A Regular Layout for Parallel Adders*

Richard P. Brent † H.T. Kung ‡
MEMBER, IEEE MEMBER, IEEE

Abstract

With VLSI architecture, the chip area and design regularity represent a better measure of cost than the conventional gate count. We show that addition of n-bit binary numbers can be performed on a chip with a regular layout in time proportional to $\log n$ and with area proportional to n.

Index Terms: Addition, area-time complexity, carry lookahead, circuit design, combinational logic, models of computation, parallel addition, parallel polynomial evaluation, prefix computation, VLSI.

1 Introduction

We are interested in the design of parallel "carry lookahead" adders suitable for implementation in VSLI architecture. The addition problem has been considered by many other authors. See, for example, [1, 4, 6, 7, 11, 13, 14]. Much attention has been paid to the tradeoff between time and the number of gates, but little attention has been paid to the problem of connecting the gates in an economical and regular way to minimise chip area and design costs. In this paper we show that a simple and regular design for a parallel adder is possible.

In §2 we briefly describe our computational model. §3 contains a description of the addition problem and shows how it reduces to a carry computation problem. The basis of our method, the reduction of carry computation to a " prefix" computation, is described in §4. Although the same idea was used by Ladner and Fischer [8], their results are not directly applicable because they ignored fan out restrictions and used the gate count rather than area as a complexity measure.

In §5 we use the results of §4 to give a simple and regular layout for carrying computation. Our construction demonstrates that the addition of n-bit numbers can be performed in time $O(\log n)$, using area $O(n \log n)$. The implied constants are sufficiently small that the method is quite practical, and it is especially suitable for a pipelined adder. In §6 we generalize the result of §5, and show that n-bit numbers can be added in time $O(n/w + \log w)$, using area $O(w \log w + 1)$, if the input bits from each operand are available w at a time (for $1 \le w \le n$). Choosing $w \sim n/\log n$ gives the result that n-bit addition can be performed in time $O(\log n)$ and area $O(n)$.

*Appeared as Report TR CMU-CS-79-131, Department of Computer Science, Carnegie-Mellon University (June 1979); and in *IEEE Transactions on Computers*, **C-31** (March 1982), 260–264 (manuscript received May 12, 1980; revised February 3, 1981 and October 1, 1981). Retyped (with corrections) in LaTeX by Frances Page, July 2000.

†R.P. Brent is with the Department of Computer Science, Australian National University, Canberra, Australia. (Current address: Oxford University Computing Laboratory, Oxford OX1 3QD, UK.)

‡H.T. Kung is with the Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213. (Current address: Department of Computer Science, Harvard University, Cambridge, MA 02138, USA.)

2 The Computational Model

Our model is intended to be general, but at the same time realistic enough to apply (at least approximately) to current VLSI technology. We assume the existence of circuit elements or "gates" which compute a logical function of two inputs in constant time. An output signal can be divided ("fanned out") into two signals in constant time. Gates have constant area, and the wires connecting them have constant minimum width (or, equivalently, must be separated by at least some minimal spacing). At most two wires can cross at any point.

We assume that a signal travels along a wire of any length in constant time. This is realistic as propagation delays are limited by line capacitancies rather than the velocity of light. A longer wire will generally have a larger capacitance, and thus require a larger driver, but we can neglect the driver area as it typically need not exceed a fixed percentage of the wire area [10].

The computation is assumed to be performed in a convex planar region, with inputs and outputs available on the boundary of the region. Our measure of the cost of a design is the *area* rather than the number of gates required. This is an important difference between our model and earlier models of Brent [1], Winograd [14], and others. For further details of our model, see [3]; for motivation and discussion of models similar to ours, see [9] and [12]. A feature of our approach is that we strive for regular layouts in order to reduce design and implementation costs. For VSLI, regularity is one of the most important design critertia; so we shall not compromise the regularity of a design for the sake of efficiency. Since "regularity" is difficult to quantize, we have not included it in our theoretical cost measure, although this would be desirable.

