NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-79-131

A REGULAR LAYOUT FOR PARALLEL ADDERS

R.P. Brent
Department of Computer Science
Australian National University
Canberra, A.C.T. 2600
Australia.

H.T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
U.S.A.

June 1979

Copyright © 1979 by R.P. Brent and H.T. Kung

This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370, NR 044-422.

ABSTRACT

With VLSI architecture the chip area is 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 in time proportional to log n and with area proportional to n log n.

Key Words and Phrases

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

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

1. Introduction

We are interested in the design of parallel "carry lookahead" adders suitable for implementation in VLSI architecture. The addition problem has been considered by many other authors. See, for example, Winograd [65], Brent [70], Tung [72], Savage [76], and Kuck [78]. 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 minimize chip area and design costs. In this paper we show that a simple and regular design for a parallel adder is possible.

In Section 2 we briefly describe our computational model. Section 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 Section 4. Although the same idea was used by Ladner and Fischer [77], their results are not directly applicable because they ignored fanout restrictions, and used the gate count rather than area as a complexity measure.

In Section 5 we use the results of Section 4 to give a simple and regular layout for carry 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 Section 6 we generalize the result of Section 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 $1 \le w \le n$).

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 capacitances 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 need not exceed a fixed percentage of the wire area: see Mead and Conway [79].

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 Winograd [65], Brent [70] and others. For further details of our model, see Brent and Kung [79].

3. Outline of the General Approach

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

$$c_0 = 0,$$
$$c_i = (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1}),$$

$$s_i = a_i \oplus b_i \oplus c_{i-1} \text{ , } i=1,\cdots,n,$$
$$s_{n+1} = c_n,$$

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's can be determined using the following scheme:

$$c_0 = 0,$$
$$c_i = g_i \vee (p_i \wedge c_{i-1}), \quad (3.1)$$

where

$$g_i = a_i \wedge b_i,$$

and

$$p_i = a_i \oplus b_i,$$

for $i=1,2,\cdots,n$. One can view the g_i and p_i as the "carry generate" and "carry propagate" conditions at bit position i. The relation (3.1) 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 Figure 3.1.

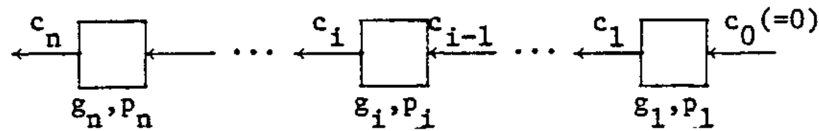

Figure 3.1: The carry chain

In Section 5 we present a layout design for computing all the carries in parallel assuming that the g_i's and p_i's are given. The design of a parallel adder is then very straightforward, and is illustrated in Figure 3.2. Note that in Figure 3.2(b), the bottom rectangle represents a buffer that transforms the a_i's and b_i's into the g_i's and p_i's. For computing the s_i's we use the fact that $s_i = p_i \oplus c_{i-1}$ for $i=1,\cdots,n$.

