Area–Time Optimal Adder Design

BELLE W. Y. WEI, MEMBER, IEEE, AND CLARK D. THOMPSON

Abstract—**In this paper, we present a systematic method of implementing a VLSI parallel adder. First, we define a family of adders, based on a modular design. Our design uses three types of component cells, which we implement in static CMOS. We then formulate the adder design as a dynamic programming problem, optimizing with respect to area and time. The result is an area–time optimal adder in our design family. We illustrate our approach by implementing a 66-bit adder for use in a floating point processor. In addition, we indicate how to use our method for implementations in technologies and design styles other than static CMOS.**

Index Terms—**Carry look-ahead adders, domino logic design, dynamic programming, parallel prefix circuits, static CMOS design, time–area optimization, TTL adder design, VLSI adders.**

I. Introduction

Addition is the heart of computer arithmetics, and the arithmetic unit is often the work horse of a computational circuit. As a result, fast circuits for addition have been studied extensively. Most authors study addition in the context of TTL design [1]–[4], for which gate count and delay are performance factors. These factors are not sufficient to evaluate a design for possible implementations in VLSI [5]. Accordingly, in this paper, we optimize our adder circuit with respect to a refined model of VLSI delay and area.

Since 1982, a few authors [6] have proposed various VLSI designs for fast addition. Their models have limited fan-out. This proves too restrictive for VLSI implementations, as fan-out can be traded off for shorter interconnects and a smaller area, which may result in a faster circuit. In other proposals [7], [8], interconnect area and its attendant delay, crucial performance parameters for VLSI design, are not explicitly modeled and evaluated.

In this paper, we present a realistic and systematic method for constructing an area–time optimal parallel adder. Our approach is based on Ladner and Fischer's "parallel prefix computation" [9], and is essentially a look-ahead addition. This is described in Section II. In Section III, we specify three basic circuit blocks of a parallel adder. Three types of cells used to implement the circuit blocks are designed and analyzed in Section IV. From the timing analysis of Section IV, we optimize our adder design with respect to latency by solving a dynamic programming problem. This is described in Section V. In Section VI, we extend the dynamic programming formulation to optimize both latency and area. Based on the results from Section VI, we implemented a 66-bit adder for use in SPUR (symbolic processing using RISC's) floating point processor [10]–[12]. The design and area–time tradeoff curve is presented in Section VII. In Section VIII, we show that our method is general to cover a wide range of technologies and circuit design styles. In the last section, we summarize our approach to adder designs, and indicate that our method is an efficient implementation scheme for wide data width addition.

Manuscript received September 13, 1987; revised July 29, 1988. This work was supported by the National Science Foundation, through its Design, Tools and Test Program, under Grant MIP 8706139.

B. W. Y. Wei was with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720. She is now with the Department of Electrical Engineering, San Jose State University, San Jose, CA 95192.

C. D. Thompson is with the Department of Computer Science, University of Minnesota, Duluth, MN 55812.

IEEE Log Number 9034543.

II. Mathematical Description

In this section, we present and review briefly the mathematical formulations and terminologies to be used in the following sections. It has been known that binary addition can be transformed into a parallel computation by introducing an associative operator o [9]. If we define the carry generate term, gIN_i, and the carry propagate term, pIN_i, for each bit position i, then the carry c_i for each bit obeys the following recurrence relation:

$$gIN_i = a_i b_i$$

$$pIN_i = a_i \oplus b_i$$

$$c_i = G_i \quad \text{for} \quad i = 1, 2, \cdots, n$$

where

$$(G_i, P_i) = \begin{cases} (gIN_1, pIN_1) & \text{if} \quad i = 1 \\ (gIN_i, pIN_i) o (G_{i-1}, P_{i-1}) & \text{if} \quad n \geq i > 1 \end{cases} \tag{2.1}$$

and o is a concatenation operator defined as follows:

$$(g_l, p_l) o (g_r, p_r) = (g_l + p_l g_r, p_l p_r). \tag{2.2}$$

Note that o is not commutative. Its left argument (g_l, p_l) is treated differently from its right argument (g_r, p_r). After the carry bit c_i is computed, the sum bit s_i is

$$s_i = pIN_i \oplus c_{i-1} \quad \text{for} \quad i = 2, \cdots, n$$

$$s_1 = p_1. \tag{2.3}$$

Given the fact that o is associative, we can choose a m such that $i \geq m > 1$ and rewrite $(G_{i,1}, P_{i,1})$ as follows:

$$(G_{i,1}, P_{i,1}) = (G_{i,m}, P_{i,m}) o (G_{m-1,1}, P_{m-1,1}) \tag{2.4}$$

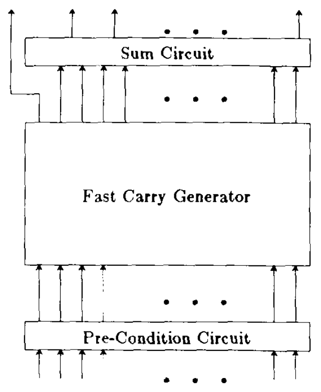


Fig. 1. Three functional blocks of an adder.

