UACEE International journal of Advances in Electronics Engineering
Vol:1 Issue:1 ISSN 2278 - 215X

Parallel-Prefix Adder Architecture With Efficient Timing-Area Characteristic

G.Jyoshna M.Tech
JNT University
Anantapur
Email: jyoshnagirika_2006@yahoo.co.in

P.Murali Krishna M.Tech
JNT University
Anantapur
Email:murali.43@gmail.com

B.Doss (Ph.D)
Lecturer
JNT University
Anantapur

Abstract

Two-operand binary addition is the most widely used arithmetic operation in modern datapath designs. To improve the efficiency of this operation, it is desirable to use an adder with good performance and area tradeoff characteristics. This paper presents an efficient carry-lookaheadadder architecture based on the parallel-prefix computation graph. In our proposed method, we define the notion of *triple-carry-operator*, which computes the *generate* and *propagate* signals for a merged block which combines three adjacent blocks. We use this in conjunction with the classic approach of the *carry-operator* to compute the *generate* and *propagate* signals for a merged block combining two adjacent blocks. The timing-driven nature of the proposed design reduces the depth of the adder. In addition, we use a ripple-carry type of structure in the nontiming critical portion of the parallel-prefix computation network. These techniques help produce a good timing-area tradeoff characteristic. The experimental results indicate that our proposed adder is significantly faster than the popular Brent–Kung adder with some area overhead. On the other hand, the proposed adder also shows marginally faster performance than the fast Kogge–Stone adder with significant time savings.

Keywords – Arithmetic and logic structures, integrated timing circuits, logic design.

I. INTRODUCTION

Integrated Circuit (IC) technology has gone through a spectacular revolution in the last two decades. The number of transistors that can be integrated on a single die has been exponentially increasing with time following the Moore's Law. Driven by increased density that can support complex applications, higher speed and reduced cost MOS transistor are scaled to nanometer ranges. Working with scaled devices to make it physically work on silicon VLSI design engineers face lot of challenges such as low power, reduced time to market etc. In order to meet these challenges the other alternative adopted is working on proven technology with parallel processing to increase the overall system performance and
design reuse reduces the time to market.

The complexity and the performance requirement of the datapath operations implemented in systems-on-chips (SoCs) operations in modern integrated circuits, they tend to play a critical role in determining the performance of the design. Hence, developing efficient adder architecture (from the standpoint of timing, area, and power) is crucial to improving the efficiency of the design. Carry lookahead adders based on parallel prefix computation methods yield the fastest adders. There are several techniques proposed for the computation of the parallel prefix. In [1], Sklansky proposes one of the earliest tree-prefix algorithms for adders, where a tree structure is used to compute the intermediate signals. In the Brent–Kung (BK) approach [2], Brent and Kung design the prefix-computation graph in an area-optimal way and the Kogge–Stone (KS) architecture [3] is optimized for timing. In [4], another prefix-computation architecture is proposed, where the fan-out of gates increases with the depth of the prefix computation tree. In [5], a hybrid adder architecture based on BK and KS is proposed. In [6], a zero-deficiency prefix adder with minimal depth was introduced. In [7] and [8], the authors present new algorithms to construct a class of depth-size optimal parallel prefix circuits. In [9], a parallel prefix adder synthesis was introduced, which performs two-step area minimization under given timing constraints. In [10], Choi and Swartzlander present a one-shot batch process that generates a wide range of designs for a group of parallel prefix adders. In [11], Dimitrakopoulos and Nikolos save one-logic level of implementation leading to faster performance of the parallel-prefix addition. In [12], a performance evaluation analysis was performed between flagged prefix adders with the other well-known prefix adders. In [13], Liu *et al.* propose an algorithmic approach to generate an irregular parallel-prefix adder. In [14], Lin *et al.* use domino logic to generate efficient parallel-prefix architecture. Our approach is different from all the other approaches mentioned earlier, because we use combination of two types of merged blocks.