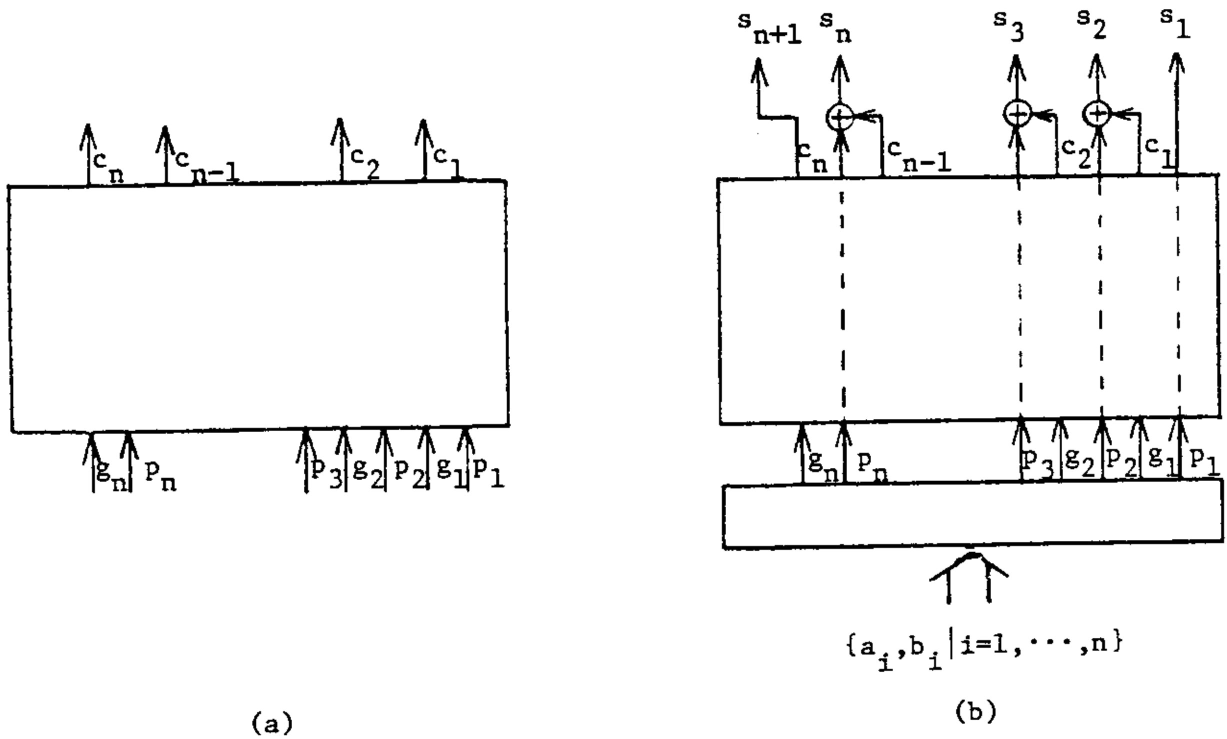

Figure 3.2: (a) Abstraction of a parallel carry chain computation, and (b) abstraction of a parallel adder based on the design for the carry chain computation.

4. Reformulation of the Carry Chain Computation

We define an operator "o" as follows:

$$(g,p) \circ (\hat{g},\hat{p}) \overset{\text{def}}{=} (g \vee (p \wedge \hat{g}), p \wedge \hat{p}),$$

for any Boolean variables g, p, $\hat{g}$ and $\hat{p}$. The following two lemmas show why the operator "o" is useful for carry computation.

Lemma 4.1:

Let

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

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

Proof:

We prove the Lemma by induction on i. Since $c_0 = 0$, (3.1) 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

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

Thus

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

and, from (3.1), we have

$$G_i = c_i.$$

The result now follows by induction. □

Lemma 4.2:

The operator "o" is associative.

Proof:

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

$$[(g_3,p_3) \circ (g_2,p_2)] \circ (g_1,p_1) = [g_3 \vee (p_3 \wedge g_2), p_3 \wedge p_2] \circ (g_1,p_1)$$
$$= [g_3 \vee (p_3 \wedge g_2) \vee (p_3 \wedge p_2 \wedge g_1),\ p_3 \wedge p_2 \wedge p_1],$$

and

$$(g_3,p_3) \circ [(g_2,p_2) \circ (g_1,p_1)] = (g_3,p_3) \circ [g_2 \vee (p_2 \wedge g_1), p_2 \wedge p_1]$$
$$= [g_3 \vee (p_3 \wedge (g_2 \vee (p_2 \wedge g_1))), p_3 \wedge p_2 \wedge p_1].$$

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 c_i it suffices to compute (G_i, P_i), but, by Lemmas 4.1 and 4.2,

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

can be evaluated in <u>any</u> order from the given g_i's and p_i's. 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

Recall that for computing the carries it suffices to compute the (G_i, P_i) for all $i=1,\cdots,n$. Consider first the simpler problem of computing only (G_n, P_n). 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 Figure 5.1 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 Figure 5.2. Note that for Figure 5.1 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 results takes unit time.

