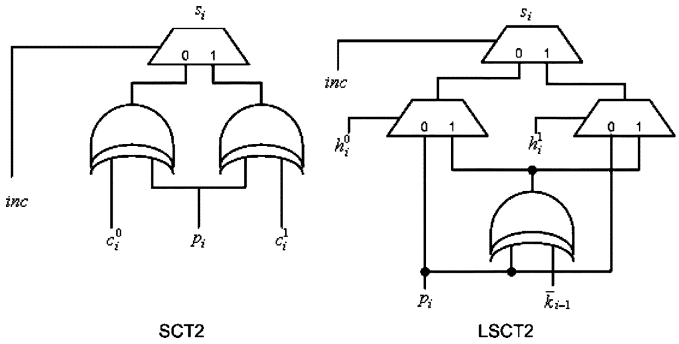

Fig. 5. Sum logic for late increment with critical INC signal.

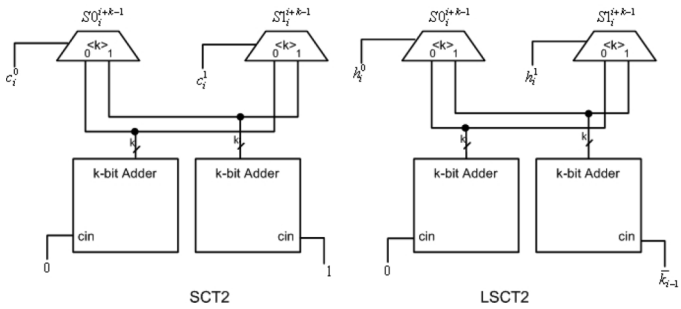

Fig. 6. Sum logic for speculative sum generation.

pipeline stage can be added after the 64-bit additions. Fig. 6 shows the sum logic for such a case. The second design is based on [12]. To reduce the cycle time, $(c_i^0, c_i^1)/(h_i^0, h_i^1)$ needs to be generated as quickly as possible and SCT2/LSCT2 are the best choices for this application because a pipeline design is generally intended for maximizing the throughput rather than for minimizing the area. Between SCT2 and LSCT2, LSCT2 is a bit faster as stated in Section II-C.

IV. Conclusion

A formal framework for speculative carry generation is proposed. The framework is successfully applied to adders using the Ling carry as well as adders with a normal carry. Including two Ling carry cases, three new speculative prefix schemes are introduced.

Several applications for speculative carry generation are presented to show how this work broadens the design space of speculative prefix adders.

References

[1] N. Burgess, "Prenormalization rounding in IEEE floating-point operations using a flagged prefix adder," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 13, no. 2, pp. 266–277, Feb. 2005.

[2] N. Burgess, "The flagged prefix adder and its application in integer arithmetic," *J. VLSI Signal Process*, vol. 31, pp. 263–271, 2002.

[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. 786–793, Aug. 1973.

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

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

[6] S. Knowles, "A family of adders," in *Proc. 15th IEEE Symp. Comput. Arithmetic*, 2001, pp. 277–281.

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

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

[9] Y. Choi and E. E. Swartzlander, Jr., "Design of a hybrid prefix adder for non-uniform input arrival times," in *Proc. SPIE Adv. Signal Process. Algorithms, Arch., Implementations XII*, 2002, pp. 456–465.

[10] H. Ling, "High-speed binary adder," *IBM J. R&D*, vol. 25, pp. 156–166, 1981.

[11] T. Lynch and E. E. Swartzlander, Jr., "A spanning tree carry lookahead adder," *IEEE Trans. Comput.*, vol. 41, no. 8, pp. 931–939, Aug. 1992.

[12] J. Grad and J. E. Stine, "A hybrid Ling carry-select adder," in *Proc. 38th Asilomar Conf. Signals Syst. Comput.*, 2004, pp. 1363–1367.

