A Unified Adder Design

Yuke Wang
Department of Computer Science
University of Texas at Dallas
yuke@utdallas.edu

Keshab K. Parhi
Department of Electrical and Computer Engineering
University of Minnesota
parhi@ece.umn.edu

Abstract

We present a unified framework to compare and optimize various adders in the literature including carry-ripple adders, carry-look-ahead adders, prefix-adders, canonic adders, block-based carry-look-ahead adders, carry-skip adders, conditional-sum adders, carry-select adders, and hybrid adders. The logic of all those adders can be separated into two parts: the carry logic and the sum logic, while carries can be generated by a uniformed carry operator implemented by any of the prefix operator, the fco operator, MUX, the AND-OR gates, and pass-transistors. The most efficient way to generate carries is by layers instead of by groups, where layers are non-consecutive non-equal length collections. We demonstrate that the conditional-sum adders and carry-select adders have redundant sum logic and sub-optimal carry logic; therefore they should be eliminated. We further improve some prefix addition algorithms such as the Brent-Kung's adder for carry generation. The ideas discussed here are independent of the implementation technology and therefore useful for all implementations and technologies.

Section 1 Introduction

Being one of the fundamental components of any digital systems, the adders have been studied for about half a century. The numbers being added can be represented in 2's complement or signed redundant numbers. The carries can be the traditional carries or various Ling's carries. The basic logic operations in the implementation can be the prefix operator, the fco operator, MUX, the AND-OR gates, complex gates, or pass transistors. The adders can be designed for different objectives such as high speed, low power, or small area, and implemented in various technologies such as Bipolar, dynamic logic, Domino logic, complementary CMOS, and BiCMOS. Finally the addition algorithms can be classified as *carry-ripple adders, carry-look-ahead adders, prefix-adders, canonic adders, block-based carry-look-ahead adders, carry-skip adders, conditional-sum adders, carry-select adders,* and *hybrid adders.*

The *first contribution* of this paper is to point out that adders designed based on all these different addition algorithms can be separated into two parts: the carry logic and the sum logic. Carry-select adders, conditional-sum adders, and any other adders based on them including hybrid adders have redundant sum logic and can be simplified.

The *second contribution* of this paper is that we show all those different operators for carry generation such as the prefix operator, the fco operator, MUX, AND-OR, and pass transistors are more or less equivalent. Therefore the design of the carry logic is largely independent of the basic carry operator used in the design.

Based on the above unified adders with the uniformed carry operator, we can compare different adder designs and simplify the adder design into the optimal design of the carry logic with the uniformed carry operator in a certain implementation technology. The implementation technology determines how the basic carry operator is implemented, and its maximum fan-in and fan-out. Therefore the optimization of adders should be technology dependent. The optimal carry logic design is for different fan-out, fan-in, and different delay parameters, to derive the most efficient carry logic. In the literature it is often believed that the design is a trade-off of hardware and speed. Many adders increase the critical path to reduce the hardware, while few works have studied the optimization problem as what is the optimal design given the fan-in/fan-out limit and delay of the implementation technology.

We finally show that the key to design efficient carry logic is to generate the most significant bit carries first. When the most significant bit carry is generated, several other carries have been generated as the by-product. This approach is called the layered carry generation, which allows maximal sharing of carry generation among different bit positions. The original CLA adders generate every bit of carry in parallel without sharing any gates. Traditional block-based CLA adders do not share enough carry logic that each block has non-sharing carry logic. Canonic adders are limited to AND-OR network implementation and their sharing of AND network is limited. In our layered carry generation approach, many bits of carries can be obtained without a single extra gate. Our design will always obtain adders with fewer carry operators for the same delay and fan-out compared to conditional-sum adders and carry-select adders. Therefore we suggest to eliminate the use of conditional-sum adders and carry-select adders.

The rest of this paper is organized as follows. In Section 2, we present the background materials on

0-7803-7147-X/01/$10.00©2001 IEEE

adders. In Section 3, we present the unified carry logic and sum logic design for adders and the equivalence of carry operators. In Section 4, we present the efficient layered carry generation algorithm. Finally, in Section 5, we conclude the paper.