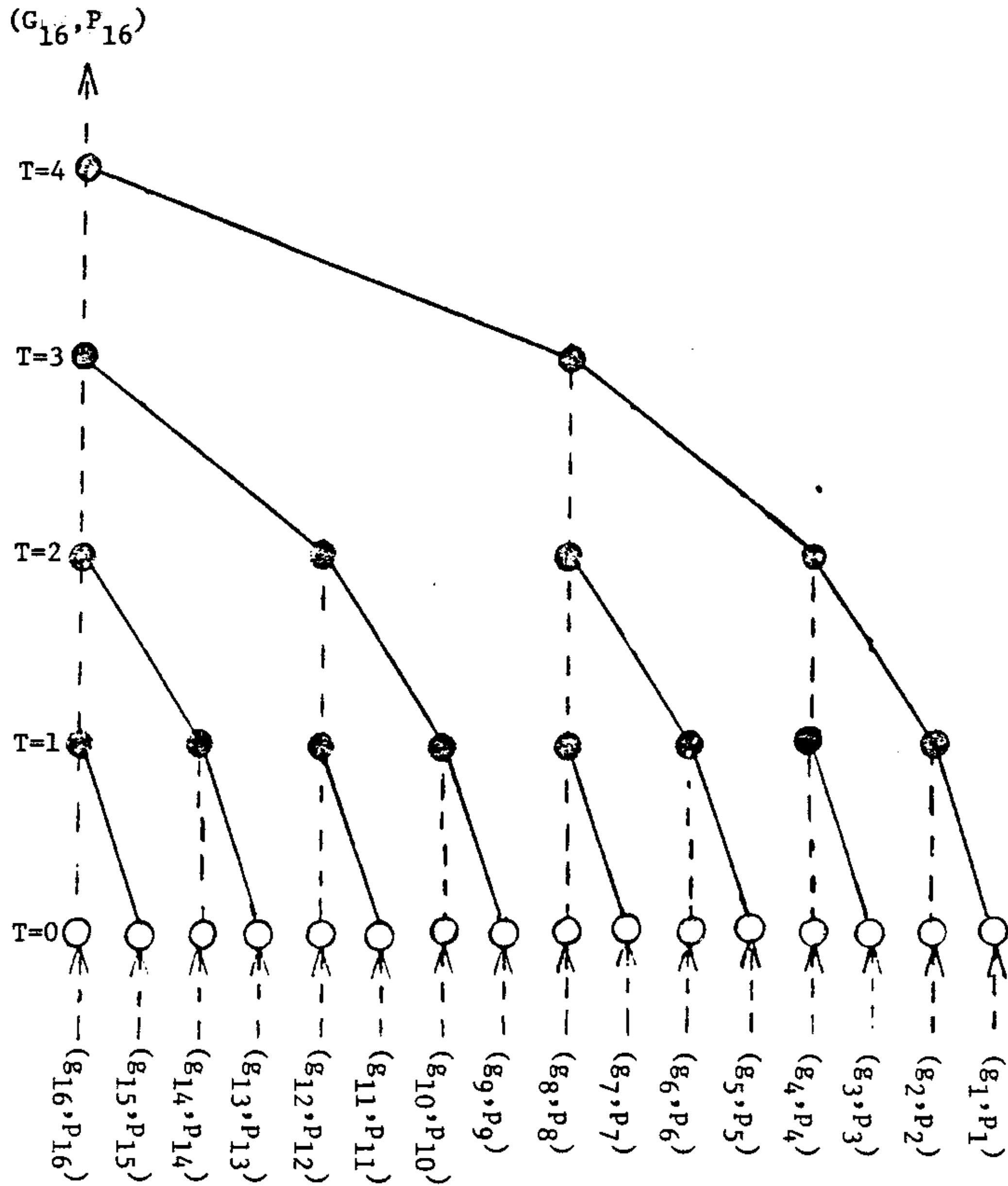

Figure 5.1: The computation of (G_{16},P_{16}) using a tree structure.

