An Algorithmic Approach for Generic Parallel Adders

Jianhua Liu, Shuo Zhou, Haikun Zhu, Chung-Kuan Cheng
Department of Computer Science and Engineering
University of California, San Diego

Abstract— **Binary addition is the most fundamental and frequently used operation. A well-designed adder should be fast and satisfy the application requirements. We propose an algorithmic approach to generate an irregular parallel-prefix adder, which has minimal delay for a given profile of input signals. It can cover different topologies such as ripple-carry, carry-skip and carry-select adders. Compared with Kogge-Stone and Brent-Kung adders, the results of the proposed approach have the smallest output delay.**

I. Introduction

Datapath module is essential for high quality ASIC design, and may dominate the whole system performance. Arithmetic components, such as adders, multipliers and shifters, are considered as basic cells to a construct datapath. Design of arithmetic components should be high performance and satisfy the application requirements. Binary addition is the most fundamental and frequently used operation in computing systems. To speed up binary addition, many different architectures have been proposed over the years.

The ripple-carry adder has the minimal area, but is quite slow. The carry-skip adder [1] can speed up binary addition with a small hardware overhead. The carry-select adder [2] accelerates binary addition further, but suffers from large hardware penalty. The carry-lookahead adder [3] [4] comes with prefix computation. It has $O(\log n)$ time and $O(n \log n)$ area. Brent-Kung parallel prefix adder and Kogge-Stone parallel prefix adder are two classical regular prefix computation structures, which reach lower bound of area and lower bound of time [5] respectively.

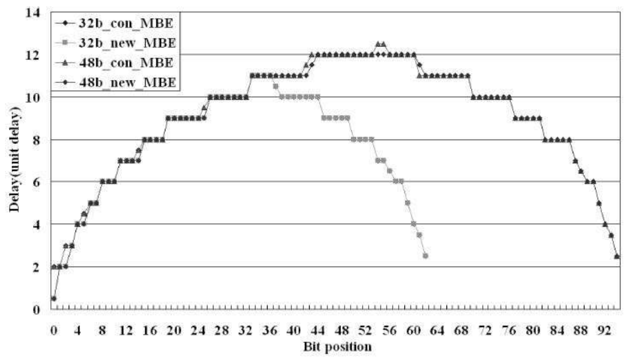


Fig. 1. Output delay profiles of PPRT

If all input signals arrive simultaneously, Kogge-Stone adder gives the fastest prefix structure. However, the arrival times of inputs may not be uniform for a specific application. For example, Fig.1 shows the output delay profile of partial product reduction tree [7], which is the input delay profile of the final adder. Under such a non-uniform input delay profile, a heuristic algorithm [8] can generate a faster prefix structure than Kogge-Stone adder. But this algorithm can not guarantee to find the delay-optimal prefix structure.

In this paper, we propose an algorithmic approach to generate an irregular parallel-prefix adder, which has the minimal delay for a given profile of input signals. The approach can cover different topologies of ripple-carry adder, carry-skip adder and carry-select adder. The time complexity of the proposed algorithm is $O(n^3)$. To minimize the area cost a heuristic backward reduction procedure is added. The experimental results show that the proposed approach outperforms both Brent-Kung and Kogge-Stone parallel prefix schemes, for particular applications.

The rest of the paper is organized as follows. The problem and timing model are defined in Section II. The algorithmic approach will be presented in Section III. In Section IV, we will show how this approach covers ripple-carry, carry-skip and carry-select adders. The backward reduction procedure will be introduced in Section V. Section VI analyzes the experimental results, before conclusions and future work in section VII.

II. Preliminaries

A. Prefix Addition[6]

We define the binary addition problem as follows: given an n-bit augend A, an n-bit addend B, and a 1-bit carry-in c_0, generate the n-bit sum S and the 1-bit carry-out c_n. Suppose $A = a_n \dots a_2 a_1$ and $B = b_n \dots b_2 b_1$, we define s_i and c_i as follows:

$$s_i = a_i \oplus b_i \oplus c_{i-1} \tag{1}$$
$$c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1} \tag{2}$$

According to prefix computation, we define g_i (*generate*) and p_i(*propagate*) as:

$$g_i = \begin{cases} c_0, & \text{if } i = 0 \\ a_i b_i, & \text{otherwise} \end{cases} \tag{3a}$$
$$p_i = \begin{cases} 0, & \text{if } i = 0 \\ a_i \oplus b_i, & \text{otherwise} \end{cases} \tag{3b}$$

(Pre-processing)