3 Outline of the General Approach

Let $a_n a_{n-1} \ldots a_1$ and $b_n b_{n-1} \ldots b_1$ be n-bit binary numbers with sum $s_{n+1} s_n \ldots s_1$. The usual method for addition computes the s_i by

$$\begin{aligned} c_0 &= 0\,, \\ c_i &= (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})\,, \\ s_i &= a_i \oplus b_i \oplus c_{i-1}, \qquad\qquad i = 1, \ldots, n\,, \\ s_{n+1} &= c_n \end{aligned}$$

where $\oplus$ means the sum mod 2 and c_i is the carry from bit position i .

It is well known that the c_i can be determined using the following scheme:

$$\begin{aligned} && c_0 &= 0\,, \\ && c_i &= g_i \vee (p_i \wedge c_{i-1})\,, && (1) \\ \text{where} && g_i &= a_i \wedge b_i \\ \text{and} && p_i &= a_i \oplus b_i \end{aligned}$$

for $i = 1, 2, \ldots, n$. One can view the g_i and p_i as the *carry generate* and *carry propagate* conditions at bit position i . The relation (1) corresponds to the fact that the carry c_i is either generated by a_i and b_i or propagated from the previous carry c_{i-1}. This is illustrated in Fig. 1.

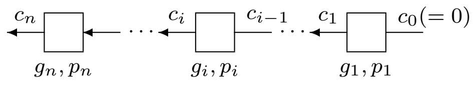
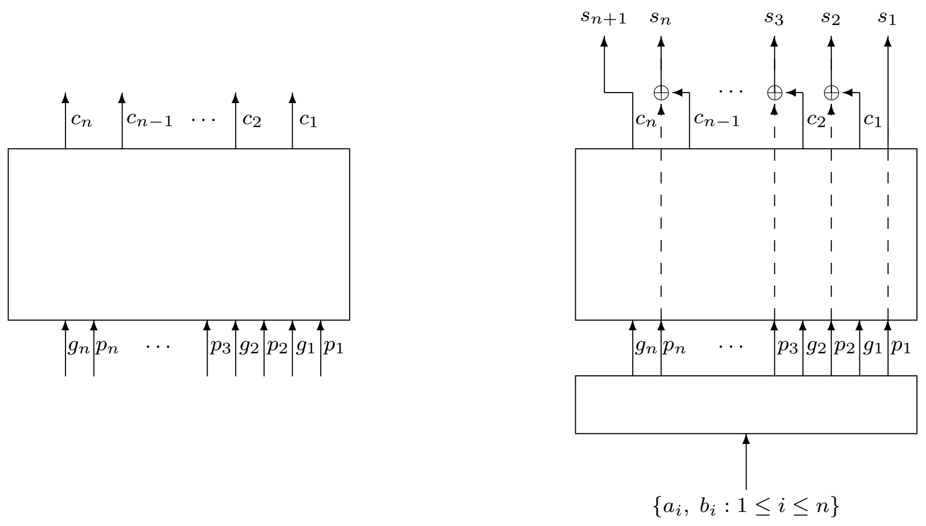


Figure 1: Carry chain

In §5 we present a regular and area-efficient layout design for computing all the carries in parallel assuming that the g_i and p_i are given. The design of a parallel adder is then straightforward and is illustrated in Fig. 2. Notice that in Fig. 2(b) the bottom rectangle represents the combinational logic that transforms the a_i and b_i into the g_i and p_i. For computing the s_i we use the fact that $s_i = p_i \oplus c_{i-1}$ for $i = 1, \ldots, n$.



c_n
c_{n-1}
$\cdots$
c_2
c_1
g_n p_n $\cdots$ p_3 g_2 p_2 g_1 p_1
s_{n+1}
s_n
s_3
s_2
s_1
$\oplus$
$\{a_i,\ b_i : 1 \leq i \leq n\}$

Figure 2: (a) Abstraction of a parallel carry chain computation, and (b) abstraction of a parallel adder based on the design for the carry chain computation.