where

$$(G_{i,m}, P_{i,m}) = \begin{cases} (gIN_m, pIN_m) & \text{if } i = m \\ (gIN_i, pIN_i) o (G_{i-1,m}, P_{i-1,m}) & \text{if } i > m. \end{cases} \quad (2.5)$$

We observe that $(G_{i,m}, P_{i,m})$ and $(G_{i-m+1,1}, P_{i-m+1,1})$ have similar functional forms. Both are functions of $i - m + 1$ consecutive input bits and both require $i - m$ applications of the associative operator o. As a result, both can be computed by the same circuit.

III. Implementation

To implement the functions defined in Section II, three circuit blocks shown in Fig. 1 are required. The first circuit (labeled *precondition circuit*) gates in adder inputs a_i and b_i to generate the initial pIN_i and gIN_i for each bit position i. The computed p and g terms are then fed into the *fast carry generator* which performs the operations defined in (2.1) and (2.2). It is the *fast carry generator* that allows accelerated carry computations and this is the focus of our study. The third block is a *sum* circuit, consisting of a row of $\oplus$ gates, to combine the carry propagate bits (pIN_i) from the first block with the carry bits (c_i) from the second block, according to (2.3).

We use three basic types of tiling cells to implement the parallel carry computation: black cells, white cells, and driver cells. The terms "black" and "white cells come from Brent and Kung [6], and are shown in Fig. 2. Note that some of the inputs to our black and white cells "pass through" the cells. Specifically, the (g_r, p_r) inputs of the black cell are available as outputs. This convention greatly simplifies our wiring diagrams.

The black cell performs the associative concatenation defined in (2.2). Based on a static CMOS implementation, the black cell is of two categories, *ba* and *bb*. The *ba* cell shown in Fig. 3(a) gates in positive-true signals and produces complemented outputs. The *bb* cell, shown in Fig. 3(b) gates in complemented inputs and outputs positive-true signals. Each *bb* or *ba* cell is further composed of p and g subcells. The

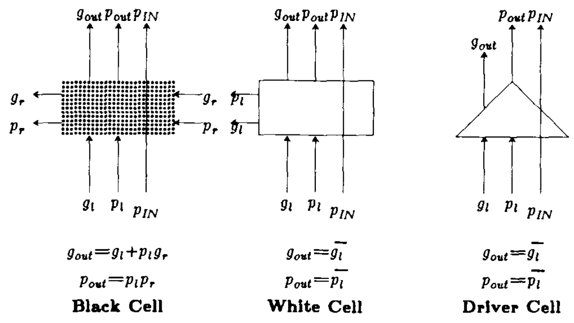


Fig. 2. Three basic types of tiling cells.

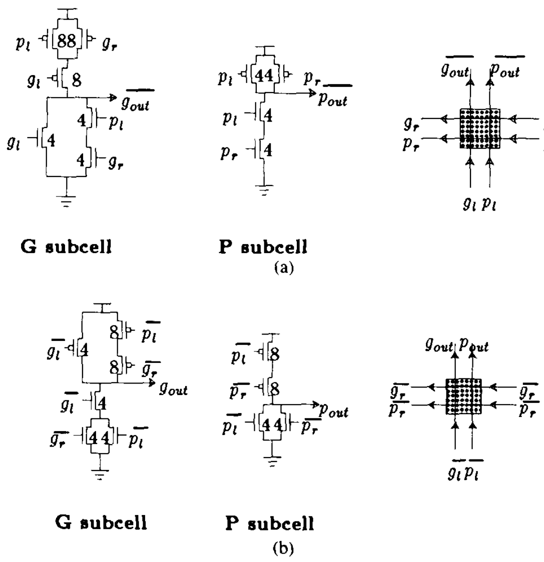


Fig. 3. Black cell implementations in static CMOS. (a) The black *ba* cell. (b) The black *bb* cell.

p subcell produces a p_{out} signal and the g subcell produces a g_{out} signal.