If g_i equals 1, a carry is generated at bit i; if p_i equals 1, a carry is propagated through bit i. The concept of generate and propagate can be extended to multiple bits. We define $G[i:k]$ and $P[i:k]$ $(i \geq k)$ as:

$$G_{[i:k]} = \begin{cases} g_i, & \text{if } i = k \\ G_{[i:k]} + P_{[i:j]} G_{[j-1:k]}, & \text{otherwise} \end{cases} \tag{4a}$$

$$P_{[i:k]} = \begin{cases} p_i, & \text{if } i = k \\ P_{[i:j]} P_{[j-1:k]}, & \text{otherwise} \end{cases} \tag{4b}$$

(Prefix computation)

To simplify the representation of G and P, an operator $\bullet$ is defined in (G, P) computation:

$$(G,P)_{[i:k]} = (G,P)_{[i:j]} \bullet (G,P)_{[j-1:k]} \tag{5}$$

s_i and c_i can be calculated from G and P:

$$c_i = G_{[i:0]} \tag{6a}$$

$$s_i = p_i \oplus c_{i-1} \tag{6b}$$

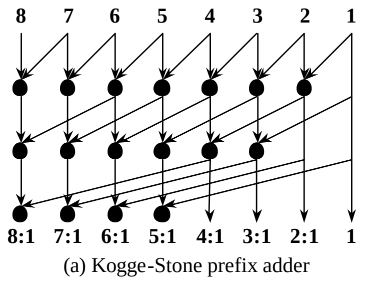
(Post-processing)

(G, P) computation has two important properties:

- **Property 1:** (G, P)**s can overlap in operation.** That is

$$\begin{aligned}(G,P)_{[i:k]} &= (G,P)_{[i:j]} \bullet (G,P)_{[j-1:k]} \\ &= (G,P)_{[i:j]} \bullet (G,P)_{[l:k]}, i > l \geq j-1 \end{aligned} \tag{7}$$

- **Property 2:** G **and** P **can have separated prefix computation structures.** This property is known from the definitions of G and P.



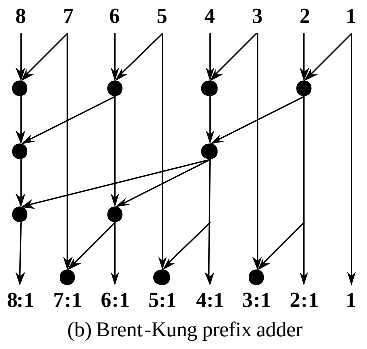


Fig. 2. Kogge-Stone and Brent-Kung prefix adders

Since pre-processing and post-processing have constant delay, prefix computation becomes the core of prefix adders and dominates the performance. Prefix structures can be represented in directed acyclic graphs. Fig.2 shows the prefix structures of an 8-bit Kogge-Stone adder and an 8-bit Brent-Kung adder.

B. Timing model

Fig.3 shows three kinds of basic components: gp generator, (G, P) adder, and sum generator in prefix adders. They are denoted by a diamond, a solid circle, and a rectangle respectively.

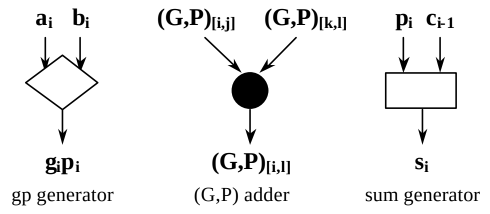


Fig. 3. Basic components

gp generator

$$T_{g_i p_i} = max(T_{a_i}, T_{b_i}) + 1 \tag{8}$$

(G, P) adder

$$T_{(G,P)_{[i,l]}} = max(T_{(G,P)_{[i,j]}}, T_{(G,P)_{[k,l]}}) + 2 \tag{9}$$

sum generator

$$T_{s_i} = max(T_{p_i}, T_{c_i} - 1) + 1 \tag{10}$$

To make representations easier, the simplest timing model is used in Sections III to V, which is similar to the timing model in [8]. A gp generator as well as sum generator takes 1 unit delay from all inputs to outputs. A (G, P) adder takes 2 unit delay from all inputs to outputs. A more accurate timing model is used in experimental results (Section VI).

III. An Algorithmic Approach

The pseudo-code of the proposed dynamic programming approach is given in Algorithm 1.