We propose a new design of an efficient addition block based on the parallel-prefix computation technique. In our approach, we use the notion of computing the *generate* and *propagate* signals for a merged block combining three adjacent blocks. We use this in conjunction with the classic approach of computing *generate* and *propagate* signals for a merged block combining two adjacent blocks. Our design is timing driven in the timing critical path. At the same

time, we optimize for area in the nontiming critical path. This is another novel aspect of our proposed approach.

Prefix adders are based on parallel prefix circuit theory which provides a solid theoretical basis for wide range of design trade -offs between delay, area and wiring complexity. This dissertation first presents an algorithm for prefix computation under the condition of non -uniform input signal arrival. To obtain the algorithm, the structure of prefix circuits is analyzed and a generalized circuit structure that is composed of two parts, a full -product generation tree and sub -product generation trees, is proposed. For the full - product generation tree, a delay optimized design algorithm is proposed and its optimality is shown. The proposed algorithm is easy of implement and fast in run -time due to its greedy strategy and it ensures the minimum depth prefix circuit design with the Ladner Fischer strategy. This dissertation also presents a one -shot batch process that generates a wide range of designs for a group of parallel prefix adders. The prefix adders are represented by two-dimensional matrixes and two vectors. This matrix representation makes it possible to compose two functions for gate sizing which calculate the delay and the total transistor width of the carry propagation graph of adders. After gate sizing, the critical path net list of the
carry propagation graph is generated from the matrix representation for spice delay calculation. The process is illustrated by generating sets of delay and total transistor width pairs for 32 -bit and 64 -bit cases.
For each bit (i = 0 to (n-1))

$Si = ai \oplus bi \oplus Carryi$

$Carryi+1 = (ai \cap bi) U (bi \cap Carryi) U (Carryi \cap ai)$

For each bit i of the adder, Generate (Gi) indicates whether a carry is generated from that bit.

$Gi = ai \cap bi$

For each bit i of the adder, Propagate (Pi) indicates whether a carry is propagated through that bit.

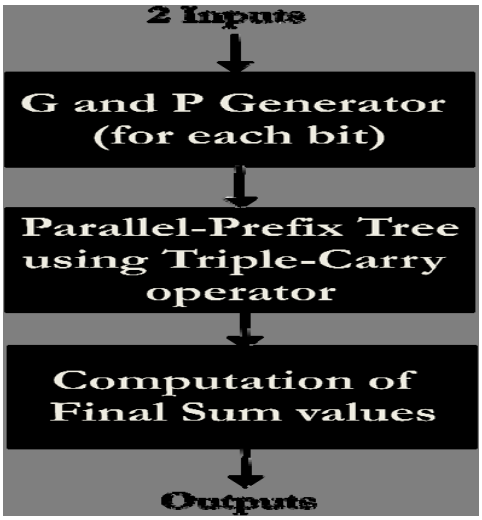
$Pi = ai \oplus bi$

Generate and Propagate concept is extendable to blocks comprising multiple bits.

Fig: Flow chart

II Approach:

Use 2-input XOR and AND gates to compute Gi and Pi values.



Use triple-carry operator in parallel-prefix tree to compute Carryi values.
Use Pi and Carryi to compute final Sumi values.

In our approach, we use the traditional way of computing the Generate (Gi) and Propagate
(Pi) for each bit.
$Gi = ai \cap bi$
$Pi = ai$? bi
If Gi is equal to 1, that indicates a Carryi+1 signal equal to 1'b1 (logic-1) is generated from the
ith bit. If Pi is equal to 1, that indicates the Carryi gets fed to the Carryi+1 signal.

a)Single Carry Operation:- A carry/majority circuit, comprising a plurality of differentialtransistor pairs coupled in parallel and forming a pair of output nodes, with a single parallel gated level. Current is steered through a leg of the transistor pair having a higher input voltage. A carry circuit is typically used in arithmetic units, such as adder or subtractors, to process a carry operation in order to transfer a carry signal to the following carry operation. The carry circuits can be arranged to form other devices such as accumulators which can be further expanded to such devices as direct digital synthesizers (DDS).