Section 2 Background

We use the following notations and definitions in this paper. The two operands in addition are $A=\{a_{n-1},...,a_1,a_0\}$ and $B=\{b_{n-1},...,b_1,b_0\}$. The sum is $S=\{S_{n-1},...,S_1,S_0\}$. The carry propagate/ generate signals are defined as $p_i=a_i+b_i$ and $g_i=a_ib_i$. We have the equation $p_ig_i=g_i$ and $p_i+g_i=p_i$. Other terms used in the literature are $k_i=\overline{a_i+b_i}$, and $p_i=a_i\oplus b_i$. The carry-in signal is denoted as $c_{in}=c_{-1}$, and the carry-out signal (c_{out}) is the carry generated by the most significant bit. The traditional carry generated at bit i is $c_i=g_i+p_ic_{i-1}=g_i+p_ig_{i-1}+p_ip_{i-1}g_{i-2}+...+p_ip_{i-1}...p_1p_0c_{-1}$, $S_i=a_i\oplus b_i\oplus c_{i-1}$. There are other variations of carries such as Ling's carries, and the carries for signed-digit based addition algorithms.

Adders are classified as *carry-ripple adders, carry-look-ahead adders, prefix-adders, canonic adders, block-based carry–look-ahead adders, carry-skip adders, conditional-sum adders, carry-select adders,* and *hybrid adders.* In total we have listed 9 different kinds of adders.

Ripple-carry adders generate carries one bit at a time. The carry c_i is generated only when all the carries $c_0 \sim c_{i-1}$ have been generated already. The *Carry-Look-Ahead* (CLA) adders on the other hand generate all the carries c_i in parallel. This can be accomplished in principle, however, would require a large number of inputs to gates and it is rendered impractical. One way to design the CLA adder is to use prefix computation which allows some sharing among different carries. *Prefix adders* include the Ladner and Fischer adder, Kogge-Stone adder, Brent-Kung's adder, and a few others. The *canonic adders* implement the parallel carry generation using multi-level AND and OR gates, while the AND network is used to generate $p_ip_{i-1}...g_j$ and the OR network is used to generate $c_i=g_i+p_ig_{i-1}+p_ip_{i-1}g_{i-2}+...+p_ip_{i-1}...p_1p_0c_{-1}$. The key idea for canonic adders is to share the AND products of $p_ip_{i-1}...g_j$ for different c_i. Prefix adders and canonic adders are mainly designed for minimal delay, e.g., canonic adders have delay of $2\lceil\log_r(n-1)\rceil+2-\delta$, and will require a large amount of hardware.

To trade-off for the hardware, the delay may be increased. *Block-based CLA adders* divide the n bit inputs into groups with normally group size 4, each group implements the carry-look-ahead logic. The groups are then interconnected by the ripple-carry method. The method of dividing into groups for generating carries is also used in the *carry-skip adders.* The n-bit adder is divided into groups with simple ripple-carry scheme used in each group. Every group also generates a group-carry-propagate signal that equals 1 if all stages internal to the group satisfy p_j=1. This is somehow similar to using ripple-carry as the building blocks and a parallel scheme for group carry generation.

Group based design can be applied further to derive the *conditional-sum adders* and *carry-select adders.* The principle for *conditional-sum adders* is to generate two sets of outputs for a given group of operand bits, each set includes k sum bits and an outgoing carry. Therefore conditional-sum adders generate sum bits directly instead of generating carries as adders discussed in previous paragraph. Similar idea is used for *carry-select* adders, where one partitions a large adder into fixed size adder sections and proceeds with simultaneous additions to generate sum-bits with appropriate carry input to select the true sum output. *Hybrid adders* are combinations of two different addition algorithms together, such as *CLA + conditional sum* or *CLA + carry select adders.*

One can see that the 9 different kinds of adders are designed for different delays. Moreover, some adder algorithms generate only carries and sum-bits are obtained by the standard formula $S_i=a_i\oplus b_i\oplus c_{i-1}$; while some other adders generate the sum bits directly. The current classification of addition algorithms has more to do with historical reasons than logical reasons. There is no comprehensive study of the logic relationships among those different adders. In Section 3, we will show that every adder can be separated into two parts: the sum logic and the carry logic.