Algorithm 1 Prefix Adder Construction

```
construct_prefix_adder (data_width)
1:  $n$ =data_width;
2:  {Forward Process}
3:  for $i = 2$ to $n$ do
4:    for $j = 1$ to $n - i + 1$ do
5:      $T = \infty$;
6:      for $k = j$ to $j + i - 2$ do
7:        $T' = \max(T_{(G,P)_{[j+i-1,k+1]}}, T_{(G,P)_{[k,j]}}) + 2$;
8:        if $T' \leq T$ then
9:          $T = T'$;
10:         Construct $(G,P)_{[j+i-1,j]}$
            from $(G,P)_{[j+i-1,k+1]}$ and $(G,P)_{[k,j]}$;
11:       end if
12:     end for
13:   end for
14: end for
15: return;
```

The forward process constructs $(G, P)s$ level by level. Initially all $g_i p_i$ are in level 1. All $(G, P)s$ with length 2 are constructed first, which are denoted as $(G, P)_{[k+1,k]}$. These $(G, P)s$ form level 2, and $(G, P)_{[k+1,k]}$ can only be generated from the pair $\{g_{k+1}p_{k+1}, g_k p_k\}$. The succeeding levels are constructed step by step. When constructing level l, all $(G, P)s$ in the previous levels are known. All possible combinations of (G, P) pairs are tried without overlap, and the pair with minimal delay is selected to construct the current (G, P).

Fig.4 illustrates the above process. Consider $(G, P)_{[3,1]}$ as an example. There are two ways to construct $(G, P)_{[3,1]}$ without (G, P) overlap, either from $\{(G, P)_{[3,2]}, g_1 p_1\}$ or $\{g_3 p_3, (G, P)_{[2,1]}\}$. These pairs have delays of 8 and 7 units respectively. $\{g_3 p_3, (G, P)_{[2,1]}\}$ is selected to construct $(G, P)_{[3,1]}$. Continue selecting pairs until $(G, P)_{[n,0]}$ is finally constructed.

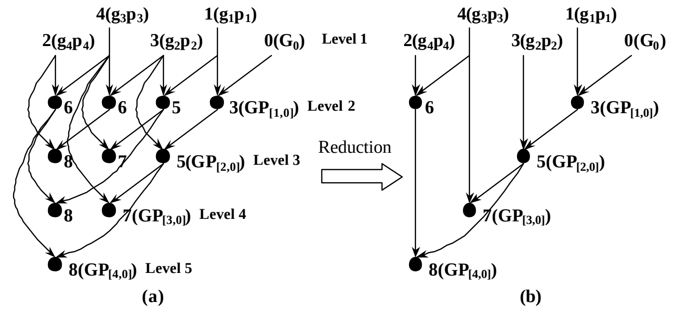


Fig. 4. Dynamic programming algorithm and reduction

Once we have all $(G, P)_{[i,0]}$, which are needed for post-processing stages, a backward process is added to minimize the area. A simple method to reduce the number of (G, P) adders is to remove the unnecessary $(G, P)s$ from the prefix structure. We do so by checking each (G, P) backward. If one (G, P) does not output to any other component, the component and its input connections are removed. In this example, $(G, P)_{[4,1]}$, $(G, P)_{[4,2]}$ and $(G, P)_{[3,1]}$ are removed first. As a result, $(G, P)_{[3,2]}$ and $(G, P)_{[2,1]}$ are now unnecessary. Following is the pseudo-code for prefix adder reduction.

Algorithm 2 Prefix Adder Reduction

```
reduce_prefix_adder (data_width)
 1: $n$ =data_width;
 2: {Backward Process}
 3: for $i = n$ to 2 do
 4:     for $j = 1$ to $n - i + 1$ do
 5:         if $j == 1$ then
 6:             label $(G, P)_{[j+i-1,j]}$ with "in_use";
 7:         end if
 8:         if $(G, P)_{[j+i-1,j]}$ is labelled with "in_use" then
 9:             label two parents of $(G, P)_{[j+i-1,j]}$ with "in_use";
10:         end if
11:     end for
12: end for
13: Remove all $(G, P)s$ without label "in_use";
14: return;
```

Obviously, the time complexity and space complexity of this algorithm are $O(n^3)$ and $O(n^2)$ respectively. From the observation of (G, P), we have two conclusions

- **Lemma 1: (G, P) overlap is not necessary to keep the minimal delay.** Overlap can be eliminated by using a (G, P) with shorter length, since the delay of (G, P) will be decreased with length reduction. This property makes the time complexity of the algorithm decrease from $O(n^4)$ to $O(n^3)$.
- **Theorem 2: When G and P are not allowed to separate, Algorithm 1 has minimal delay.** This property is based on the following two observations: 1) To achieve minimal delay at one (G, P), all $(G, P)s$ in its critical path should have minimal delay; 2) it's not harmful to make all $(G, P)s$ not in the critical path as early as possible. As a direct inference, a minimal delay can be obtained from minimal delays in the previous levels. Since level 1 and level 2 have minimal delays by nature, Theorem 2 holds.