In a floating point addition or subtraction procedure two shift operations of the operand fraction may be required. The first shift operation, based on the difference between the operand exponent arguments, involves aligning one of the operand arguments so that the addition or subtraction procedure between the operand fractions can be performed. In order to complete the associated computations correctly, it is necessary to

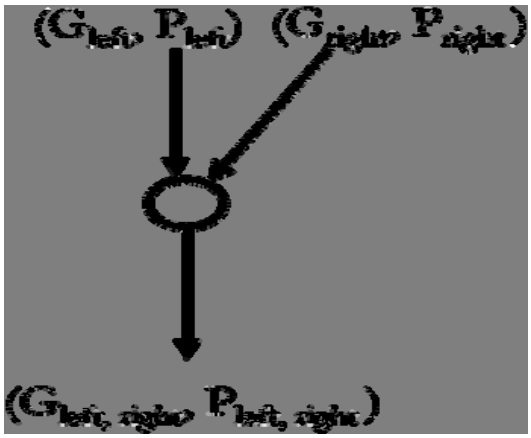


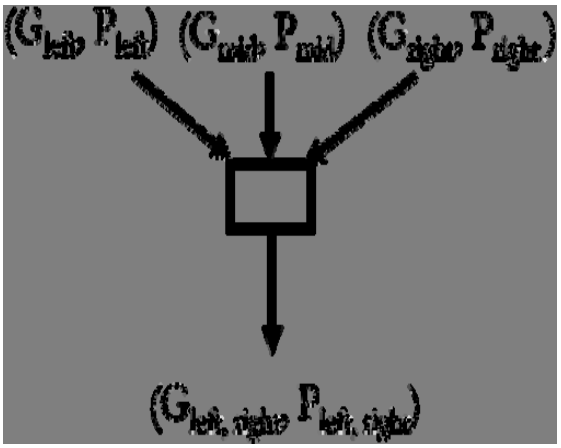
Fig: Single carry operator

know if any of the fraction positions removed from the fraction by the shift operation include non-zero signals.

If two blocks (comprising one or more bits) have the GP value-pairs as (Gleft, Pleft) and (Gright, Pright), then the combined block has the GP values as follows:
Gleft, right = Gleft (Pleft ∩ Gright)
Pleft, right = Pleft ∩ Pright
This operation is performed by a carry-operator or o-operator.

b)Tripple Carry Operation:- Triple-carry-operator, which computes the *generate* and *propagate* signals for a merged block which combines three adjacent blocks. We use this in conjunction with the classic approach of the carry-operator to compute the *generate* and *propagate* signals for a merged block combining two adjacent blocks. The timing-driven nature of the proposed design reduces the depth of the adder.

If three blocks (or bits) have the GP value-pairs as (Gleft, Pleft), (Gmid, Pmid) and (Gright, Pright), then the combined block generates a Carry only if:
• Left block generates a Carry OR
• Middle block generates a Carry and Left block propagates that OR
• Right block generates a Carry and both Middle and Left blocks propagate that Carry.
The combined block propagates only if:
• Each of the three blocks propagates the input Carry.
• If three blocks (consisting of one or more bits) have the GP value-pairs as (Gleft, Pleft), (Gmid, Pmid) and (Gright, Pright), then the combined block has the GP values as follows:
Gleft, right = Gleft (Pleft ∩ Gmid) (Pleft ∩ Pmid ∩ Gright)
Pleft, right = Pleft ∩ Pmid ∩ Pright .