Definition: The fan-out f is the number of subcells that a signal drives.

For example, the fan-out for p_l (or $\overline{p_l}$) inside a black cell is 2, as it drives both p and g terms. However, the fan-out for g_l, g_r, p_r and their complements is 1 as they each drive only one subcell.

If we employ static CMOS implementations which feature inverting logic, sometimes we need inverters, called "white" cells, to ensure a proper signal polarity. These cells shown as *wa* are depicted in Fig. 4. Also in Fig. 4 is a modified white cell, *wb*, which provides a "turning corner" for signals. In the case of long wire interconnects or large fan-outs, we use a specially ratioed inverting driver, either in single stage or cascaded stages. In summary, the black cell is the "computation" cell. White cells are used for electrical requirements and driver cells are for performance improvements.

IV. Circuit Models

To construct a fast adder, we first evaluate the signal delay associated with each type of cell. Given a static CMOS

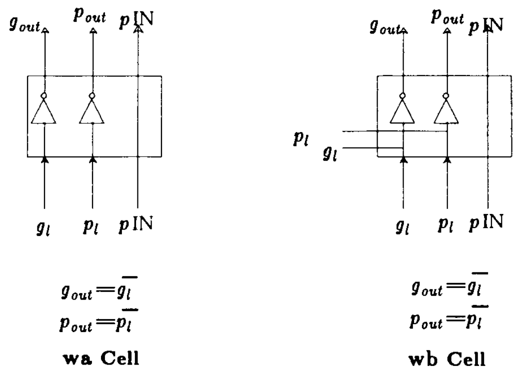


Fig. 4. White cells: *wa* and *wb*.

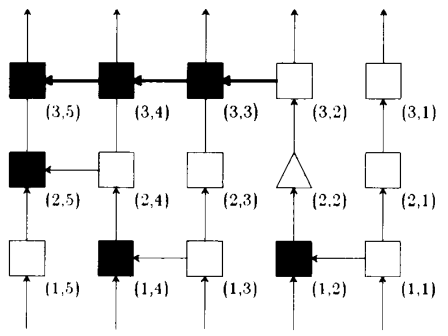


Fig. 5. A 5-bit fast carry generator.

design, we can estimate the cell resistances and capacitances, and compute the associated signal delay. Furthermore, we explicitly model drivers which are an integral part of our circuit layout.

In implementing the black and white cells, we use minimum-length transistors for the pull-down network of each subcell. PMOS pull-up transistors are ratioed so that the maximum (over all possible input conditions) of pull-up and pull-down channel resistances are equal (examples of transistor ratios are shown in Fig. 3). We define this resistance as R_t. For simplicity, we also assume that the capacitance C_i and resistance R_i of the horizontal interconnect between neighboring cells are the same as the R and C values of the vertical interconnect. This condition depends on the specific implementation and is discussed further in Section VI.

In a static CMOS design, a pair of a PMOS pull-up and an NMOS pull-down transistors constitutes a basic inverting unit. The input signal drives the gates of both the pull-up and pull-down transistors. Let C_g be the total gate capacitance of such an inverting unit, then we can approximate the generation time of its output signal, t_{out}, as a function of its interconnect and channel resistances (R_i and R_t, respectively), its load capacitance, its fan-out f, and its input ready time, t_{in}:

$$t_{\text{out}} = t_{\text{in}} + (R_t + R_i f)(C_i + C_g)f.$$

In the case when metal is used for interconnects and $R_t > R_i f$, then t_{out} becomes

$$t_{\text{out}} = t_{\text{in}} + R_i(C_i + C_g)f.$$

Let τ be a normalized time constant, defined as

$$\tau = R_i(C_i + C_g)$$

then

$$t_{\text{out}} = t_{\text{in}} + \tau f \tag{4.1}$$

which is a simple, conventional timing model [13], [14].

Notice that in (4.1), the fan-out f of a subcell is variable, depending on the type (p or g) of the subcell, and on the type and number of its succeeding cells. This can be illustrated using the example of a 5-bit adder shown in Fig. 5. In this circuit, each cell is identified by a pair of height and bit coordinates.

For example, Cell (2,5) refers to the second cell on the vertical path of input bit number 5. It is a black cell. In total, there are six black cells: (1,2), (1,4), (2,5), (3,3), (3,4), (3,5), four *wa* cells: (1,5), (2,1), (2,3), and four *wb* cells: (1,1), (1,3), (2,4), (3,2). Cell (2,2) is the single driver cell.