In our experiments, G and P separation cannot lead to better timing results. So we did not include G and P separation in the dynamic programming algorithm.

IV. Transformation to generic adders

The proposed algorithm can cover topologies of ripple-carry, carry-skip and carry-select adders. Based on these topologically identical structures, the generated structures can be partially converted to different adders. According to the structure comparisons, the resulting structure is not only optimal in parallel-prefix adders, but also optimal in combinations of different adders.

A. Transformation to ripple adder

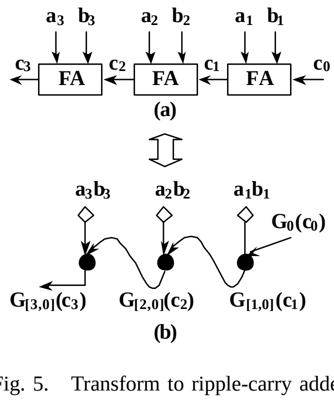


Fig. 5. Transform to ripple-carry adder

Fig.5 shows a ripple-carry structure and the corresponding result in the proposed approach. In ripple-carry adders, carry propagates bit by bit. In the proposed approach, $G_{[i:0]}$ can also propagate column by column, and $G_{[i:0]}$ is essentially the carry c_i. In this case, $P_{[i:0]}$ is not necessary. The $(G, P)_{[i:0]}$ serial-propagated chain can be converted to a ripple-carry adder. These two structures have almost the same timing and area.

B. Transformation to carry-skip adder

Fig.6 shows a carry-skip structure and the corresponding result in the proposed approach. Two dash-line boxes denote two corresponding blocks. They have the same inputs and their outputs are connected to the same logic. Note that the carry-skip logic in carry-skip adder is exactly the same as the G part in (G, P) adder.

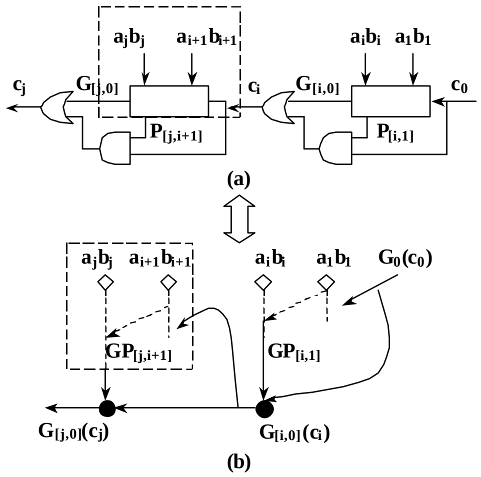


Fig. 6. Transform to carry-skip adder

The only difference is the meaning of one signal between the block and the carry-propagate logic. In carry-skip adder, one input to carry-skip logic is c_j, or say $G_{[j,0]}$. The corresponding input is $G_{[j,i+1]}$. However, when both $G_{[i,0]}$ and $P_{[j,i+1]}$ equal 1, $G_{[j,0]}$ in the carry-skip adder will be skipped. So the signal $G_{[j,0]}$ works as $G_{[j,i+1]}$ in the proposed approach. Two structures shown in Fig.6 are identical.

C. Transformation to carry-select adder

Fig.7 shows a carry-select structure and the corresponding result in the proposed approach. In the carry-select adder, all possible results are pre-calculated to wait for carry-in. When carry-in is ready, it will feed all columns simultaneously, without propagation between columns. There exists the same topology in the resulting structure.

Two dash-line boxes denote two corresponding carry-select blocks. The principal difference is the moment to process the carry-in c_0. In carry-select adder, s_i^0 and s_i^1 are generated before the carry-in processing. So the delay from c_0 to s_i is only the delay of a MUX. In the resulting structure, carry-in is processed earlier. The delay from c_0 to s_i increases to the delay of a (G, P) adder plus the delay of a sum generator.