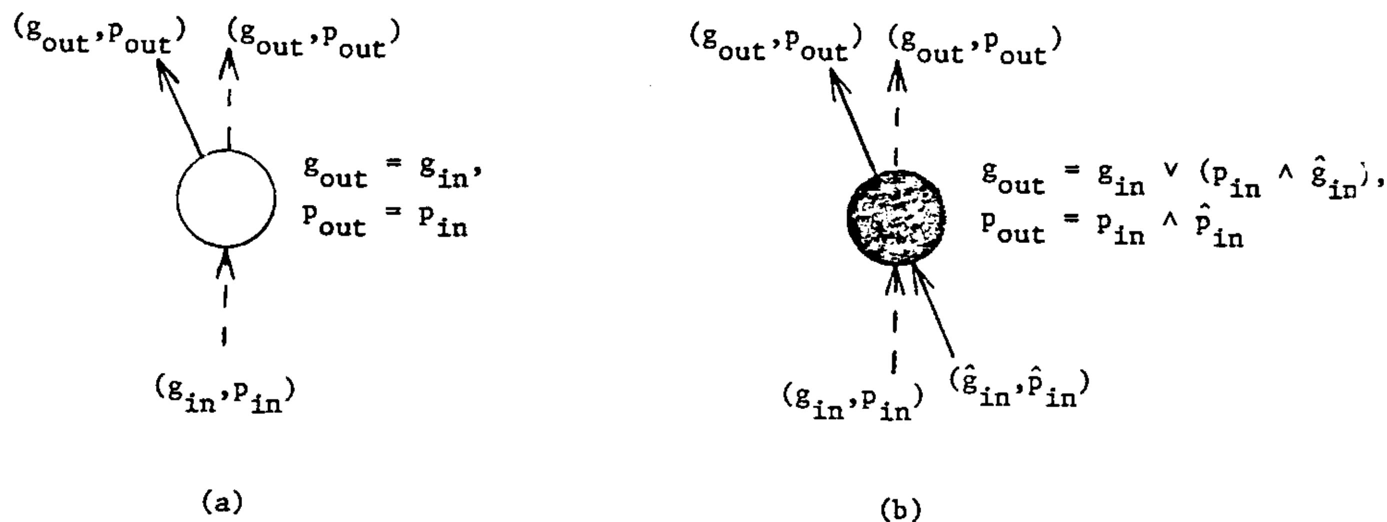

Figure 5.2: (a) The white processor, and (b) the black processor.

Consider now the general problem of computing <u>all</u> the (G_i,P_i) for $i=1,\cdots,n$. This computation can be performed by using the tree structure of Figure 5.1 once more, this time in the reverse order. We illustrate the computation, for the case $n=16$, in Figure 5.3. 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 Figure 5.3, we have the following theorem.

<u>Theorem 5.1</u>: All the carries in an n-bit addition can be computed in time proportional to $\log n$ and in area proportional to $n \log n$, $n \geq 2$.

<u>Corollary 5.1</u>: Addition of two n-bit binary numbers can be performed in time proportional to $\log n$ and in area proportional to $n \log n$, $n \geq 2$.