Now let us consider the black cell at (3,5). Recall that a black cell implements the binary o operation: $(g_l, p_l)o(g_r, p_r) = (g_l + p_l g_r, p_l p_r)$. The "left" operand of Cell (3,5), namely (g_l, p_l), is supplied by Cell (2,5) which precedes Cell (3,5) in a vertical connection. In other words, p_{out} and g_{out} of Cell (2,5) become black cell (3,5)'s p_l and g_l, respectively. The fan-out for p_{out} of Cell (2,5) is 2 as it drives both the p and g subcells of Cell (3,5), and the fan-out is 1 for g_{out} of Cell (2,5) as it drives only the g term of Cell (3,5). We can extend the analysis to a vertical white-cell-to-black-cell connection. Consider Cell (2,4) to Cell (3,4). The fan-outs for the output signals of Cell (2,4) are the same as those previously obtained: $f_{p\text{out}} = 2$, $f_{g\text{out}} = 1$.

The "right" operand of Cell (3,5), namely (g_r, p_r), comes from Cell (2,2) whose output signals make a turn in *wb* Cell (3,2) and continue on to supply the "right" operand to each of black cells (3,3), (3,4), and (3,5). This passing through a horizontal array of black cells is indicated by a bold wire in Fig. 5. The fan-out of g_{out} (or p_{out}) of the driver cell (2,2) is 4, since it drives one subcell in each of (3,2), (3,3), (3,4), and (3,5).

Signal propagation time through a cascaded driver d is a function of the driver's fan-out f_d, the ratio r between successive stages, and the number s of cascaded stages [13]. The minimum delay can be obtained if the driver has the ratio of $r = f_d^{1/s+1}$. This is shown in Fig. 6.

The corresponding minimum propagation delay, *delay* (s, f_d), of the cascaded driver is

$$\mathit{delay}(s, f_d) = (s+1)\left(f_d^{\frac{1}{s+1}}\right)\tau. \tag{4.2}$$

Thus, for an s-stage driver of fan-out f_d,

$$t_{d\text{out}} = t_{d\text{in}} + \mathit{delay}(s, f_d). \tag{4.3}$$

Note that when $s = 0$ and no driver is used, (4.3) is the same as (4.1).

Given the above analysis, we can evaluate the generation time for each circuit signal as the sum of its input ready time

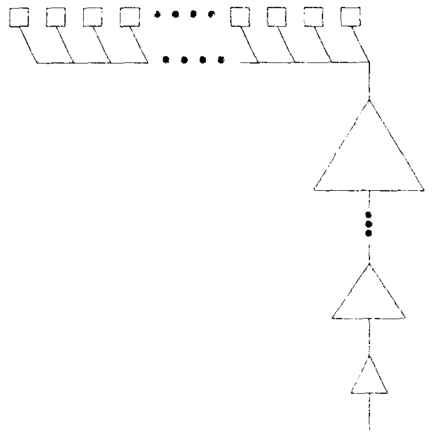


Fig. 6. Multistaged driver.

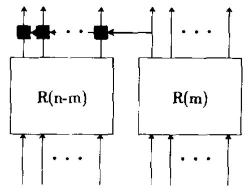


Fig. 7. Recursive construction of an $R(n)$ fast carry generator.

and delay factor. Define $t_{g\text{out}}$ as the time when signal g_{out} is ready, t_{gl} as $t_{g\text{out}}$ of the cell producing g_l, and t_{gr} as $t_{g\text{out}}$ of the cell producing g_r. Similar definitions apply to t_{pl} and t_{pr}. We can then formulate the input ready time for the g subcircuit of a black cell, $t_{g\text{in}}$, as

$$t_{g\text{in}} = \max\{t_{gl}, t_{pl}, t_{gr}\}.$$

And $t_{p\text{in}}$ for the p subcircuit of a black cell becomes

$$t_{p\text{in}} = \max\{t_{pl}, t_{pr}\}.$$

If we define f_g (resp., f_p) to be the fan-out of the subcell under analysis, then

$$t_{g\text{out}} = t_{g\text{ in}} + delay(s, f_g) \tag{4.4}$$

A similar formulation can be written for signal p_{out}:

$$t_{p\text{out}} = t_{p\text{in}} + delay(s, f_p). \tag{4.5}$$

Equations (4.4) and (4.5) parameterize the signal delay with respect to fan-out which is determined by the interconnection of modular cells.

V. Fast Carry Generator

Having evaluated the timing behavior of basic cells, we are facing the problem of how to place and interconnect them into the fastest circuit. Consider the design space of a family of circuits, $R(n)$, shown in Fig. 7. In this figure, black boxes represent the concatenation operation, and are either *ba* or *bb* cells as appropriate.