To reduce the delay from c_0 to s_i, a carry-select block in the resulting structure can be transformed to the structure shown in Fig.8(b). The original function of the carry-select block is

$$s_i = \bar{c_0}(G_{[i-1,1]} \oplus p_i) + c_0((G_{[i-1,1]} + P_{[i-1,1]}) \oplus p_i) \quad (11)$$

Note that $G_{[i-1,1]}$ and $P_{[i-1,1]}$ are mutually exclusive. So the term $(G_{[i-1,1]} + P_{[i-1,1]})$ can be replaced by $(G_{[i-1,1]} \oplus$

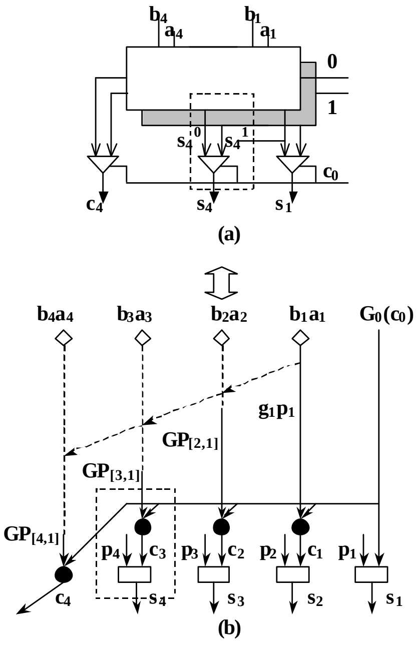


Fig. 7. Transform to carry-select adder

$P_{[i-1,1]})$. Then function(11) changes to

$$s_i = \bar{c_0}(G_{[i-1,1]} \oplus p_i) + c_0((G_{[i-1,1]} \oplus p_i) \oplus P_{[i-1,1]}) \quad (12)$$

which is the function of Fig.8(b). By doing this transformation, the resulting structure will have not only the same topology with a carry-select adder but also the same delay property.

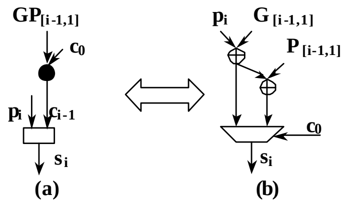


Fig. 8. Transformation of carry-select block

V. Backward Reduction

In the previous algorithm, the forward dynamic programming algorithm generates all $(G, P)s$ with optimal delay. After removing all (G, P) adders without fanout, the prefix structure shrinks to the minimal size with optimal delays at all internal $(G, P)s$. However, keeping optimal delays at all internal $(G, P)s$ is not necessary for the global delay-optimal. Only the $(G, P)s$ in critical path have to keep their optimal delays. Hence there is a potential to reduce the size further by relaxing some internal $(G, P)s$ not in critical path.

To relax a (G, P), its timing requirement should be calculated first. A relaxed timing can not exceed this limit, otherwise the global delay-optimality will be broken. Required times are backward calculated from outputs to inputs. Required times of outputs are given. One (G, P)'s required time will limit the required times of its parents. Fig.12(a) shows an

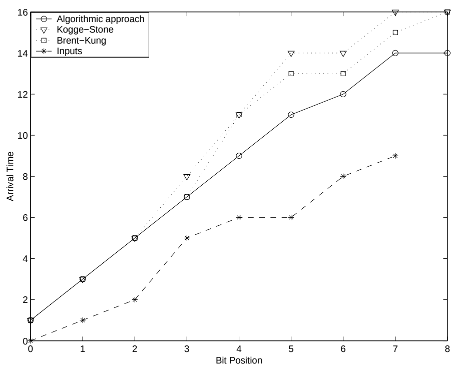


Fig. 9. Result of 8-bit case1

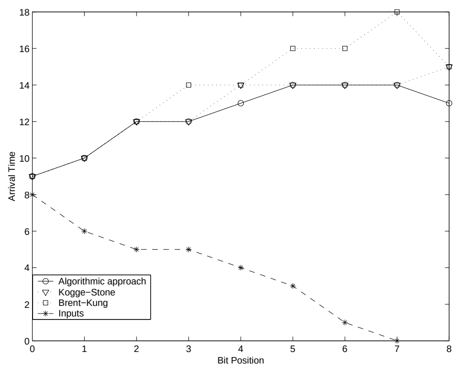


Fig. 10. Result of 8-bit case2

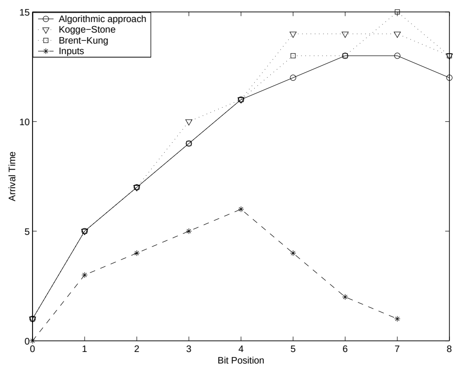


Fig. 11. Result of 8-bit case3

example. Assume all outputs should keep their optimal delays. The required time of each (G, P) in the current netlist is indicated in parenthesis.