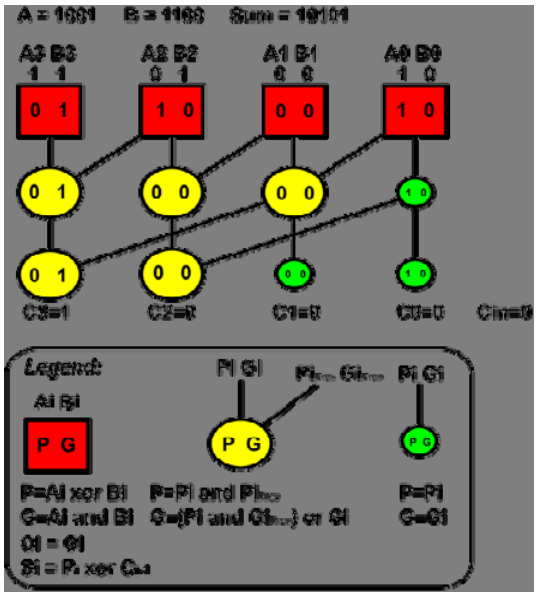


• This operation is performed by a triple-carry operator or o3-operator.

Fig : Tripple carry operator

Typically, delay of a triple-carry operator is about 110% to 130% of the delay of a traditional carry-operator. Area of a triple-carry operator is about 150% to 180% of the area of a traditional carry-operator.

c)Advantages of Triple Carry Operator
• Triple carry operator to avoid the problem of high fan-out nets. To maintain this strictlimit on fanouts.
• Triple carry operator reduces the depth along that path (at the expense of additional
hardware) and improves the performance of parallel prefix adder.
• Use triple-carry operator in the parallel-prefix computation tree to reduce delay of the
critical-path this delay characteristic makes triple-carry operator an efficient choice in
the parallel prefix network.



• We only use "*Triple carry" operator in the timing critical portion* of the parallel prefix tree.

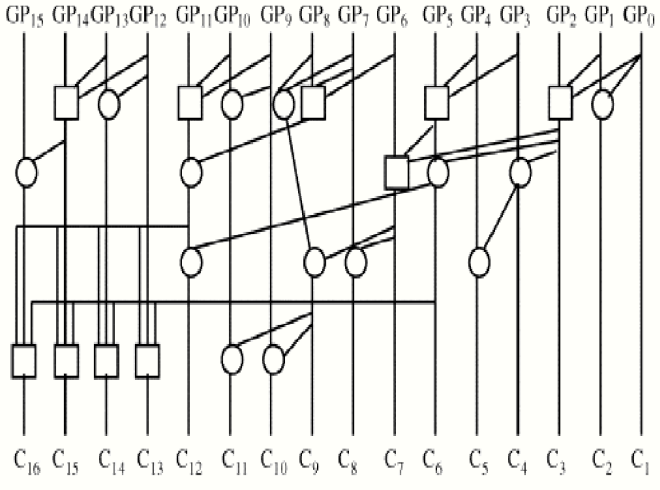


Fig: Example of 4-bit

Fig: Proposed parallel prefix network (for input width of 16 bits).

III. EXPERIMENTAL RESULTS

To collect different data points regarding the quality of results for the adder blocks, we used the following variations.

• Adder blocks of different input widths:

We have used adders having different input widths. In Table I, we have shown the final results for adders having input bitwidths (n) equal to 16, 24, 32, 48, and 64 bits. We refer to these blocks as Adder-16, Adder-24, Adder-32, Adder-48, and Adder-64, respectively.

• Different technologies and libraries:

two commercial libraries (L1 and L2) for 0.13 μ;
two commercial libraries (L3 and L4) for 0.09 μ ;

• Different input arrival time constraints:

We used the following input arrival time constraints.