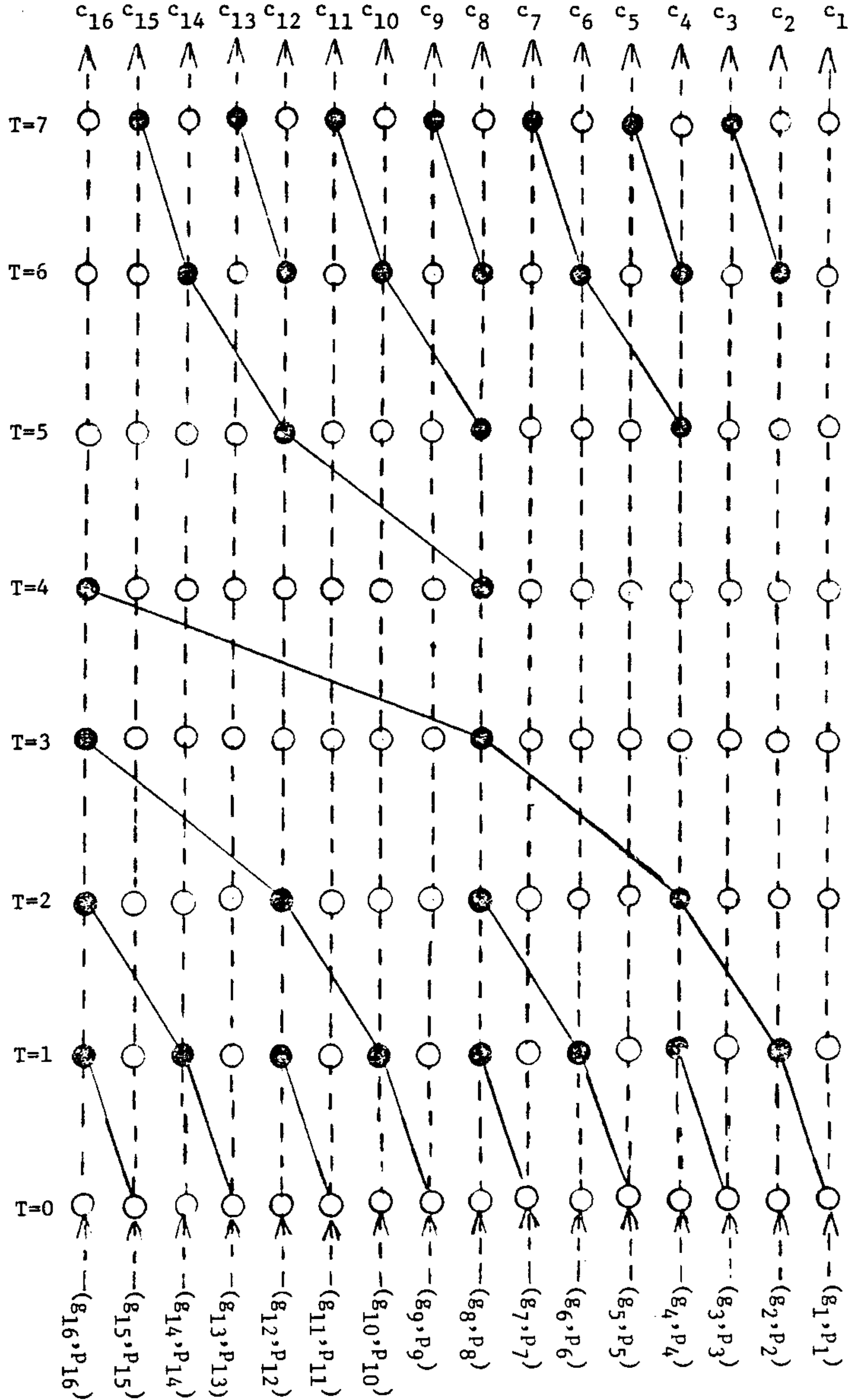

Figure 5.3: The computation of all the carries for n=16.

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 Figure 5.3, $w = 16$. We have hitherto assumed that the width of a network is equal to the number n of bits in each operand. In this section 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 Figure 5.3.

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 Figure 5.3, 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 i-th 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 "$\circ$". To facilitate this modification, we superimpose another tree structure on the top half of the network, as shown in Figure 6.1. Using this additional tree, the contents of the "square" processor (denoted by "$\square$") are broadcast to all the leaves, which are black processors. The square processor, shown in Figure 6.2, 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. By this pipeline scheme and Theorem 5.1, we have the following result:

Theorem 6.1: 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 $w \log w + 1$.

From Theorem 6.1, the area-time product is $O(n \log w + w \log^2 w + n)$, which is $O(n \log^2 n)$ when w=n, and $O(n)$ when w is a constant. When w=1 the method outlined in this section is essentially the usual serial carry-chain computation.

T=7

T=6

T=5

T=4

This is the same left-most processor at level T=4 of the network as in Figure 5.3.

Figure 6.1: The additional tree structure to be superimposed on the top half of the network in Figure 5.3.