The carry logic of an adder can be implemented by different logic operators such as the prefix operator, the fco operator, MUX, AND-OR gates, complex gates, and pass transistors. We first show the prefix operator and the fco operator are the same.

The prefix operator $\bullet$ introduced by Ladner and Fisher is defined as $\left(\frac{g}{k}\right)_i\bullet\left(\frac{g}{k}\right)_j=\left(\frac{g_i+\bar{k}_ig_j}{k_ik_j}\right)$ or as $(g_i,p_i)\bullet(g_j,p_j)=(g_i+p_ig_j,p_ip_j)$. The carry c_i is computed as $c_i=\left(\frac{g}{k}\right)_i\bullet\left(\frac{g}{k}\right)_{i-1}\bullet...\bullet\left(\frac{g}{k}\right)_0\bullet\left(\frac{c_{-1}}{1}\right)$. The fco operator introduced by Brent and Kung is defined as $(p_{i:j},g_{i:j})=(p_{i:k+1},g_{i:k+1})fco(p_{k:j},g_{k:j})$, where $p_{i:j}=p_{i:k+1}p_{k:j}$ and $g_{i:j}=g_{i:k+1}+p_{i:k+1}g_{k:j}$.

The carry c_i is computed as $(p_{i:0}, c_i) = (p_{i:i-1}, g_{i:i-1}) fco \ldots . fco(1, c_{-1})$.

The above two operators have been modified from the original literature to cope with the case that initial carry c_{-1} is not 0. It is easy to see that except notation differences, the prefix operator $\bullet$ and the *fco* operator perform exactly the same logic operation.

A new format of carry representation called function graphs is introduced in [1]. A function graph contains two kinds of vertices: non-terminal vertices shown as circles, and terminal vertices shown as squares. A non-terminal vertex has two children. A terminal vertex has no child. Vertices are labeled by variables or Boolean functions. For a non-terminal vertex labeled by a variable x, the two edges connected to two children represent the positive variable x and its complement $\bar{x}$, marked by "+" and "-" at the edges, and called "negative edge" and "positive edge" respectively. A terminal vertex represents a Boolean function which is the label of itself. A non-terminal vertex labeled by variable x represents a function $f = \bar{x} f_{lo} + x f_{hi}$, where f_{lo} is the function represented by the child node connected by the negative edge, and f_{hi} is the function represented by the child node connected by the positive edge.

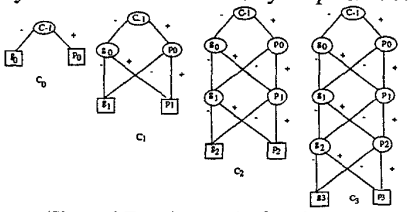

Figure 1 Function graph of carries c_0, c_1, c_2, c_3

Section 3 Unified Adder Design

In this section, we show that carry-ripple adders, conditional-sum adders, carry-select adders can be separated into sum logic and carry logic parts. Hybrid adders using carry-select adders/conditional-sum adders have sum logic and carry logic as well. All other additional algorithms generate only carries. We therefore arrive at a unified framework for adder design – generate all the carries first.

3.1 Carry Logic and Sum Logic

Figure 2 (b) shows a 4-bit carry-ripple adder; while (c) shows how to separate it into the sum logic, which are XOR gates represented by shadowed circles, and the carry logic which are AND-OR gates shown in (d).

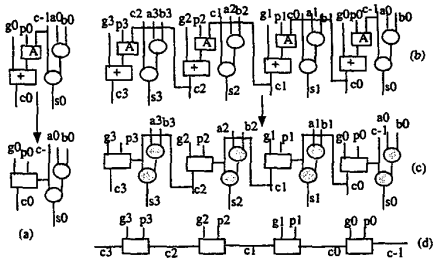

Figure 2 Sum logic and carry logic of ripple-carry adder

Following the notation in [14], Figure 3 (a) shows how to separate an 8-bit conditional-sum adder into the sum logic which are XOR gates marked by shadowed circles, and the carry logic which are MUXes represented by squares. For each position *i*, two different sum bits and carry bits are generated as $S_i^0 = a_i \oplus b_i, C_{i+1}^0 = a_i b_i = g_i, S_i^1 = a_i \overline{\oplus} b_i, C_{i+1}^1 = a_i + b_i = p_i$.