Different input bits of signals a and b arrive at different times. The motivation for this is as follows. There exists an adder sub-block inside every arithmetic sum-of product (SOP) and multiplier block. Due to the wide usage of SOP and multipliers in the modern digital designs, the performance of this adder block is crucial to determine the performance of the design. Thus, we model this timing constraint [15]. Since an adder is an internal part of a SOP and multiplier block, the arrival times of different inputs of the adder block are not identical. Hence, we cannot directly write timing constraints to control the arrival times for the inputs of the adder. As a result, we specified the arrival time constraints for the inputs of the SOP and the multiplier. Once the input arrival times are specified for SOPs and multipliers, the synthesis tool propagates the arrival times through each sub-block inside the SOP and multiplier. We then report the actual arrival-time numbers to the input of the *adder sub-block inside SOP and multiplier*. In this manner, we collected significant amount of data on the arrival- times of the adder inputs. From this arrival-time data, we derived the following equation. We believe that this equation closely represents the actual arrival timing-constraint for the adder sub-blocks inside real-life SOPs and multiplier blocks. We refer to this category of timing constraints as Ar(lte)ar . Let us denote Arr(ai) as the arrival time of the signal ai. Assuming that k is a constant and δ is the delay of the fastest two-input AND gate in the technology library, the following is the Arr(late) timing constraint (n is the width of the adder inputs):

(Arr ai) = δk ; 0≤ i ≤[3n/5]
(Arr ai) = [3n/5]kδ - (i- [3n/5]) kδ;
[(3n/5)] < i < n
Arr(bi) =δk ; 0 ≤i ≤[3n/5]
Arr(bi) =[3n/5]kδ - (i – [3n/5]) kδ;
[(3n/5)] <:i < n

All input bits of the signals a and b arrive at the same time.We refer to this constraint as Asmarer)(. If k is a constant number, then the Arr(same) constraint can be represented as

Arr(ai) =k; 0≤i <n
Arr(bi) =k; 0≤i <n:

We have implemented the BK adder [2], the KS adder [3], and our proposed adder for different operand widths. We optimized each of the architectures by using a best-in-class commercially available datapath synthesis tool (run on a workstation with dual 2.2-GHz processors, 4 GB memory, and RedHat 7.1 Linux). The synthesis tool performed the operations like technology-independent optimizations, constant propagation, redundancy removal, technology mapping, timingdriven optimization, area-driven optimization, incremental optimization, etc. Due to the licensing agreements, we are unable to mention the name of the commercial tool we used. In Table I, we present the post-synthesis worst-case delay and the total area results for the adder block for each of the three architectures (as reported by the synthesis tool). To compute worst-case delay, the static timing computation engine inside the datapath synthesis tool was used. To compute total area, the technology library cell information was used.