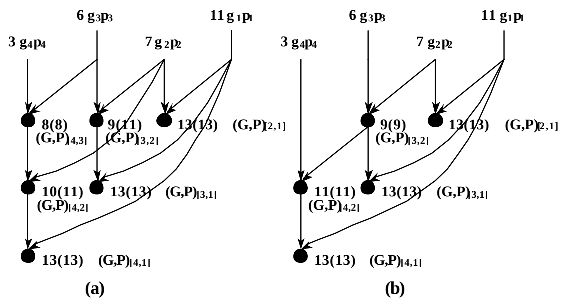


Fig. 12. Backward reduction

As soon as the required time of (G, P) is decided, all possible input combinations can be enumerated, based on all optimal delays calculated in the forward procedure. For example, $(G, P)_{[3,1]}$ in Fig.12(a) can be generated from the pair $\{(G, P)_{[3,2]}, g_1p_1\}$ only. Otherwise the required time can not be satisfied. However $(G, P)_{[4,2]}$ can be generated from the pair $\{(G, P)_{[4,3]}, g_2p_2\}$ or the pair $\{g_4p_4, (G, P)_{[3,2]}\}$. Either way the required time can be satisfied. In these possible combinations, we choose the one whose right parent has the maximal length. So the pair $\{g_4p_4, (G, P)_{[3,2]}\}$ are selected to generate $(G, P)_{[4,2]}$. This strategy makes the structure approach the ripple structure, which has the minimal size.

When a netlist changes, the required time of its parents will be changed. Fortunately, the prefix structure is a tree structure. So that the interconnection and the required time can be decided and calculated backward, level-by-level. Fig.12(b) shows the result of backward reduction from the initial structure Fig.12(a). The number of (G,P) adders reduces by 1. The backward reduction has the same time complexity as the forward dynamic programming procedure. Algorithm 3 gives the pseudo-code of the backward reduction procedure.

VI. EXPERIMENTAL RESULTS

Different input delay profiles and different data widths are used. The shapes of different input delay profiles include monotonously increasing, decreasing, and convex curves. As

Algorithm 3 Backward Reduction

```
backward_reduction (data_width)
1: $n$ =data_width;
2: for $i = n$ to 2 do
3:   for $j = 1$ to $n - i + 1$ do
4:     if $j == 1$ then
5:       label $(G, P)_{[j+i-1,j]}$ with "in_use";
6:     else if $(G, P)_{[j+i-1,j]}$ is labelled with "in_use" then
7:       for $k = j + i - 2$ to $j$ do
8:         $T = \max(T_{(G,P)_{[j+i-1,k+1]}}, T_{(G,P)_{[k,j]}}) + 2$;
9:         if $T \leq RequiredTime\_GP_{[j+i-1,j]}$ then
10:          Connect $GP_{[j+i-1,j]}$ to $GP_{[j+i-1,k+1]}$ and $GP_{[k,j]}$;
11:          $k = j - 1$;
12:        end if
13:      end for
14:    end if
15:    if $(G, P)_{[j+i-1,j]}$ is labelled with "in_use" then
16:      label two parents of $(G, P)_{[j+i-1,j]}$ with "in_use";
17:      Update Required Time of two parents of $(G, P)_{[j+i-1,j]}$;
18:    end if
19:  end for
20: end for
21: Remove all $(G, P)s$ without label "in_use";
22: return;
```

mentioned before, the outputs from partial products reduction tree in multiplier have convex arrival times. Different data widths include 8-bit and 16-bit. Fig.9, Fig.10, and Fig.11 show results of 8-bit circuits compared with Kogge-Stone and Brent-Kung prefix adders on outputs timing, while Fig.13, Fig.14, and Fig.15 indicate results of 16-bit circuits.

In these figures the horizontal axis denotes bit position from low to high, and the vertical axis denotes arrival time. The dashed line with star points shows the input delay profile. The solid line with circle points gives the result of the proposed algorithm. Two dotted lines with triangle and square points represent results of Kogge-Stone and Brent-Kung schemes respectively. Case1, case 2, and case3 use monotonously increasing, decreasing, and convex input delay profiles.