A Novel Hybrid Parallel-Prefix Adder Architecture With Efficient Timing-Area Characteristic

Sabyasachi Das and Sunil P. Khatri

***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-lookahead adder 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 adder hand, the proposed adder also shows marginally faster performance than the fast Kogge–Stone adder with significant area savings.**

***Index Terms*—Arithmetic and logic structures, integrated circuits, logic design.**

I. Introduction

The complexity and the performance requirement of the datapath operations implemented in systems-on-chips (SoCs) has increased considerably over the years. Since binary adders are one of the most basic and widely used arithmetic datapath operations in modern integrated circuits, they tend to play a critical role in determining the performance of the design. Hence, developing an 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

Manuscript received March 18, 2007; revised June 11, 2007.

S. Das is with Asyst Technologies, Freemont, CA 94538 USA (e-mail: sabya@asyst.com).

S. P. Khatri is with the Department of Electrical and Computer Engineering, Texas A&M University, College Station, TX 77843 USA (e-mail: sunilkhatri@tamu.edu).

Digital Object Identifier 10.1109/TVLSI.2007.915507

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 an 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.

In this paper, 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.

We have organized the rest of this paper as follows. In Section II, we present some background information about the parallel-prefix architecture. In Section III, we discuss our proposed approach in detail. Section IV presents the experimental results. Conclusions are drawn in Section V.

II. Preliminaries

In this section, we briefly explain the concept of the carry lookahead adder and the parallel-prefix network, using the example of a two-operand (a and b) addition block.

In every bit (i) of the two-operand adder block, the two input signals (a_i and b_i) are added to the corresponding carry-in signal (carry_i) to produce the sum output (sum_i).

The equation to produce the sum output is:

$$\mathrm{sum}_i = a_i \oplus b_i \oplus \mathrm{carry}_i. \tag{1}$$

Computation of the carry-in signals at every bit is the most critical and time-consuming operation. In the carry-lookahead scheme of adders, the focus is to design a circuit which can efficiently compute the $(n-1)$ carry-in signals (c_1 to c_n) based on the $2n$ input bits ($a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$). For any given bit-position, the *generate* (g_i) and *propagate* (p_i) signals are defined as follows:

$$g_i = a_i \wedge b_i \tag{2}$$

$$p_i = a_i \oplus b_i. \tag{3}$$

The key idea behind the parallel prefix computation is as follows.

Let $B_{i,j+1}$ and $B_{j,k}$ be two adjacent blocks in an adder module. These two blocks consist of $(i-j)$ and $(j-k+1)$ bits, respectively, and $B_{i,j+1}$ consists of more significant bits than $B_{j,k}$. The concept of *propagate* and *generate* of individual bits is applicable to blocks of adjacent bits also. The *propagate* and *generate* value-pairs of these two blocks are referred to as $(g_{i,j+1}, p_{i,j+1})$ and $(g_{j,k}, p_{j,k})$, respectively. In this paper, we denote these pairs of *generate* and *propagate* values as $GP_{i,j+1}$ and $GP_{j,k}$. If the block consists of only one bit, then to represent the value pair of (g_i, p_i), we use the notation of GP_i (instead of $GP_{i,i}$). Now, if we combine these two adjacent blocks to form a single continuous block having $(i-k+1)$ bits, the equations for computing the *generate* and *propagate* values of the combined block is as follows:

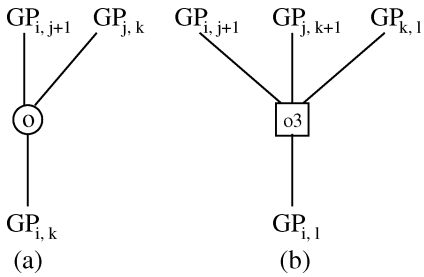

Fig. 1. Block-diagrams of "*o*" (carry) and "*o3*" (triple-carry) operators.

$$g_{i,k} = g_{i,j+1} \vee (p_{i,j+1} \wedge g_{j,k}) \tag{4}$$

$$p_{i,k} = p_{i,j+1} \wedge p_{j,k}. \tag{5}$$

The final output of a parallel prefix computation tree is the set of all the $(g_{i,0}, p_{i,0})$ value pairs (for $i = 0, 1, \ldots, (n-1)$). For a two-operand addition block, the value of the signal $g_{i,0}$ at every bit is equal to the value of the signal carry_{i+1} (for $i = 0, 1, \ldots, (n-1)$).