The construction of $R(n)$ composes blocks of data sizes smaller than n, that is, m and $n-m$ as shown. The $R(m)$ and $R(n-m)$ are, in turn, composed of blocks of even smaller sizes. In other words, we have a recursive construction of $R(n)$. The basis of the construction is $R(1)$ and $R(2)$. $R(1)$ and $R(2)$ are shown in Fig. 8. The $R(n)$ circuit has a large

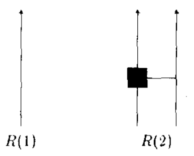


Fig. 8. $R(1)$ and $R(2)$ blocks.

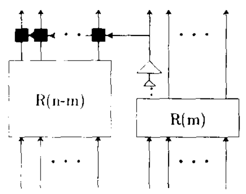


Fig. 9. Drivers used in building an $R(n)$ fast carry generator.

fan-out from the most significant bit of the right block, that is, bit m, broadcasting it to all bit positions of the left block.

Note that, in general, the depth of $R(n-m)$ may be larger than the depth of $R(m)$. This allows us to place a multistaged driver on the output of the most significant bit of $R(m)$. See Fig. 9.

We observe that there are two possible critical paths in $R(n)$. One is the propagation through the leftmost vertical path. The other is the delay through the leftmost bit of $R(m)$, which drives large fan-out and interconnect capacitances. Since the critical paths converge at the leftmost top cell, we focus our analysis on this cell. We can thus compare the adders in $R(n)$ by comparing the times at which their leftmost (p, g) outputs are produced. This "principle of optimality" validates a dynamic programming approach to choosing the best place m at which to decompose an n-bit adder into subcircuits $R(n-m)$ and $R(m)$. Our design problem, then, is to evaluate $t_{g\text{in}}(n, 1)$ in the following recurrence:

$$\begin{aligned} t_{g\text{in}}(i,j) &= \min_{j \le m < i} \{\max[t_{g\text{in}}(i, m+1) + 1\tau, \\ &\qquad t_{p\text{in}}(i, m+1) + 2\tau,\ t_{g\text{in}}(m, j) + f(i, m, j)\tau]\} \\ t_{p\text{in}}(i,j) &= \min_{j \le m < i} \{\max[t_{p\text{in}}(i, m+1) + 2\tau, \\ &\qquad t_{p\text{in}}(m, j) + f(i, m, j)\tau]\} \end{aligned} \tag{5.1}$$

where

$t_{g\text{in}}(i, j)$	is the input ready time for the g term of the most significant bit of an adder block of size $i-j+1$;
$t_{p\text{in}}(i, j)$	is the input ready time for the p term of the most significant bit of an adder block of size $i-j+1$;
$f(i, m, j)$	is the load function of the block of size $m-j+1$ driving that of size $i-m$.

The load function $f(i, m, j)$ is defined as

$$f(i, m, j) = \min_{\substack{0 \le s \le u \\ \text{and } s \equiv u \bmod 2}} \{delay(s, i-m+1)\} \tag{5.2}$$

where

$$u = \max\{0, depth(i, m+1) - depth(m, j)\}.$$

In (5.2), s is chosen to minimize the signal delay through the driver. The depth of the s-stage driver is limited to u, which

TABLE I
OPTIMAL n-BIT BLOCKS, $1 \leq n \leq 32$

n	*left*	*right*	$t_{pin(gin)}(\tau)$	*height*	*driver stages*
1	0	1	1.00	0	0
2	1	1	2.00	1	0
3	1	2	4.00	2	0
4	2	2	5.00	2	0
5	3	2	6.00	3	1
6	4	2	7.00	3	1
7	5	2	8.00	4	2
8	5	3	8.90	4	1
9	6	3	9.29	4	1
10	7	3	10.00	5	2
11	8	3	10.90	5	2
12	8	4	11.24	5	2
13	9	4	11.46	5	2
14	10	4	12.00	6	1
15	10	5	12.67	6	2
16	11	5	12.90	6	2
17	12	5	13.24	6	2
18	13	5	13.46	6	2
19	14	5	14.00	7	1
20	15	5	14.67	7	3
21	16	5	14.90	7	3
22	16	6	15.12	7	3
23	17	6	15.24	7	3
24	18	6	15.46	7	3
25	19	6	16.00	8	2
26	19	7	16.46	8	3
27	20	7	16.67	8	3
28	21	7	16.90	8	3
29	22	7	17.12	8	3
30	23	7	17.24	8	3
31	24	7	17.46	8	3
32	24	8	17.84	8	3

is the depth difference between two component adder blocks, $R(i-m)$ and $R(m-j+1)$. Furthermore, the output polarity of the driver must match that of the left adder block, $R(i-m)$. Thus, the parity of s must be the same as that of u.

