic*, pages 91–97, 1995.

[6] M. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kauffmann, first edition edition, 2004.

[7] J. Grad and J.E. Stine. A Standard Cell Library for Student Projects. In *International Conference on Microelectronics Systems Education*, pages 98–99. IEEE Computer Society Press 2003, 2003.

[8] S. Knowles. A family of adders. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 30–34, 1999.

[9] P. M. Kogge and H. S. Stone. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22:783–791, 1973.

[10] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27:831–838, 1980.

[11] H. Lindkvist and P. Anderson. Techniques for Fast CMOS-based Conditional Sum Adders. In *Proceedings of the 1994 International Conference on Computer Design*, pages 626–635, October 1994.

[12] J. M. Rabaey, A. Chandrakesan, and B. Nikolic. *Digital Integrated Circuits, A Design Perspective*. Prentice Hall, second edition edition, 1995.

[13] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(6):226–231, 1960.

[14] J. Stine, C. Babb, and V. Dave. Constant Addition utilizing Flagged Prefix Structures. In *Seventh Euromicro Conference on Digital System Design*, 2004.

[15] A. Tyagi. A reduced-area scheme for carry-select adders. *IEEE Transactions on Computers*, 42:1163–1170, 1993.

[16] S. Winograd. On the time required to perform addition. *Journal of the ACM*, pages 277–285, April 1965.

[17] R. Zimmermann. *Binary Adder Architectures for Cell-based VLSI and their synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1997.