4 Reformulation of the Carry Chain Computation

We define an operator "o" as follows:

$$(g,p)o(\widehat{g},\widehat{p}) \quad = \quad (g \vee (p \wedge \widehat{g}), p \wedge \widehat{p})$$

for any Boolean variables $g, p, \widehat{g}$ and $\widehat{p}$.

Lemma 1: Let

$$(G_i, P_i) = \begin{cases} (g_1, p_1) & \text{if } i = 1\,, \\ (g_i, p_i)o(G_{i-1}, P_{i-1}) & \text{if } 2 \leq i \leq n\,. \end{cases}$$

Then

$$c_i = G_i \qquad \text{for } i = 1, 2, \ldots, n\,.$$

Proof: We prove the lemma by induction on i. Since $c_0 = 0$, (1) above gives

$$c_1 = g_1 \vee (p_1 \wedge 0) = g_1 = G_1$$

so the result holds for $i - 1$. If $i > 1$ and $c_{i-1} = G_{i-1}$, then

$$\begin{aligned} (G_i, P_i) &= (g_i, p_i)o(G_{i-1}, P_{i-1}) \\ &= (g_i, p_i)o(c_{i-1}, P_{i-1}) \\ &= (g_i \vee (p_i \wedge c_{i-1}), p_i \wedge P_{i-1})\,. \end{aligned}$$

Thus $\qquad G_i = g_i \vee (p_i \wedge c_{i-1})$

and from (1) we have $\qquad G_i = c_i\,.$

The result now follows by induction. □

Lemma 2: The operator "o" is associative.

Proof: For any $(g_3, p_3), (g_2, p_2), (g_1, p_1)$, we have

$$\begin{array}{rcl} [(g_3,p_3)o(g_2,p_2)]o(g_1,p_1) & = & [g_3 \vee (p_3 \wedge g_2), p_3 \wedge p_2]o(g_1,p_1) \\ & = & [g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1), p_3 \wedge p_2 \wedge p_1] \\ \text{and} \quad (g_3,p_3)o[(g_2,p_2)o(g_1,p_1)] & = & (g_3,p_3)o[g_2 \vee (p_2 \wedge g_1), p_2 \wedge p_1] \\ & = & [g_3 \vee (p_3 \wedge (g_2 \vee (p_2 \wedge g_1))), p_3 \wedge p_2 \wedge p_1]\,. \end{array}$$

One can check that the right-hand sides of the above two expressions are equal using the distributivity of "$\wedge$" over "$\vee$". (The dual distributive law is not required.) □

To compute the c_i it suffices to compute all the (G_i, P_i), but by Lemmas 1 and 2

$$(G_i, P_i) = (g_i, p_i)o(g_{i-1}, p_{i-1})o \cdots o(g_1, p_1)$$

can be evaluated in any order from the given g_i and p_i. This is the motivation for the introduction of the operator "o". (Intuitively, G_i may be regarded as a "block carry generate" condition, and P_i as a "block carry propagate" condition.)

5 A Layout for the Carry Chain Computation

Consider first the simpler problem of computing (G_i, P_i) for $i = n$ only. Since the operator "o" is associative, (G_n, P_n) can be computed in the order defined by a binary tree. This is illustrated in Fig. 3 for the case $n = 16$. In the figure each black processor performs the function defined by the operator "o" and each white processor simply transmits data. The white and black processors are depicted in Fig. 4. Note that for Fig. 3 each processor is required to produce only one of its two identical outputs, and the units of time are such that one computation by a black processor and propagation of the result takes unit time.