The Brent and Kung adder and [2] the Kogge and Stone adder [3] use the "*o*" operator, which performs the computation described in (4) and (5) (for any given *generate* and *propagate* value pairs $(g_{i,j+1}, p_{i,j+1})$ and $(g_{j,k}, p_{j,k})$). The block diagram of the "*o*" operator is shown in Fig. 1(a).

III. Our Approach

Throughout the rest of this paper, we assume two operands (a and b) of the adder are n-bit wide, and the output (sum) of the adder is $(n+1)$-bit wide. In our approach, we compute the *generate* and *propagate* signals for each of the individual bits by using the logic presented in (2) and (3). After computing all the GP_i values (g_i, p_i) for each of the individual bits $(i = 0, 1, 2, \ldots, (n-1))$, these get transmitted to the proposed parallel-prefix carry computation tree, described in the following.

We define the notion of computing the *generate* and *propagate* signals for a merged block comprising three adjacent blocks. Let $B_{i,j+1}$, $B_{j,k+1}$, and $B_{k,l}$ be three adjacent blocks in an adder module. These blocks consist of $(i-j)$, $(j-k)$, and $(k-l+1)$ bits, respectively. In addition, suppose that $B_{i,j+1}$ consists of more significant bits than $B_{j,k+1}$, and $B_{j,k+1}$ consists of more significant bits than $B_{k,l}$. The *propagate* and *generate* value pairs of these three blocks are $(g_{i,j+1}, p_{i,j+1})$, $(g_{j,k+1}, p_{j,k+1})$, and $(g_{k,l}, p_{k,l})$, respectively. Now, if we combine these three adjacent blocks to form a single continuous block having $(i-l+1)$ bits, then the combined block $(B_{i,l})$ propagates the carry only if each of the three blocks ($B_{i,j+1}$, $B_{j,k+1}$, and $B_{k,l}$) propagates the carry. On the other hand, the combined block $(B_{i,l})$ generates carry in the following three situations.

- If the block $B_{i,j+1}$ generates a carry. In other words, if $(g_{i,j+1} = 1)$.

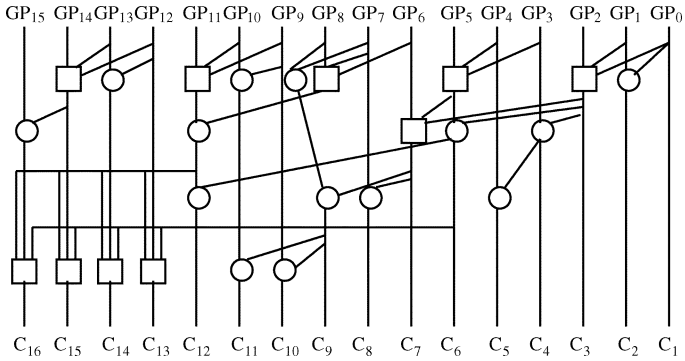

Fig. 2. Our proposed parallel prefix network (for input width of 16 bits).

- If the block $B_{j,k+1}$ generates a carry and the block $B_{i,j+1}$ propagates that carry. In other words, if $(g_{j,k+1} = 1)$ and $(p_{i,j+1} = 1)$.
- If the block $B_{k,l}$ generates a carry and the blocks $B_{i,j+1}$, $B_{j,k+1}$ propagates that carry. In other words, if $(g_{k,l} = 1)$, $(p_{i,j+1} = 1)$, and $(p_{j,k+1} = 1)$.