Figure 3 (b) shows that by moving the XOR gates from the top level to the bottom level, we can save those MUXes marked by shadow. The original design requires 2*9+10=28 MUXes, while the new design requires 5*2+7=17 MUXes, with an increased delay of one XOR gate. This extra delay would not be a problem if the conditional sum adder is not used in the critical path of a big adder; otherwise, we could keep bit positions 5, 6, and 7 unchanged and arrive at a design of the same delay with 5 MUXes less out of 28 MUXes.

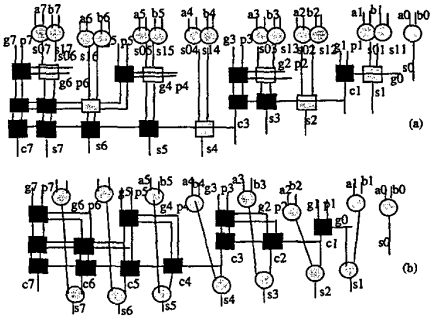

Figure 3 Sum logic and carry logic of conditional-sum adders

Finally, Figure 4 (a) shows the 4-bit carry-select adder; the sum logic is the shadowed XOR gates, and the carry logic is the squares of AND-OR/MUX gates. We can remove the redundant XOR gates and arrive at (b).

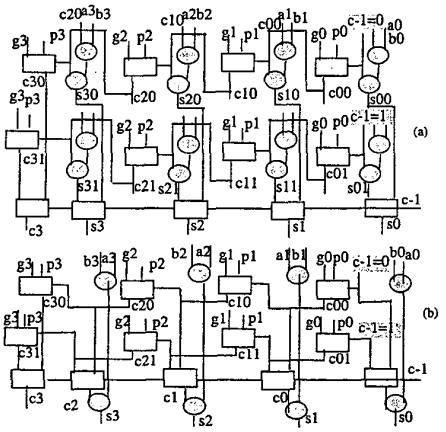

Figure 4 Sum logic and carry logic of carry-select adders

3.2 Uniformed Logic Operator for Carry Logic

One of the key properties of adders is the relation $g_i < p_i$, which implies that $g_i + p_i = p_i, g_i p_i = g_i$. The key function for carry generation is $c_i = g_i + p_i c_{i-1} = \bar{c}_{i-1} g_i + c_{i-1} p_i$. We call this basic logic operator as the *carry operator.*

Since $c_0 = g_0 + p_0 c_{-1} = \bar{c}_{-1} g_0 + c_{-1} p_0$, an one–bit adder can be designed using MUX or AND-OR gate, shown in Figure 5. For the same reason, the AND-OR gates can be interchanged with MUX in Figure 4 for the carry-select adder.

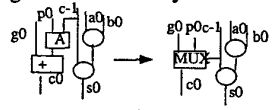

Figure 5 AND-OR equivalent to MUX for one-bit adder

Now we show the prefix operator based design can be interchanged with MUX as well. The fco operator/prefix operator is a logic operator with four inputs and two outputs defined as $(p_{i:j}, g_{i:j}) = (p_{i:k+1}, g_{i:k+1}) fco (p_{k:j}, g_{k:j})$, where $p_{i:j} = p_{i:k+1} p_{k:j}$ and $g_{i:j} = g_{i:k+1} + p_{i:k+1} g_{k:j}$. Due to the special relation that $g_{i:k+1} < p_{i:k+1}$, we have

$$g_{i:j} = g_{i:k+1} + p_{i:k+1} g_{k:j} = g_{i:k+1} \bar{g}_{k:j} + (g_{i:k+1} + p_{i:k+1}) g_{k:j} = g_{i:k+1} \bar{g}_{k:j} + p_{i:k+1} g_{k:j}.$$

Figure 6 shows the MUX based implementation of 4-bit carry generation using Brent-Kung's fco operator. It is easy to see that the new MUX based implementation can save much hardware while preserving the same connection topology.