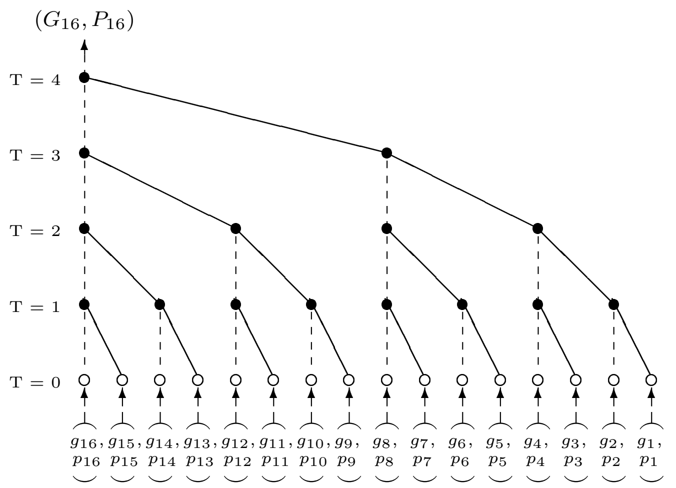
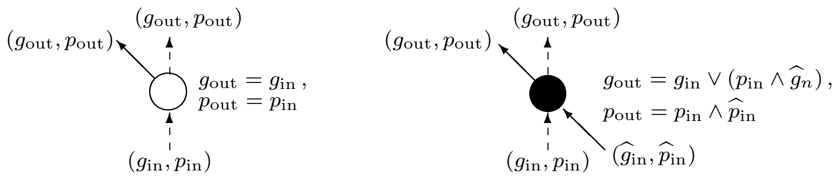


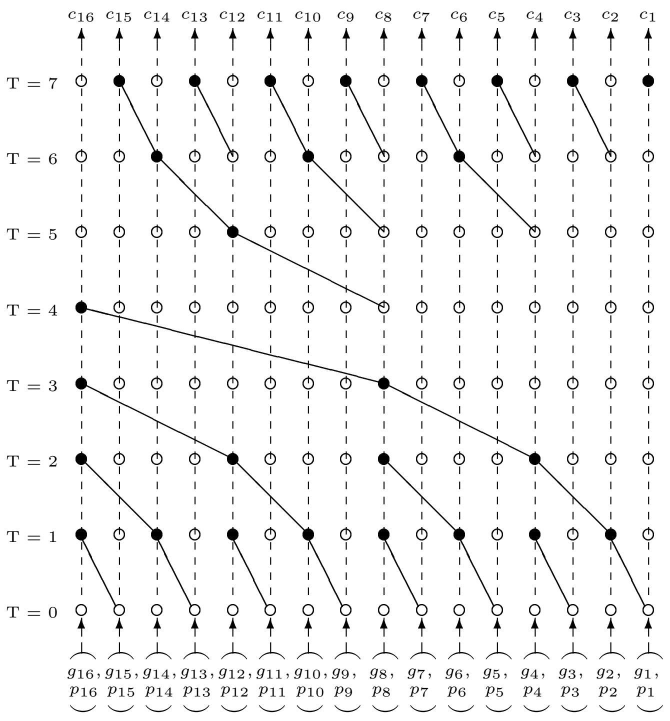
Figure 3: Computation of (G_{16}, P_{16}) using a tree structure



$(g_{\text{out}}, p_{\text{out}})$
$(g_{\text{out}}, p_{\text{out}})$
$g_{\text{out}} = g_{\text{in}}$,
$p_{\text{out}} = p_{\text{in}}$
$(g_{\text{in}}, p_{\text{in}})$
$(g_{\text{out}}, p_{\text{out}})$
$(g_{\text{out}}, p_{\text{out}})$
$g_{\text{out}} = g_{\text{in}} \vee (p_{\text{in}} \wedge \widehat{g}_n)$,
$p_{\text{out}} = p_{\text{in}} \wedge \widehat{p}_{\text{in}}$
$(g_{\text{in}}, p_{\text{in}})$
$(\widehat{g}_{\text{in}}, \widehat{p}_{\text{in}})$

Figure 4: (a) White processor and (b) black processor

Consider now the general problem of computing the (G_i, P_i) for all $1 \leq i \leq n$. This computation can be performed by using the tree structure of Fig. 3 once more, this time inverted (that is, the root is visited first). We illustrate the computation, for the case $n = 16$, in Fig. 5.