Assume that the input signals to the adder, a_i and b_i, are available at time 0. Further, assume that the signals emerging out of the *fast carry generator* have unit fan-out, then the basis for the dynamic programming is

$$t_{\text{gin}}(j, j) = t_{\text{pin}}(j, j) = \tau. \tag{5.3}$$

Lemma 1: The optimal circuit for adding bits $j+r$ through j is identical to the optimal circuit for adding bits $r+1$ through 1. Thus, $t_{\text{pin}}(j+r, j) = t_{\text{pin}}(r+1, 1)$ and $t_{\text{gin}}(j+r, j) = t_{\text{gin}}(r+1, 1)$

Proof sketch: Equations (5.1), (5.2), and (5.3) are not sensitive to the absolute magnitude of i and j. Only the difference $i-j$ is important. ∎

Note that in (5.1), the optimal m minimizing $t_{\text{gin}}(i, j)$ may not be the same as the m minimizing $t_{\text{pin}}(i, j)$. We have to keep a list of optimal: one for p and one for g. However, we prove, in the following lemma, that both p and g signals synchronize upon the p signal. The optimal splitting m value for the p signal is the same as that for the g signal. As a result, we have a one-dimensional dynamic programming problem, optimizing with respect to the generation of the p signal.

Lemma 2: $t_{\text{gin}}(i, j) = t_{\text{pin}}(i, j)$ for $i \geq j$.

Proof: We prove the lemma by induction on i. From (5.3), the basis $j = i$ is trivially true. If $i > j$, and $t_{\text{gin}}(i, j) = t_{\text{pin}}(i, j)$, then from Lemma 1:

$$t_{\text{gin}}(i+1, m+1) = t_{\text{gin}}(i, m)$$
$$t_{\text{pin}}(i+1, m+1) = t_{\text{pin}}(i, m)$$

and by inductive hypothesis,

$$t_{\text{gin}}(i, m) = t_{\text{pin}}(i, m) \qquad \text{for } j \leq m \leq i$$
$$t_{\text{gin}}(m, j) = t_{\text{pin}}(m, j) \qquad \text{for } j \leq m \leq i.$$

From (5.1),

$$t_{\text{gin}}(i+1, j) = \min_{j \leq m < i+1} \{\max [t_{\text{gin}}(i+1, m+1) + 1\tau, t_{\text{pin}}(i+1, m+1) + 2\tau, t_{\text{gin}}(m, j) + f(i+1, m, j)\tau]\}.$$

Since τ is an RC time constant, it must be nonnegative. Thus,

$$t_{\text{gin}}(i+1, j) = \min_{j \leq m < i+1} \{\max [t_{\text{pin}}(i+1, m+1) + 2\tau, t_{\text{gin}}(m, j) + f(i+1, m, j)\tau]\}.$$

Substituting $t_{\text{pin}}(m, j)$ for $t_{\text{gin}}(m, j)$, we have

$$t_{\text{gin}}(i+1, j) = \min_{j \leq m < i+1} \{\max [t_{\text{pin}}(i+1, m+1) + 2\tau, t_{\text{pin}}(m, j) + f(i+1, m, j)\tau]\} = t_{\text{pin}}(i+1, j)$$

hence the lemma. ∎

We can use dynamic programming to calculate the optimal *fast carry generator* configurations for n up to any desired data width. The results are presented in Table I where, for ease of discussion, the data width is limited to 32. Using this table to construct an optimal n-bit *fast carry generator*, the left block should be *left* bits wide and the right block should be *right* bits wide. The column labeled *driver stages* indicates the number of stages in the driver connected to the output of the most significant bit of the right block. The *height* entries indicate the number of rows of modular cells in the optimal n-bit block. The generation time of the most significant bit for the n-bit structure is shown in the column labeled $t_{\text{pin}(\text{gin})}$.

```
initialize the recursion basis of DataWidth = 1;
for DataWidth := 2 to n do
    begin

    initialize the recursion basis of Height = 2 ripple adders;
    for Height := 3 to RCkt[DataWidth]·Height do
        begin
        {find the optimum fast carry generator of (DataWidth,Height)
            in terms of two smaller blocks.}

            HCkt[DataWidth][Height]·Delay = INFINITY;

            for Split := 1 to DataWidth − 1 do
            begin
                LeftDelay = HCkt[Split][Height − 1]·Delay
                    + BlackCellTime;

                {evaluate the right block delay in terms of its
                    datawidth and all possible heights}
                for RHeight := 1 to (Height − 1) do
                begin
                    RightDelay := HCkt[Split][RHeight]·Delay +
                    DelayofSelectedDriver(Fanout,
                    Height − 1 − RHeight);

                    {select the fastest driver which drives
                        the fan-out of Split and fit
                        the area of Height − 1 − RHeight};

                    Delay := MAX(LeftDelay, RightDelay);

                    if(Delay < HCkt[DataWidth][Height]·Delay)
                        {Found a better block than
                        what we had previously;
                        Update the set of values for
                        HCkt[DataWidth][Height]};

            end {for RHeight}
          end {for Split}
        end {for Height}
    end. {for DataWidth}.
```