TABLE I
DELAY AND AREA COMPARISON OF ADDER BLOCKS GENERATED BY BK, KS, AND OUR APPROACH

Design	Technology Library	Timing Constraint	Worst-case Delay (ps)					Area (μ^2)				
			BK	KS	Ours	BK v/s Ours (%)	KS v/s Ours (%)	BK	KS	Ours	BK v/s Ours (%)	KS v/s Ours (%)
Adder-16	L_1 (0.13μ)	Arr(late)	2863	2178	2268	20.78%	-4.13%	1664	2106	1723	-3.78%	18.19%
Adder-24	L_1 (0.13μ)	Arr(late)	3128	2431	2503	19.98%	-2.96%	2371	3758	2609	-5.82%	30.57%
Adder-32	L_1 (0.13μ)	Arr(late)	3194	2595	2582	19.16%	0.5%	3273	4841	3612	-10.36%	25.38%
Adder-48	L_1 (0.13μ)	Arr(late)	3591	2816	2739	23.72%	2.73%	4835	7521	5314	-9.9%	29.34%
Adder-64	L_1 (0.13μ)	Arr(late)	3708	2938	2815	24.08%	4.19%	6780	11306	7296	-7.61%	35.46%
Adder-16	L_3 (0.09μ)	Arr(late)	1752	1363	1447	17.41%	-6.16%	7320	10926	8061	-10.12%	26.22%
Adder-24	L_3 (0.09μ)	Arr(late)	1986	1529	1572	20.84%	-2.81%	10462	17932	11196	-7.01%	37.56%
Adder-32	L_3 (0.09μ)	Arr(late)	2041	1615	1621	20.57%	0.37%	14187	22971	14820	-4.46%	35.48%
Adder-48	L_3 (0.09μ)	Arr(late)	2314	1852	1801	22.16%	2.75%	20719	29568	22139	-6.85%	25.12%
Adder-64	L_3 (0.09μ)	Arr(late)	2469	1997	1886	23.61%	5.56%	29619	40211	31217	-5.39%	22.37%
Adder-16	L_2 (0.13μ)	Arr(same)	2481	1803	1924	22.45%	-6.71%	2682	3658	3022	-12.67%	17.38%
Adder-24	L_2 (0.13μ)	Arr(same)	2847	2148	2173	23.67%	-1.16%	3729	6409	4107	-10.13%	35.91%
Adder-32	L_2 (0.13μ)	Arr(same)	2963	2207	2245	24.23%	-1.72%	4911	7638	5780	-17.69%	24.32%
Adder-48	L_2 (0.13μ)	Arr(same)	3406	2742	2569	24.57%	6.31%	7438	12428	8269	-11.17%	33.46%
Adder-64	L_2 (0.13μ)	Arr(same)	3622	2881	2672	26.22%	7.25%	8576	17869	9482	-10.56%	46.93%
Adder-16	L_4 (0.09μ)	Arr(same)	2174	1563	1629	25.06%	-4.22%	1983	2639	2148	-8.32%	18.61%
Adder-24	L_4 (0.09μ)	Arr(same)	2559	1782	1737	32.12%	-2.52%	2841	4671	3294	-15.94%	29.47%
Adder-32	L_4 (0.09μ)	Arr(same)	2682	1856	1883	29.79%	-1.45%	4062	6119	4437	-9.23%	27.48%
Adder-48	L_4 (0.09μ)	Arr(same)	2971	2338	2176	26.75%	6.92%	6028	10593	6692	-11.01%	36.82%
Adder-64	L_4 (0.09μ)	Arr(same)	3065	2406	2252	26.52%	6.4%	8347	14172	9163	-9.77%	35.34%
Adder-16	L_4 (0.09μ)	Arr(late)	2319	1781	1857	23.19%	-4.26%	2168	2914	2396	-10.52%	17.78%
Adder-24	L_4 (0.09μ)	Arr(late)	2916	2047	2071	29.80%	-1.17%	3104	5027	3582	-15.39%	28.74%
Adder-32	L_4 (0.09μ)	Arr(late)	3102	2263	2194	27.04%	3.04%	4637	7158	4927	-6.25%	31.16%
Adder-48	L_4 (0.09μ)	Arr(late)	3684	2854	2689	22.53%	5.78%	7382	12491	7819	-5.92%	37.40%
Adder-64	L_4 (0.09μ)	Arr(late)	4259	3286	3061	22.84%	6.84%	9841	16872	10743	-9.16%	36.32%
Average						23.96%	0.77%				-9.39%	29.71%

In Table I, we report 25 sets of data points for adders of different widths, timing constraints, and technology libraries. On an average, our proposed approach results in a 23.96% faster adder (column 7 of Table I), with 9.39% area penalty (colum 12). When comparing with the KS adder, then our proposed approach results in a marginally (0.77%) faster implementation (column 8), with a significant (29.71%) area improvement (column 13). Note that like the BK and KS approaches, our approach generates the same structure irrespective of the input arrival timing constraints. Then, depending on the arrival timing constraint, the technology mapping algorithms will choose different technology cells to yield different final worst delay (and area) numbers. To verify the correlation of post-synthesis experimental data with the post place-and-route data, we performed placement and routing on one Adder-32 and one Adder-64 design. For these two testcases, the average post-routing worst delay of BK adder, KS adder, and our proposed adder are (normalized to the worst delay of the BK adder): 1.0, 0.78, and 0.76, respectively. Similarly, the post-routing total area of the BK adder, KS adder, and our proposed adder are (normalized to the area of the BK adder): 1.0, 1.34, and 1.07, respectively. The individual results for the Adder-32 and Adder-64 designs correlate closely with the post-synthesis numbers reported in Table I. These results after place and route confirm our conclusion about the efficient timing area characteristic of our approach. For the reference purposes, we implemented the ripple adder and measured its delay and area numbers across all our adder designs, libraries, and timing constraints. The experimental data showed that, on an average, our proposed adder is about 62% faster and 239% larger than the ripple adder.