c_{16} c_{15} c_{14} c_{13} c_{12} c_{11} c_{10} c_9 c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1
T = 7
T = 6
T = 5
T = 4
T = 3
T = 2
T = 1
T = 0
g_{16}, p_{16} g_{15}, p_{15} g_{14}, p_{14} g_{13}, p_{13} g_{12}, p_{12} g_{11}, p_{11} g_{10}, p_{10} g_9, p_9 g_8, p_8 g_7, p_7 g_6, p_6 g_5, p_5 g_4, p_4 g_3, p_3 g_2, p_2 g_1, p_1

Figure 5: Computation of all the carries for $n = 16$

It is easy to check that at time $T = 7$, all the (G_i, P_i) are computed along the top boundary of the network. As the final outputs, we only keep the G_i, which are the carries c_i. From the layout shown in Fig. 5, we have the following results.

Theorem 3: For $n \geq 2$, all the carries in an n-bit addition can be computed in time proportional to $\log n$ and in area proportional to $n \log n$, and so can the addition.

6 A Pipeline Scheme for Addition of Long Integers

We define the *width* w of a parallel adder to be the number of bits it accepts at one time from each operand. For the parallel adder corresponding to the network in Fig. 5, $w = 16$. We have hitherto assumed that the width of a network is equal to the number n of bits in each operand. Here we consider the case $w < n$. We show that this case can be handled efficiently using a pipeline scheme on a network which is a modification of the one depicted in Fig. 5.

For simplicity, assume that n is divisible by w. One can partition an n-bit integer into n/w segments, each consisting of w consecutive bits. To illustrate the idea, suppose that $w = 16$. Then the carry chain computation corresponding to each segment can be done on the network in Fig. 5, and the computations for all the segments can be pipelined, starting from the least significant segment. The results coming out from the top of the network are not the final solutions, though. Results corresponding to the ith least significant segment ($i > 1$) have to be modified by applying $(G_{(i-1)w}, P_{(i-1)w})$ on the right using the operator "o". To facilitate this modification, we superimpose another tree structure on the top half of the network, as shown in Fig. 6.

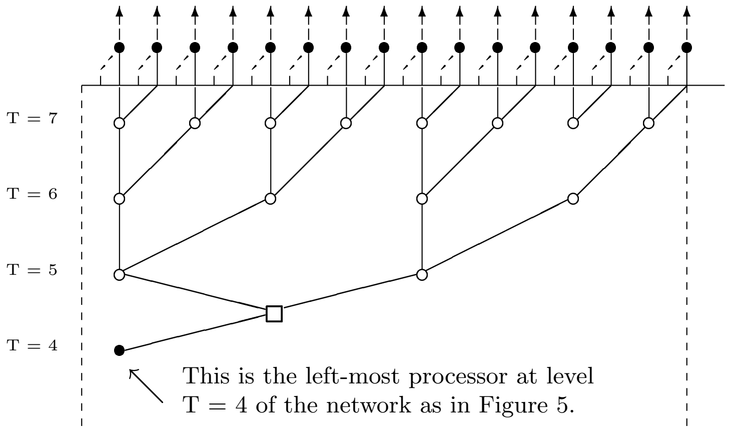

T = 7
T = 6
T = 5
T = 4
This is the left-most processor at level T = 4 of the network as in Figure 5.

Figure 6: Additional tree structure to be superimposed on the top half of the network in Fig. 5

Using this additional tree, the contents of the "square" processor (denoted by "□") are sent to all the leaves, which are black processors. The square processor, shown in Fig. 7, is an accumulator which initially has value $(g, p) = (0, 1)$, and successively has values $(g, p) = (G_{(i-1)w}, P_{(i-1)w})$ for $i = 2, 3, \ldots$ At the time when a particular $(G_{(i-1)w}, P_{(i-1)w})$ reaches the leaves, it is combined with the results just coming out from the old network there.