The equations for computing the *generate* and *propagate* values of the combined block are as follows:

$$g_{i,l} = g_{i,j+1} \vee (p_{i,j+1} \wedge g_{j,k+1}) \vee (p_{i,j+1} \wedge p_{j,k+1} \wedge g_{k,l}) \quad (6)$$

$$p_{i,l} = p_{i,j+1} \wedge p_{j,k+1} \wedge p_{k,l}. \quad (7)$$

We denote the previous expressions [in (6) and (7)] as the "*o3*" operator (or the *triple-carry operator*), which takes three pairs of *generate* and *propagate* values as inputs and produces the *generate* and *propagate* values of the combined large block. The block diagram of the "*o3*" operator (*triple-carry-operator*) is shown in Fig. 1(b).

We use this in conjunction with the classic approach of computing *generate* and *propagate* signals for a merged block by combining two adjacent blocks [as explained in the (4) and (5)]. This is called the "*o*" operator. The block diagram of the "*o*" operator is shown in Fig. 1(a).

The key idea in our approach is to find opportunities to use the triple-carry-operator ("*o3*" operator). By analyzing several technology libraries provided by commercial vendors, we have found that the worst delay through a triple-carry operator is between 110% and 130% of the traditional carry operator. Since the "*o3*" operator produces the *generate* and *propagate* value pair by combining three blocks as opposed to two blocks in the traditional carry operator, the additional 10% to 30% of delay is well justified. Since the "*o3*" operator processes one additional block compared to the *"o" operator*, it reduces the depth of the parallel-prefix network. The area of the "*o3*" operator is 50% to 80% more than the area of the *"o" operator*, hence, we only use the "*o3*" operator in the timing critical portion of the parallel prefix tree. This delay characteristic makes triple-carry operator an efficient choice in the parallel prefix network.

The block diagram of a 16-bit wide proposed parallel prefix graph (of an adder block) is shown in Fig. 2. In addition, Fig. 3 represents the block-diagram of a 24-bit wide proposed parallel prefix computation network. Since the carry_{i+1} is equal to $g_{i,0}$, we label all the outputs in Figs. 2 and 3 as C_i (for each value of i). Due to the lack of space in the diagrams, C_i is used as an abbreviation instead of carry_i. Both the diagrams are drawn in a *levelized* fashion. Let us assume that the inputs to the parallel prefix tree (GP_i) are at level (or *depth*) 0, shown at the top of Figs. 2 and 3. As we proceed downwards in Figs. 2 and 3, the level (or depth) increases by one. In these designs, at level 1, we initially use a large number of "*o3*" operators. We instantiate triple-carry operators to combine every GP_{3p}, GP_{3p+1}, and GP_{3p+2} (until each GP_i participates in an operator). Then, in the second level,

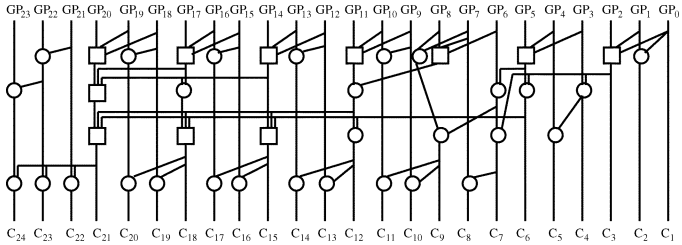

Fig. 3. Our proposed parallel prefix network (for input width of 24 bits).

we mostly perform the traditional carry operation with the "*o*" operator (with some "*o3*" operators as well). In levels lower than 1, we perform timing-driven optimization and use a combination of the two types of operators. To avoid problems due to high fanout nets, we restrict the maximum fanout of any net to 5. To maintain this strict limit on fanouts, we use the triple-carry operators quite aggressively in the bits near the most significant bit. We note that, in most of the parallel prefix computation tree designs, the critical paths primarily go through the outputs, which are placed near the most significant bit $(n - 1)$. Hence, we try to instantiate triple-carry operators in the paths which go through the critical pins. This reduces the depth along those paths (at the expense of additional hardware) and improves the performance of the parallel prefix block. On the other hand, we also note that bits near the least significant bit typically have positive slack. To exploit this fact and to perform area reduction, we use a ripple type structure in that part of the design (without impacting the overall performance of the block). As a result, we claim that our design is timing driven in the timing critical paths and area driven in the nontiming critical (and area critical) paths.