Since the arrival time of inputs is not uniform, the delay between the last input and the last output is used to compare these results. Table.I shows the comparison on the delay and the number of (G, P) adders. The number of (G, P) adders relates to the area of a prefix structure. Two different reduction procedures are used in the algorithmic approach, the backward

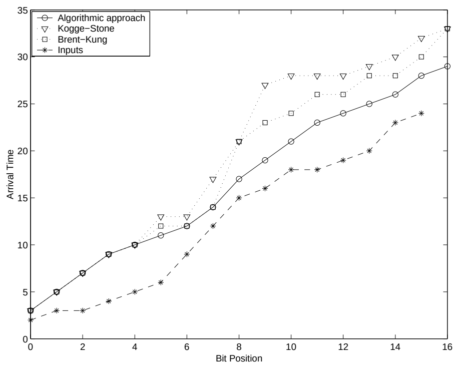


Fig. 13. Result of 16-bit case1

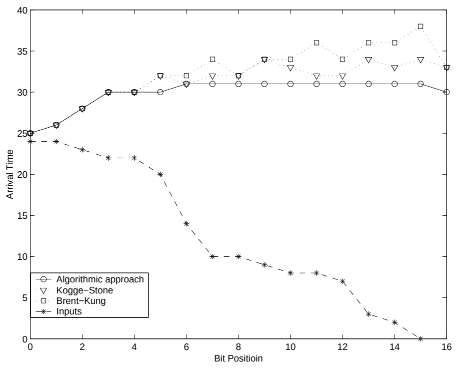


Fig. 14. Result of 16-bit case2

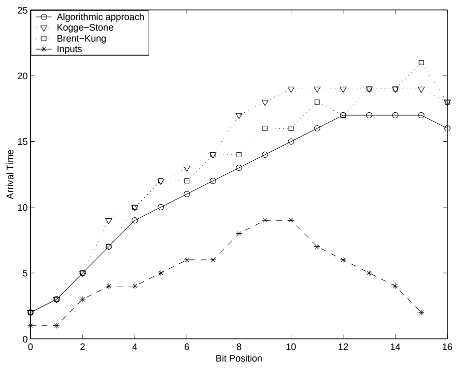


Fig. 15. Result of 16-bit case3

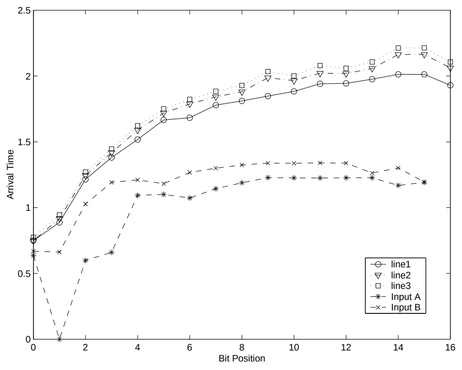


Fig. 16. Result of 16-bit case I

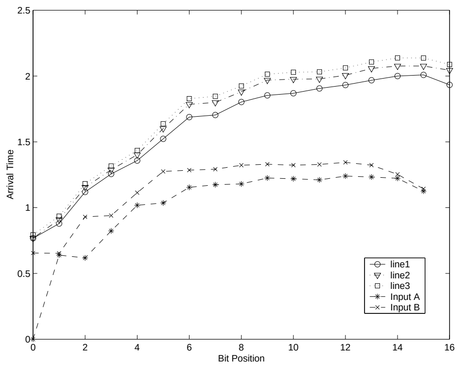


Fig. 17. Result of 16-bit case II

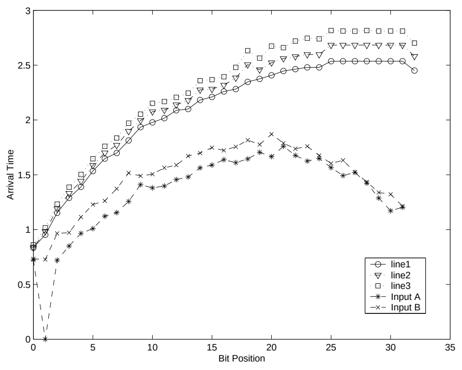


Fig. 18. Result of 32-bit case I

TABLE I

Cases	Distance from inputs			Number of (G, P) adders		
	Arithmetic	Kogge-Stone	Brent-Kung	Arithmetic	Kogge-Stone	Brent-Kung
8-bit 1	5	7	7	9(9)	17	11
8-bit 2	6	7	10	16(18)	17	11
8-bit 3	7	8	9	14(14)	17	11
16-bit 1	5	9	9	21(23)	49	26
16-bit 2	7	10	14	70(84)	49	26
16-bit 3	8	10	12	30(33)	49	26

reduction and the simple reduction, in which all (G, P) adders without fanout are removed. The resulting number of cells from the simple reduction is listed in parenthesis.