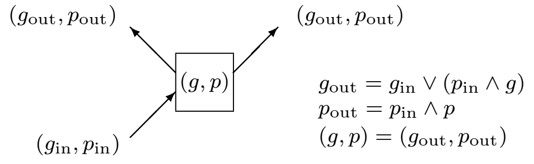

$(g_{\text{out}}, p_{\text{out}})$
$(g_{\text{out}}, p_{\text{out}})$
(g, p)
$g_{\text{out}} = g_{\text{in}} \vee (p_{\text{in}} \wedge g)$
$p_{\text{out}} = p_{\text{in}} \wedge p$
$(g, p) = (g_{\text{out}}, p_{\text{out}})$
$(g_{\text{in}}, p_{\text{in}})$

Figure 7: The "square" processor that accumulates $(G_{(i-1)w}, P_{(i-1)w})$

By this pipeline scheme, we have the following result.

Theorem 4: Let $1 \leq w \leq n$. Then all the carries in an n-bit addition can be computed in time proportional to $(n/w) + \log w$ and in area proportional to $w \log w + 1$, and so can the addition. When $w = 1$, the method outlined in this section is essentially the usual serial carry-chain computation. From Theorem 4 we have the following.

Corollary 1: The area-time product for n-bit addition is $O(n \log w + w \log^2 w + 1)$, which is $O(n \log^2 n)$ when $w = n$, $O(n \log n)$ when $w = n/\log n$, and $O(n)$ when w is a constant.

One can similarly obtain an upper bound on AT^α (where A and T stand for area and time, respectively) for any $\alpha \geq 0$, and for each α one can choose w to minimise the upper bound [2].

7 Summary and Conclusions

The preliminary and final stages of binary addition with our scheme (generation of (g_i, p_i) and computation of $s_i = p_i \oplus c_{i-1}$ respectively) are straightforward. Figs. 4 and 5 illustrate that the intermediate phase (fast carry computation) is conceptually simple, and the layout illustrated in Fig. 5 is regular. The design of the white processor is trivial, and the black processor is about as complex as a one-bit adder. After these two basic processors are designed, we can simply replicate them and connect their copies in the regular way illustrated in Fig. 5. We conclude that using the approach of this paper, parallel adders with carry lookahead are well-suited for VLSI implementation.

Mead and Conway [10] considered several lookahead schemes, but concluded that "they added a great deal of complexity to the system without too much gain in performance". To show that this comment does not apply to our scheme, suppose that the operations "$\wedge$", "$\vee$" and "$\oplus$" take unit time. Table 1 gives the computation time for our scheme and for a straightforward serial scheme, where the c_i are computed from (1) for various n. (n is the number of bits in each operand.) For $n = 2^k$ the general formulas are $4k$ and $2n - 1$ respectively.

TABLE 1
COMPARISON OF PARALLEL AND SERIAL ADDITION TIMES

n	Time (parallel)	Time (serial)
8	12	15
16	16	31
32	20	63
64	24	127

Based on our scheme, Guibas and Vuillemin [5] have designed a 32-bit parallel adder and implemented it on a chip using NMOS. They estimate that with the particular technology they used, their 32-bit parallel adder is about 4 times faster that a 32-bit straightforward serial adder.

In this paper we assumed a binary number system and restricted our attention to two's complement arithmetic. Only minor modifications of our results are required to deal with one's complement arithmetic or sign and magnitude representations of signed integers.

Brent and Kung [3] consider the problem of multiplying n-bit integers, and show that the area A and time T for any method satisfy

$$\begin{aligned} AT &\geq K_1 n^{3/2} \\ \text{and} \qquad AT^2 &\geq K_2 n^2 \end{aligned}$$

for certain constants $K_i > 0$ (assuming the model of §2 with some mild additional restrictions). For binary addition we can achieve

$$\begin{aligned} AT &= O(n) && \text{by a trivial serial method,} \\ \text{and} \qquad AT^2 &= O(n \log^2 n) && \text{by the results in §6.} \end{aligned}$$

Thus, asymptotically speaking, implementing binary multiplication is harder that implementing binary addition if either AT or AT^2 is used as the complexity measure. More discussions on the area-time complexity of binary arithmetic can be found in [2], where a general measure AT^α for any $\alpha \geq 0$ is used.