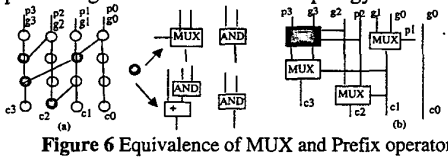

Figure 6 Equivalence of MUX and Prefix operator

Figure 7 shows an example of a carry-ripple chain implemented in various operators such as MUX (b), AND-OR (c), MUX-AND based prefix-operator (d), general prefix-operator (e), and pass transistors (f)(MCC chain, Figure 4 of [9]). A complex gate based implementation can also be derived easily, which we do not show it here.

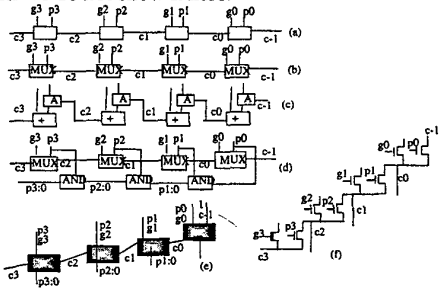

Figure 7 Different operators to implement the carry-chain.

3.3 Unified Adders with Uniformed Carry Operator

So far, we have established the fact that all adders have two parts: the sum logic and the carry logic; moreover, the carry logic is implemented using the same carry operator for various algorithms. This unified framework provides a base for comparison and optimization of various addition algorithms. Previously it is hard to compare one adder algorithm to another since they are designed and implemented in different ways and technologies.

Figure 8 shows 4-bit adders designed based on carry-ripple (a), carry-select (b), and conditional-sum algorithm (c). The shadowed circles and squares are equivalent to XOR gates for the sum logic, and all other gates are the carry operators. The 4-bit adder is a basic building block of many larger adders; therefore the optimization of the 4-bit adder is very important. We will present a new design in Section 4.

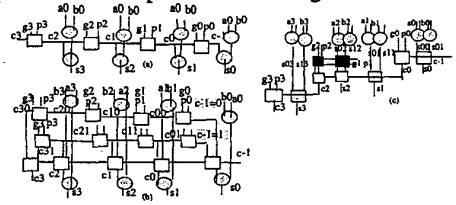

Figure 8 Unified 4-bit adder with carry-operators

Section 4 Layered Carry Generation

Besides the unified sum logic and carry logic implemented by the uniformed carry operator, almost all previous addition algorithms share two more properties: (1) carries are generated in groups starting from the least significant bits; (2) each class of addition algorithm generates carries for fixed delays,

e.g., conditional-sum adders for $O(\log n)$ delay and carry-select adders for $O(n)$ delay.

In this section, we propose a layered carry generation scheme for flexible delay and fan-out parameters. The algorithm generates carries starting from the most significant bits instead of the least significant bits. One key observation is that once the most significant bit carry is generated, several other carries have been generated as the by-product. The most significant bit carry together with those by-product carries is called a layer. Therefore carries in layers are groups of non-consecutive carries while the traditional group based algorithms generate consecutive carries in groups.

Assuming g_i, p_i are available for each i, the following algorithm generates all the carries c_i. The algorithm accepts three parameters (n,d,f), where n is the number of inputs bit, d is the delay in terms of the number of carry-operators, f is the maximum fan-out of the carry operator for the given technology.

Layered-Carry Generation Algorithm - (n,d,f)

(1) Design the most significant carry c_n using the carry operator; as the by-product, a layer of carries will be generated as well. If the delay d is the minimal delay, then the divided-and-conquer method has to be used to achieve the minimal delay; otherwise, there will be many possible ways to achieve the given delay d for c_n.

(2) Repeat (1) for the next most significant carry which has not been generated and its layer of carries.

(3) Fine-tune each carry operator into the most efficient one for different positions. We may implement a group of carry operators by two-level or multi-level logic gates or complex gates in CMOS.

In the following, we show by examples how to design the carries by the above algorithm for minimal delays and non-minimal delays.