Note that we do not extend the concept of "*o3*" operators to combine four adjacent blocks to form a single block ("*o4*" block). This is because an "*o4*" operator is a combination of two levels of "*o*" operators (total of three "*o*" operators) arranged in a tree-like fashion. Hence, unlike the usage of "*o3*" operator, usage of "*o4*" operator is not an architectural optimization. Depending on the availability of the cells in the technology library, a high-quality technology mapping algorithm in commercial logic synthesis tools should be able to efficiently use the cells required for the "*o4*" operator.

IV. 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 bit-widths (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 (L_1 and L_2) for 0.13 μ;
  - — two commercial libraries (L_3 and L_4) 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.

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

			Worst-case Delay (ps)					Area (μ^2)				
Design	Technology Library	Timing Constraint	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%

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 $\mathrm{Arr(late)}$. Let us denote $\mathrm{Arr}(a_i)$ as the arrival time of the signal a_i. 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 $\mathrm{Arr(late)}$ timing constraint (n is the width of the adder inputs):

$$\mathrm{Arr}(a_i) = ik\delta;\ \ 0 \le i \le \lceil 3n/5 \rceil$$
$$\mathrm{Arr}(a_i) = \lceil 3n/5 \rceil k\delta - (i - \lceil 3n/5 \rceil)\, k\delta;\ \ \lceil (3n/5) \rceil < i < n$$
$$\mathrm{Arr}(b_i) = ik\delta;\ \ 0 \le i \le \lceil 3n/5 \rceil$$
$$\mathrm{Arr}(b_i) = \lceil 3n/5 \rceil k\delta - (i - \lceil 3n/5 \rceil)\, k\delta;\ \ \lceil (3n/5) \rceil < i < n.$$

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

$$\mathrm{Arr}(a_i) = k; \quad 0 \le i < n$$
$$\mathrm{Arr}(b_i) = k; \quad 0 \le i < n.$$

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, timing-driven 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.