In deriving the layout of Fig. 5 we used only one distributive law. Thus, the layout could be used to evaluate arithmetic expressions of the form

$$g_n + p_n\{g_{n-1} + p_{n-1}[\ldots p_3(g_2 + p_2 g_1)\ldots]\} \tag{2}$$

where the g_i, p_i are numbers and the black processor in Fig. 4(b) now computes $g_{\text{out}} = g_{\text{in}} + p_{\text{in}}\widehat{g}_{\text{in}}$ and $p_{\text{out}} = p_{\text{in}}\widehat{p}_{\text{in}}$. Note that when $p_2 = \cdots = p_n = x$ expression (2) corresponds to the polynomial

$$g_n + g_{n-1}x + \cdots + g_1 x^{n-1}\,.$$

References

[1] R P Brent, "On the addition of binary numbers", *IEEE Trans Comput*, vol C-19, pp 758–759, 1970.

[2] R P Brent and H T Kung, "The chip complexity of binary arithmetic", in *Proc 12th Annual ACM Symposium Theory of Comput*, April 1980, pp 190–200.

[3] —, "The area-time complexity of binary multiplication", *J Ass Comput Mach*, vol 28, pp 521–534, July 1981.

[4] H L Garner, "A survey of some recent contributions to computer arithmetic", *IEEE Trans Comput*, vol C-25, pp 1277–1282, 1976.

[5] L Guibas and J Vuillemin, private communication, August 1980.

[6] K Hwang, *Computer Arithmetic: Principles, Architecture and Design*, New York: Wiley, 1978.

[7] D J Kuck, *The Structure of Computers and Computations*, New York: Wiley, 1978.

[8] R E Ladner and M J Fisher, "Parallel prefix computation", *J Ass Comput Mach*, vol 27, pp 831–838, October 1980.

[9] C E Leiserson, "Area-efficient VLSI computation", PhD dissertation, Department of Comput Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[10] C A Mead and L A Conway, *Introduction to VLSI Systems*, Reading, MA: Addison-Wesley, 1980.

[11] J E Savage, *The Complexity of Computing*, New York: Wiley, 1976.

[12] C D Thompson, "Area-time complexity for VSLI", in *Proc 11th Annual ACM Symposium Theory of Comput*, May 1979, pp 81–88.

[13] C Tung, "Arithmetic", in *Computer Science*, A F Cardenas, L Press and M A Marin (eds), New York: Wiley-Interscience, 1972.

[14] S Winograd, "On the time required to perform addition", *J Ass Comput Mach*, vol 12, no 2, pp 277–285, 1985.

Richard P. Brent (M'72) was born in Melbourne, Australia, on April 20, 1946. He received the BSc (Hons) degree in mathematics from Monash University, Australia, in 1968, and the MS and PhD degrees in computer science from Stanford University, Stanford, CA, in 1970 and 1971, respectively.

From 1971 to 1972 he was employed in the Mathematical Sciences Department at the IBM T J Watson Research Center, Yorktown Heights, NY. Since 1972 he has been at the Australian National University, Canberra, Australia, where he is currently[1] Professor and Head of the Department of Computer Science. His research interests include VLSI design, computer arithmetic, analysis of algorithms, and computational complexity.

H T Kung (M'78) graduated from National Tsing-Hua University, Taiwan, in 1968 and recieved the PhD degree from Carnegie-Mellon University, Pittsburgh, PA, in 1974.

Currently[2], he is an Associate Professor of Computer Science at Carnegie-Mellon University, where he leads a research group in the design and implementation of high-performance VLSI systems. From January to September 1981 he was an Architecture Consultant to ESL Inc, a subsidiary of TRW. His research interests are in paradigms of mapping algorithms and applications directly on chips, and in theoretical foundations of VLSI computations.

Dr Kung serves on the editorial board of the *Journal of Digital Systems* and is the author of over 50 technical papers in computer science.

[1]This was written in 1981. Brent is currently Professor of Computing Science at Oxford University, UK.

[2]Kung is currently William H Gates Professor of Computer Science and Electrical Engineering at Harvard University, Cambridge, MA.