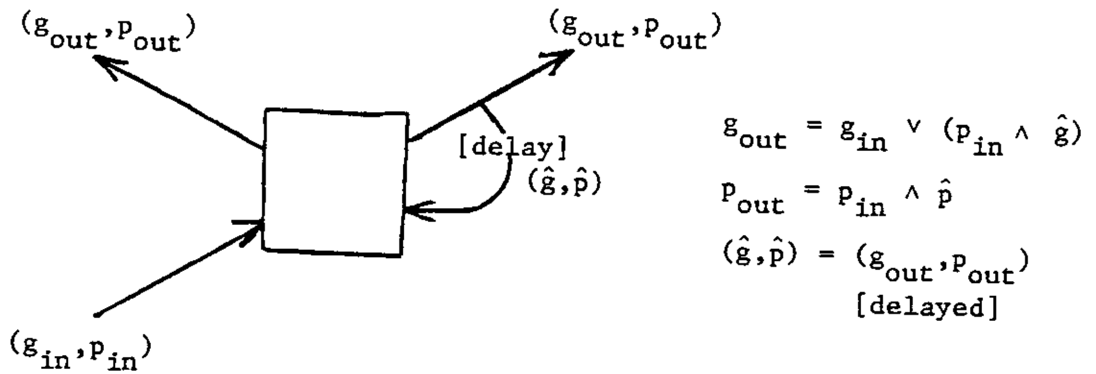

Figure 6.2: The "square" processor that accumulates $(G_{(i-1)_w}, P_{(i-1)_w})$

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. Figures 5.2 and 5.3 illustrate that the intermediate phase (fast carry computation) is conceptually simple, and the layout illustrated in Figure 5.3 is extremely 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 Figure 5.3. We conclude that, using the approach of this paper, parallel adders with carry lookahead are well-suited for VLSI implementation.

Mead and Conway [79, Chapter 5] considered several lookahead schemes, but concluded that "they added a great deal of complexity to the system without 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 7.1 gives the computation time for our scheme and for a straightforward serial scheme where the c_i are computed from (3.1) for various n. (n is the number of bits in each operand.) For $n=2^k$ the general formulae are $4k$ and $2n-1$ respectively.

Table 7.1 Comparison of parallel and serial addition times

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

In this paper we assumed a binary number system and restricted our attention to 2's complement arithmetic. Only minor modifications of our results are required to deal with 1's complement arithmetic or sign and magnitude representations of signed integers.

Brent and Kung [79] consider the problem of multiplying n-bit binary integers, and show that the area A and time T for any method satisfy

$$AT \ge K_1 n^{3/2}$$

and

$$AT^2 \ge K_2 n^2$$

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

$$AT = O(n)$$

by a trivial serial method, and

$$AT^2 = O(n \log^3 n)$$

by the method of Section 5. Thus, binary multiplication is harder than binary addition if either AT or AT^2 is used as the complexity measure.

In the proof of Lemmas 4.1 and 4.2 we used only one distributive law. Thus, the layout of Figure 5.1 could be used to evaluate arithmetic expressions of the form

$$(7.1) \qquad g_n + p_n\{g_{n-1} + p_{n-1}[\ldots p_3(g_2+p_2g_1)\ldots]\}$$

where g_i, p_i are numbers and the black processor in Figure 5.2(b) now computes $g_{out} = g_{in} + p_{in}\hat{g}_{in}$ and $p_{out} = p_{in}\hat{p}_{in}$. Note that the case $p_2 = \cdots = p_n = x$ of (7.1) is the polynomial

$$g_n + g_{n-1}x + \cdots + g_1x^{n-1} .$$

REFERENCES

Brent, R.P. [70], "On the Addition of Binary Numbers", IEEE Transactions on Computers, C-19 (1970), pp. 758-759.

Brent, R.P. and Kung, H.T. [79], "The Area-Time Complexity of Binary Multiplication", to appear as a Technical Report, Dept. of Computer Science, Carnegie-Mellon Univ., July 1979.

Kuck, D.J. [78], The structure of computers and computations, Vol. 1, John Wiley & Sons, New York, 1978.

Ladner, R.E. and Fischer, M.J. [77], "Parallel Prefix Computation", Proc. 1977 International Conference on Parallel Processing, 1977, pp. 213-223.

Mead, C.A. and Conway, L.A. [79], Introduction to VLSI Systems, Addison Wesley, 1979.

Savage, J.E. [76], The Complexity of Computing, John Wiley & Sons, New York, 1976.

Tung, C. [72], "Arithmetic", in Computer Science, ed. by A.F. Cardenas, L. Press, M.A. Marin, Wiley-Interscience, New York, 1972.

Winograd, S. [65], "On the Time Required to Perform Addition", J.ACM 12 (1965), pp. 277-285.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-79-131	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A REGULAR LAYOUT FOR PARALLEL ADDERS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) R.P. BRENT & H.T. KUNG		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE JUNE 1979
		13. NUMBER OF PAGES 19
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)