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 indi-

TABLE II
LEAKAGE AND DYNAMIC POWER COMPARISON OF ADDER BLOCKS GENERATED BY BK, KS, AND OUR APPROACH

			Leakage Power (μW)					Dynamic Power (μW)				
Design	Technology Library	Timing Constraint	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)	3	7	4	-33.33%	42.85%	21	31	24	-14.29%	22.58%
Adder-24	L_1 (0.13μ)	Arr(late)	5	12	7	-40.00%	41.67%	30	46	32	-6.67%	30.43%
Adder-32	L_1 (0.13μ)	Arr(late)	8	16	9	-12.50%	43.75%	43	63	48	-11.63%	23.81%
Adder-48	L_1 (0.13μ)	Arr(late)	10	23	11	-10.00%	52.17%	52	75	59	-13.46%	21.33%
Adder-64	L_1 (0.13μ)	Arr(late)	13	29	15	-15.38%	48.27%	68	89	73	-7.35%	17.98%
Adder-16	L_3 (0.09μ)	Arr(late)	14	28	16	-14.28%	42.85%	69	93	74	-7.24%	20.43%
Adder-24	L_3 (0.09μ)	Arr(late)	23	41	25	-8.69%	39.02%	88	117	96	-9.09%	17.94%
Adder-32	L_3 (0.09μ)	Arr(late)	26	49	31	-19.23%	36.73%	116	160	124	-6.89%	22.50%
Adder-48	L_3 (0.09μ)	Arr(late)	49	73	57	-14.03%	21.92%	149	196	161	-8.06%	17.86%
Adder-64	L_3 (0.09μ)	Arr(late)	65	85	73	-12.31%	14.12%	192	271	219	-14.06%	19.18%
Adder-16	L_2 (0.13μ)	Arr(same)	2	4	2	-0.00%	50.00%	26	42	29	-11.53%	30.95%
Adder-24	L_2 (0.13μ)	Arr(same)	4	9	5	-25.00%	44.44%	37	58	41	-10.81%	29.31%
Adder-32	L_2 (0.13μ)	Arr(same)	5	12	7	-40.00%	41.67%	49	73	57	-16.33%	21.92%
Adder-48	L_2 (0.13μ)	Arr(same)	8	17	11	-37.50%	35.29%	71	99	80	-12.68%	19.19%
Adder-64	L_2 (0.13μ)	Arr(same)	11	19	12	-9.09%	36.84%	98	129	106	-8.16%	17.83%
Adder-16	L_4 (0.09μ)	Arr(same)	5	11	6	-20.00%	45.45%	32	47	34	-6.25%	27.65%
Adder-24	L_4 (0.09μ)	Arr(same)	9	15	11	-22.22%	26.67%	46	66	49	-6.53%	25.76%
Adder-32	L_4 (0.09μ)	Arr(same)	11	24	15	-36.36%	37.50%	58	89	63	-8.62%	29.21%
Adder-48	L_4 (0.09μ)	Arr(same)	17	31	22	-29.41%	29.03%	93	137	102	-9.67%	25.55%
Adder-64	L_4 (0.09μ)	Arr(same)	24	37	27	-12.50%	27.02%	129	164	138	-6.98%	15.85%
Adder-16	L_4 (0.09μ)	Arr(late)	13	21	14	-7.69%	50.00%	43	64	49	-13.95%	23.43%
Adder-24	L_4 (0.09μ)	Arr(late)	17	26	19	-11.76%	26.92%	57	74	62	-8.77%	16.21%
Adder-32	L_4 (0.09μ)	Arr(late)	26	37	29	-11.54%	21.62%	74	103	82	-10.81%	20.38%
Adder-48	L_4 (0.09μ)	Arr(late)	31	53	36	-16.13%	32.07%	116	152	123	-6.03%	19.07%
Adder-64	L_4 (0.09μ)	Arr(late)	59	82	70	-18.65%	14.63%	159	203	174	-9.43%	14.29%
Average						-19.10%	36.10%				-9.81%	22.03%

vidual 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 $\mathrm{Arr(late)}$. 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.

In Table II, we present the post-synthesis leakage and dynamic power results for the adder block for each of the three architectures (as reported by the synthesis tool). By analyzing the result, we note that the power consumption of our adder is more than that of the BK adders, but significantly less than that of the KS adder.

State-of-the-art designs need to be designed while considering different cost metrics (timing, area, power, etc.). The efficient timing-area characteristics of our proposed adder design (over the well-known BK and KS adders) is consistent across multiple sizes of adders, timing constraints and technology libraries. This underscores the utility and scalability of our design. Since addition is a frequently used part of many critical operations in an IC, we believe that many real-life designs can significantly benefit from our proposed architecture.

To achieve the proposed architecture's benefit, the designer does not require to perform any extra task. Typically a state of the art datapath synthesis tool has multiple architectures available for adder and it selects the appropriate one depending on the timing constraint, library, the design, etc. As a result, when this architecture is the *best* in the given situation for the given adder block, the synthesis tool will automatically select this architecture without the designer doing anything special.

V. CONCLUSION

In this paper, 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 μ, 0.09 μ, etc.). The experimental results indicate that our proposed adder is significantly faster than the popular BK adder with some area overhead. On the adder hand, the proposed adder also shows marginally faster performance than the fast KS adder with significant area 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, "A regular 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 zero-deficiency 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.

[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.