We also performed some additional experimentation by using different values of δ in the equation for Ar(late) r . The modified values of δ we tried are equal to: 1) a two-input XOR gate delay from the technology library; 2) a two-input OR gate delay; 3) an inverter gate delay; 4) 1 (constant number). In each of these cases, the resulting delay and area numbers of our adders exhibit substantially same timing area characteristics as reported in Table I.

IV. CONCLUSION

we have presented a hybrid approach of implementing an adder block based on the fast parallel prefix architecture. The proposed adder exhibits very efficient timing area tradeoff characteristics. Our hybrid architecture is based on the *triple-carry operator* ("*o3*") and the classical *carry-operator* ("*o*"). It works seamlessly with adder blocks of different widths and across different technology domains (0.13 _, 0.09 _, etc.). The experimental results indicate that our proposed adder is significantly faster than the popular BK adder with some area overhead. On the other hand, the proposed adder also shows marginally faster

performance than the fast KS adder with significant time savings.

REFERENCES

[1] J. Sklansky, "Conditional sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, no. 6, pp. 226–231, 1960.

[2] R. P. Brent and H. T.Kung, "Aregular layout for parallel adders," *IEEE Trans. Comput.*, vol. 31, no. 3, pp. 260–264, Mar. 1982.

[3] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 783–791, Aug. 1973.

[4] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[5] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proc.8th Symp. Comput. Arithmetic*, 1987, pp. 49–56.

[6] H. Zhu, C. K. Cheng, and R. Graham, "On the construction of zerodeficiency parallel prefix circuits with minimum depth," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 2, pp. 387–409, 2006.

[7] Y. C. Lin and C. C. Shih, "A new class of depth-size optimal parallel prefix circuits," *J. Supercomput.*, vol. 14, no. 1, pp. 39–52, 1999.
IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, VOL. 16, NO. 3, MARCH 2008 331

[8] Y. C. Lin and C. Y. Su, "Faster optimal parallel prefix circuits: New algorithmic construction," *J. Parallel Distrib. Comput.*, vol. 65, no. 12, pp. 1585–1595, 2005.

[9] T. Matsunaga and Y. Matsunaga, "Area minimization algorithm for parallel prefix adders under bitwise delay constraints," in *Proc. 17th Great Lakes Symp. VLSI*, 2007, pp. 435–440.

[10] Y. Choi and E. E. Swartzlander, Jr, "Parallel prefix adder design with matrix representation," in *Proc. 17th IEEE Symp. Comput. Arithmetic (ARITH)*, 2005, pp. 90–98.

[11] G. Dimitrakopoulos and D. Nikolos, "High-speed parallel-prefix VLSI ling adders," *IEEE Trans. Comput.*, vol. 54, no. 2, pp. 225–231, Feb.2005.

[12] V. Dave, E. Oruklu, and J. Saniie, "Performance evaluation of flagged prefix adders for constant addition," in *Proc. IEEE Int. Conf. Electro/ inf. Technol.*, 2006, pp. 415–420.

[13] J. Liu, S. Zhou, H. Zhu, and C. K. Cheng, "An algorithmic approach for generic parallel adders," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2003, pp. 734–740.

[14] R. Lin, K. Nakano, S. Olariu, and A. Y. Zomaya, "An efficient parallel prefix sums architecture with domino logic," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 9, pp. 922–931, Sep. 2003.

[15] P. F. Stelling and V. G. Oklobdzija, "Design strategies for optimal hybrid final adders in a parallel multiplier," *J. VLSI Signal Process.*, vol. 14, no. 3, pp. 321–331, 1996.