The time unit τ is a normalized RC time constant. Fig. 10 illustrates the optimal 32-bit *fast carry generator*.

VI. Area and Latency Tradeoff

Using the formulation in Section V, we can find the fastest *fast carry generator* for data width between 1 and n. We also know that the ripple circuit features the smallest area among all blocks. Between these two extremes of speed and area, we can generate a "hybrid" structure called the H circuit. The construction of H structures is identical to that of R circuits shown in Fig. 7. The difference lies in the component blocks. The component blocks for the H circuits can be either R or ripple blocks. The ripple blocks are generated as the base case of minimum area (height = 2) and are shown in Fig. 11.

The best $H(n)$ circuit is characterized by an optimal tradeoff between area and speed. the algorithm for producing the optimal $H(n)$ structure is a two-dimensional dynamic programming. It is presented as follows:

We observe from the above algorithms that we use *BlackCellTime*, *DelayofSelected-Driver*, and *RippleCellTime* (used for the basis) to evaluate the circuit delay. These timing parameters come from our analysis in Section IV.

VII. Layouts

Since our design algorithms are optimization driven, "what if" questions can be answered easily. For example, so far we have assumed that there is no limit to the number of driver

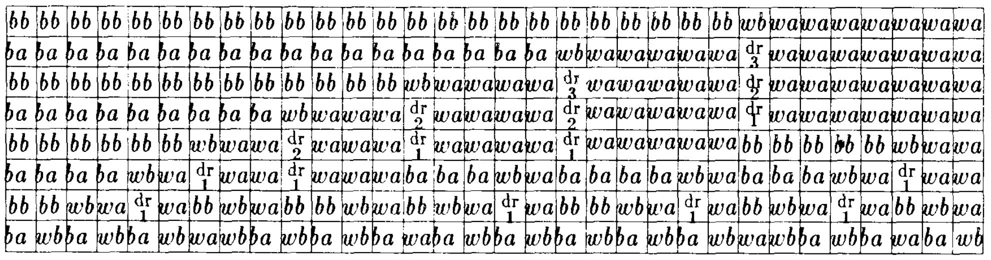


Fig. 10. The optimal 32-bit $R(n)$ fast carry generator.

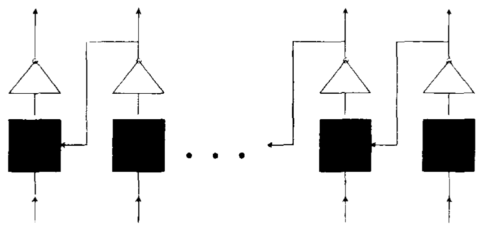


Fig. 11. The ripple block.

TABLE II
OPTIMAL 66-BIT H CIRCUITS

n	*height*	*left*	*left height*	*right*	*right height*	*delay*(τ)	*driver stages*
66	2	0	0	0	0	197	0
66	3	39	2	27	2	120	0
66	4	44	3	22	3	85	0
66	5	46	4	20	3	62.5	1
66	6	47	5	19	4	49	1
66	7	48	6	18	4	38	2
66	8	50	7	16	5	30.75	2
66	9	51	8	15	5	26.25	3
66	10	54	9	12	4	24.875	3
66	11	53	10	13	5	24.375	3
66	12	55	11	11	5	25	3

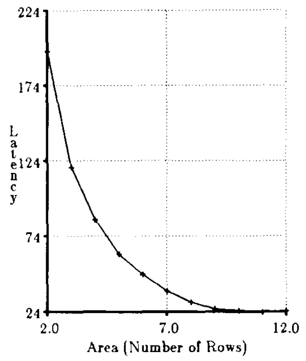


Fig. 12. Area-time tradeoff for 66-bit H circuits.

stages we can use. In practice, we want to get away with the minimum number of driver stages as to minimize layout efforts. We can make the number of stages an input to the algorithm, and evaluate the performance of corresponding design. In addition, the driver ratio between successive stages is usually an integer two or three instead of the "optimal" fractional ratio discussed in Section IV. If this is the case, we can use (4.1) instead of (4.2), where f becomes the driver ratio, to compute the delay through the driver.