Table.I in all these three kinds of input delay profiles indicates that the prefix adder generated by the proposed approach is faster than either Kogge-Stone or Brent-Kung adders. Particularly in monotonously decreasing profiles, results of the proposed approach improve at least 40% in terms of speed, in comparison with Brent-Kung adders. However, a drawback is the incremented area. The penalty of keeping the minimal delay is a small area increment in a convex profile and a large area increment in a monotonously decreasing profile.

To check the soundness of the proposed algorithm, we tested the timing performance of generated adders under $0.13\mu m$ technology. The proposed algorithm first creates a netlist based on the gate delay model with constant average load capacitance. The timing of the resulting netlist is then reevaluated considering accurate gate delay, and further wire delay. The gate delay is calculated by the look-up table method, while the wire delay is computed using Elmore delay model. The delay profiles of the two inputs are given by a sum array and a $carry$ array, which are the outputs of a partial product reduction tree.

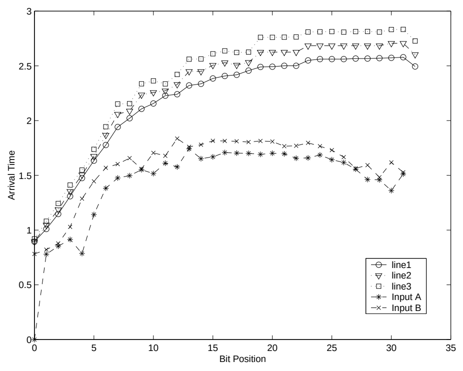


Fig. 19. Result of 32-bit case II

Two 16-bit and two 32-bit final adders are shown in Fig.16 to Fig.19. Two dashed lines with star and x-mark points represent two input delay profiles. The solid line with circle points gives the estimated timing result of the proposed algorithm. The dotted line with triangle points shows the recalculated timing including only gate delay. The timing including gate and wire delay is shown by the dotted line with square points.

Table.II summarize the max distances. Row 1 shows the max distance between the estimated timing and the timing with pure gate delay. Row 2 shows the max distance between the estimated timing and the timing with wire delay. Row 3 gives the error percentage introduced by wire delay.

TABLE II

	16-bit I	16-bit II	32-bit I	32-bit II
Distance between line1 and line2(ns)	0.151	0.116	0.158	0.136
Distance between line1 and line3(ns)	0.201	0.161	0.285	0.271
Percentage of wire delay	24.9%	28.0%	44.6%	49.8%

From the results, the error carried by load-capacitance is not quite large and is almost constant with different bit-width. The error introduced by wire delay originally is small, but increases quickly with the increase of data width. The proposed algorithm is practical if data width is not huge. But, with the increase of data width, the algorithm should be modified to regard the influence of interconnect. Buffer insertion and gate sizing also should be considered in delay estimation.

VII. Conclusions

In this paper, we proposed an algorithmic approach to generate an irregular parallel-prefix adder for a specific application. The prefix structure generated by the proposed algorithm has minimal delay at all outputs, and can cover all topologies of generic adders. The time complexity and space complexity of the algorithm are $O(n^3)$ and $O(n^2)$ respectively. The experimental results have the smallest delay of outputs, compared with Brent-Kung and Kogge-Stone prefix adders. This approach can be applied to generate the optimal final adder for a specific multiplier.

Acknowledgements

This work was supported in part under grants from NSF project number MIP-9987678 and the California MICRO program. The authors would like to thank Michael Chang, Tony Chen, Wenjing Rao, Shengrong Lin and the anonymous reviewers for their insightful comments.

References

[1] M. Lehman and N. Burla, ”Skip Techniques for high-speed carry-propagation in binary arithmetic circuits”, IRE Trans. Electron. Comput., pp.691-698, Dec 1961

[2] J. Hennesey and D. A. Patterson, *Computer Architecture - A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 1990. Appendix A.

[3] R. P. Brent and H. T. Kung, ”A regular layout for parallel adder”, IEEE Trans. Comput., Vol. C-31, no. 3, pp. 260-264, Mar. 1983.

[4] 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.

[5] B. Sugla and D. A. Carlson, ”Extreme Area-Time Tradeoffs in VLSI”, IEEE Trans. Comput., Vol. 39, no. 2, pp. 251-257, Feb. 1990.

[6] B. Parhami, *Computer Arithmetic*, New York, Oxford: Oxford University Press, 2000.

[7] W. C. Yeh and C. W. Jen, ”High-Speed Booth Encoded Parallel Multiplier Design”, IEEE Trans. Comput., Vol.49, no. 7, pp. 692-701, Jul. 2000.

[8] J.P. Fishburn, ”A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between”, 27th ACM/IEEE Design Automation Conference Proceedings, pp.361-4, 1990.