Low Power Design of Precomputation-Based Content-Addressable Memory

Shanq-Jang Ruan, Chi-Yu Wu, and Jui-Yuan Hsieh

***Abstract*—Content-addressable memory (CAM) is frequently used in applications, such as lookup tables, databases, associative computing, and networking, that require high-speed searches due to its ability to improve application performance by using parallel comparison to reduce search time. Although the use of parallel comparison results in reduced search time, it also significantly increases power consumption. In this paper, we propose a Block-XOR approach to improve the efficiency of low power precomputation-based CAM (PB-CAM). Through mathematical analysis, we found that our approach can effectively reduce the number of comparison operations by 50% on average as compared with the ones-count approach for 32-bit-long inputs. In our experiment, we used Synopsys Nanosim to estimate the power consumption in TSMC 0.35-μm CMOS technology. Compared with the ones-count PB-CAM system, the experimental results show that our proposed approach can achieve on average 30% in power reduction and 32% in power performance reduction. The major contribution of this paper is that it presents theoretical and practical proofs to verify that our proposed Block-XOR PB-CAM system can achieve greater power reduction without the need for a special CAM cell design. This implies that our approach is more flexible and adaptive for general designs.**

***Index Terms*—Content-addressable memory (CAM), low-power, precomputation.**

I. Introduction

A content-addressable memory (CAM) is a critical device for applications involving asynchronous transfer mode (ATM), communication networks, LAN bridges/switches, databases, lookup tables, and tag directories, due to its high-speed data search capability. A CAM is a functional memory with a large amount of stored data that simultaneously compares the input search data with the stored data. Once matching data are found, their addresses are returned as output as shown in Fig. 1. The vast number of comparison operations required by CAMs consumes a large amount of power.

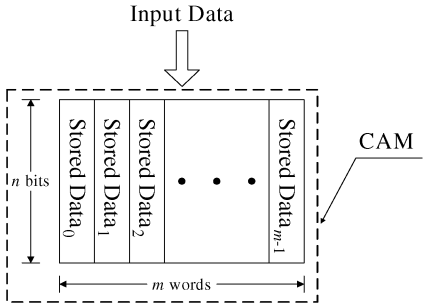

Fig. 1. Conventional CAM architecture.

In the past decade, much research on energy reduction has focused on the circuit and technology domains ([1] provides a comprehensive survey on CAM designs from circuit to architectural levels). Several works on reducing CAM power consumption have focused on reducing match-line power [2]–[4]. Although there has been progress in this area in recent years, the power consumption of CAMs is still high compared with RAMs of similar size. At the same time, research in associative cache system design for power efficiency at the architectural level continues to increase. The filter cache [5], [6] and location cache techniques [7] can effectively reduce the power dissipation by adding a very small cache. However, the use of these caches requires major modifications to the memory structure and hierarchy to fit the design. Pagiamtzis *et al.* proposed a cache-CAM (C-CAM) that reduces power consumption relative to the cache hit rate [8]. Lin *et al.* presented a ones-count precomputation-based CAM (PB-CAM) that achieves low-power, low-cost, low-voltage, and high-reliability features [9]. Although Cheng [10] further improved the efficiency of PB-CAMs, the approach proposed requires considerable modification to the memory architecture to achieve high performance. Therefore, it is beyond the scope of the general CAM design. Moreover, the disadvantage of the ones count PB-CAM system [9] is that it adopts a special memory cell design for reducing power consumption, which is only applicable to the ones-count parameter extractor.

In this paper, we present a Block-XOR approach for reducing comparison operations in the second part for the PB-CAM. Our approach employs a brand new parameter extractor, which can better reduce the comparison operations required than the ones-count approach [9]. Our approach reduces power consumption by reducing comparison operations.

The remainder of this paper is organized as follows. In Section II, we briefly describe the PB-CAM architecture. Our new architecture is described in Section III, where the design of our Block-XOR parameter extractor is provided and we exploit mathematical analysis to prove the effectiveness of our proposed architecture. In Section IV, the experimental results are provided to further verify our mathematical analysis. In addition, we also give a comprehensive comparison between [9] and our approach. Finally, we give a conclusion in Section V.

II. Previous Work and Observation

To understand our approach more clearly, we need to briefly describe the architecture of the PB-CAM proposed in [9].

Manuscript received March 27, 2006; revised March 12, 2007, April 9, 2007, and June 14, 2007.

The authors are with the Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan, R.O.C. (e-mail: sjruan@mail.ntust.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2007.915514