4.1 Minimal Delay Layered Carry Generation

Minimal delay is defined as the $\log(n+1)$ carry operator delay for n bit input with a non-zero carry-in. It is interesting to see that the conditional-sum adders are of minimal delay, and the prefix adder Brent-Kung's adders are not of minimal delay. Other minimal delay adders include Kogger-stone adders. Many algorithms have been provided in [12] to design minimal delay adder for various fan-out limit.

Figure 9 (a) shows the new design of a 4-bit adder based on the layered carry generation algorithm. We generate the carry c_3 by divided-and-conquer to achieve the minimal delay. In the process of generating c_3, we have already generated c_0, c_2; the only remaining carry to be generated is c_1, which can be obtained by one extra carry operator. Therefore the 4 carries are divided into two layers $(c_3,\ c_2, c_0)$ and c_1; while the logic needed for the layer $(c_3,\ c_2, c_0)$ is the same logic for one single bit carry c_3.

For comparison purpose, Figure 9 (b) is a MUX adder presented in [2], (c) is a design of the carry c_3 based on canonic adder algorithm, and 3 more 4-bit adders can be found in Figure 8. It is easy to see that for one c_3, the canonic adder needs 7 AND gates and 5 OR gates, which is the same number of gates for all the carries in (a). However, the canonic adder still needs to generate c_2, c_1, c_0, which needs about 10 gates. The performance of various adders is summarized in Table 1. The new layered carry generation uses about half amount of gates while achieve the minimal delay and lowest fan-out.

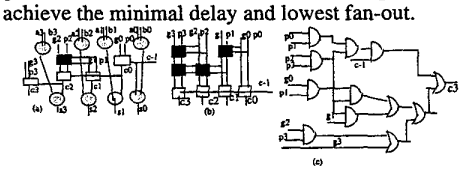

Figure 9 4-bit carries

Table 1 Performance Comparison

#ofbits	algorithm	delay	fan-out	operators	Layer/group
4	C-select	4	4	13	0123
4	C-sum	3	3	11	0, 12, 3
4	MUX	3	4	12	0123
4	canonic	3	3	>10	
4	layered	3	2	6	320, 1
8	C-select	6	5	24	0123,4567
8	C-sum	4	5	30	0,12,3456,7
8	layered	4	3	16	7620,542,3
16	layered	5	4	40	

Figure 10 shows the design of 8-bit adders. The conditional sum adder is shown in (a) and the new design is shown in (b). The layers are as $(c_7, c_6, c_2, c_0), (c_5, c_4, c_2), (c_3)$. Finally Figure 11 is the 16-bit layered carry logic with the most significant layer as (15, 14, 6, 2, 0).

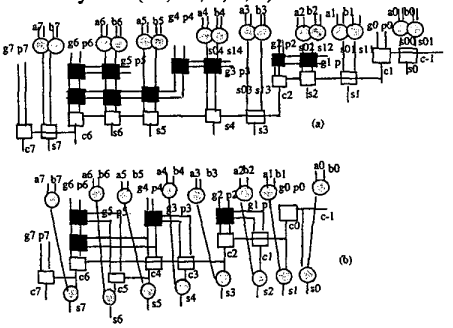

Figure 10 8-bit adders generated by layers

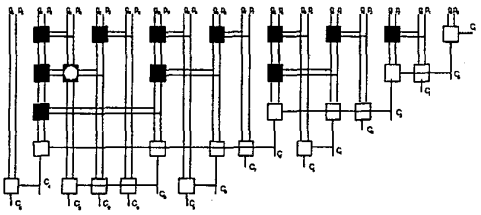

Figure 11 16-bit carry logic generated by layers

One thing to notice here is the carry c_5 in Figure 10 (b) is designed differently from c_5 in Figure 3 (b). The key difference is that Figure 3 has no carry-in c_{-1} and therefore the minimal delay is 3 carry operators, while Figure 10 has the carry-in c_{-1} with the minimal delay of 4. Given 4 delay, we do not need the two extra carry operators used in Figure 3 to generate c_5, instead c_5 can be generated by one extra carry operator after c_4 is available. No group based carry design algorithm can derive this kind of optimal result.

The above examples show and we can further prove that conditional-sum adders always use more carry operators with larger fan-out to achieve the same delay compared to the layered carry generation adder. Moreover, the minimal delay layered adders have fewer carry operators than the carry-select adders with larger delay and fan-out limits. Therefore we suggest to eliminate the use of conditional-sum adders and carry-select adders. Any hybrid adders based on these two can be improved as well.

4.2 Non-Minimal Delay Layered Carry Generation

Our Layered-Carry Generation Algorithm can be applied to general non-minimal delay adders, which is very different from other algorithms in the literature. Figure 12 (a) is the original Brent-Kang design with 28 carry operators while (b) is a new design with 23 carry operators satisfying exactly the same principle as the original design. The most significant layer is shown as (15, 14, 13, 11, 7, 3, 1). Notice here the operators are general carry-operators, not necessarily prefix operators. For example, they can be substituted into MUX or AND-OR.

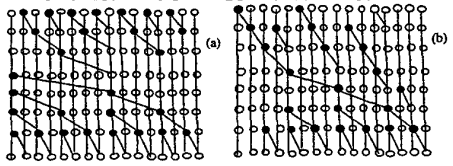

Figure 12 16-bit Brent-Kung carry logic

Section 5 Conclusions

The logic of all adders can be separated into two parts: the carry logic and the sum logic. Conditional-sum adders, carry-select adders, and any other adders based on these two have redundant sum logic; while many other adders have non-optimal carry logic. The basic logic operators for carry generation are almost more or less equivalent. We show how to design efficient carry logic by layers of carries and improve some prefix addition algorithms such as the Brent-Kung's adders for carry generation. The ideas discussed here are independent of implementation technology and therefore useful for all implementations and technologies. The ideas in this paper open up many further researches to design optimized carry logic.

The authors want to thank many people who have given comments on early drafts and talks given in different schools.

References

[1] Yuke Wang, and K. Parhi, " New Low Power Adders Based on New Representations of Carry Signals", Asilomar Conference, 2000.

[2] K. Parhi, "Low energy CSMT carry-generators and binary adders", IEEE Trans. on VLSI, December 1999, pp. 450-462.

[3] M. J. Flynn and S. F. Oberman, "Modern Research in Computer Arithmetic", Class notes, Stanford University, Autumn quarter, 1998-1999.

[4] H. Ling, "High speed binary adder", IBM J. Res. Develop., Vol. 25, No. 3, May 1981, pp. 156-166.

[5] R. W. Doran, "Variants of an Improved carry look ahead adder", IEEE Trans. on Computers, Vol. 37, No. 9, September 1988. pp. 1110-1113.

[6] N. Quach and M. J. Flynn, "High Speed Addition in CMOS", IEEE Trans. on Computers, Vol. 41, No. 12, December 1992, pp. 1612-1615.

[7] R. Brent and H. T. Kung, "A regular layout for parallel adders", IEEE Trans. on Computers, Vol. C-31, No. 3, March 1982, pp. 260-264.

[8] J. Dobson and G. Blair,"A fast twos-complement VLSI adder design", IEE Electronics Letters , Vol. 31, No. 20, pp. 1721-1722, Sept 1995.

[9] T. Lynch and E. Swartzlander, "A Spanning Tree Carry Lookahead Adder", IEEE Trans. on Computers, Vol. 41, No. 8, August 1992, pp. 931-939.

[10] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations", IEEE Trans. on Computers, Vol. C-22, pp. 786-792, Aug. 1973.

[11] R. E. Ladner and M. J. Fischer, "Parallel prefix computation", J. ACM, Vol. 27, pp. 831-838, 1980.

[12] S. Knowles, "A family of adders", Proc. of Symp. Comput. Arithmetic, 1999.

[13] B. Parhami, "Computer arithmetic, algorithms and hardware design", Oxford University Press, 2000.

[14] K. Hwang, *Computer arithmetic, principles, architecture, and design*, John Wiley and Sons, 1979, New York,

[15] I. Koren, "*Computer arithmetic algorithms*", Prentice Hall, 1993, Englewood Cliffs, New Jersey.