Using three-stage drivers with a ratio of two, we have calculated the optimal $H(n)$ circuits for n from 1 to 66. These circuits are indexed by both data width (66) and *height*. For ease of discussion, we tabulate only $H(66)$ results. See Table II and Fig. 12.

Note that the fastest 66-bit *fast carry generator* has *height* 11. Decreasing the height by 4 saves 36% in area, but increases adder delay from 24.375τ to 38τ, a 56% increase. Also note that the possibility of using small ripple blocks makes our $H(n)$ class a superset of the $R(n)$ class.

The 66-bit H circuit of *height* 10, $H(66,10)$, emerges as the best choice in terms of area and latency. It has been implemented in 1.6 μm, double-metal CMOS technology, and is used in the SPUR floating point processor [10]. Its SPICE [15] simulation shows a latency of 21.13 ns, which compared to 24.875τ of Table II results in a τ being 0.85 ns. As indicated in Fig. 1, a complete 66-bit adder can be built from a precondition circuit, an $H(66,10)$ fast carry generator, and a sum circuit. According to SPICE, this 66-bit adder has a latency of 31.5 ns. Its layout area is $4757\lambda \times 395\lambda$ for $H(66,10)$; $4757\lambda \times 473\lambda$-$3806 \times 378$ μm^2-for the complete adder.

VIII. Extensions to Other Technologies and Design Styles

Our discussion has centered on a static CMOS implementation. We can extend the proposed algorithm to NMOS and bipolar technologies. In addition, the algorithm is valid to dynamic circuit design such as domino logic.

It is rather straightforward to adopt the algorithm for an NMOS design. The timing analysis for static NMOS is similar to that of CMOS. The difference is the magnitude of parameters in the timing models. For example, in a static NMOS circuit, the input signals drive the gates of NMOS pull-down network. This is in contrast with static CMOS whose inputs drive both pull-up and pull-down networks. As a result, the C_g value of part of (4.1) is different for NMOS. Nevertheless, since the optimization is based on a normalized time constant τ, changes in τ will not affect the solution determining the interconnection of modular cells.

Signal delay of bipolar TTL parts is insensitive to fan-out and interconnect capacitances, thus eliminating the need for driver circuits. This insensitivity is in contrast with static CMOS/NMOS circuits whose delay is linearly proportional to fan-out and interconnect capacitances (4.1). CMOS/NMOS and TTL circuits have different granularity of implemented logic functions. In static CMOS/NMOS, a single network can perform complex logic functions. For example, the G subcell of a black cell uses a single pull-down and a corresponding pull-up network to implement an AND-OR function (Fig. 3). If the G subcell has a unit fan-out, then based on (4.1) there is a single time-constant (τ) delay through the subcell for all the inputs: p_l, g_r, and g_l. On the other hand, we use two TTL gates to implement the AND-OR function of the G subcell: one for AND; one for OR. Consequently, p_l and g_r have a two-gate delay through the G subcell, and g_l has a one-gate delay. Another difference between CMOS/NMOS and TTL circuits is that the latter has a "hard" fan-out limit whose violation results in a failed circuit.

Using the proposed algorithm with an objective function minimizing the gate delay through the critical path and timing behavior unique to TTL parts, we have calculated optimal TTL *fast carry generator* configurations for n from 1 to 32. A fan-out limit of 10 is used. See Table III.

The design algorithm is also applicable to dynamic circuit implementations such as NORA (No-Race) and domino logic, as long as their timing behavior satisfies the linearity of the recursion step described by (5.1). There may be differences in the magnitude of delay constants and load functions. For example, consider the domino circuit which features an NMOS pull-down network, an NMOS evaluation device, and a PMOS precharging device. The output of this domino stage is followed by a static inverter before it drives another domino stage. Because of this required inverter, there is an extra delay of 1τ between inputs of successive domino stages. Consequently, the delay constants 1τ and 2τ of (5.1) are replaced by 2τ and 3τ, respectively. As for the load function, consider the following. The ratio of the required inverter can be modified to drive a large load. The modified inverter thus becomes a built-in first-stage driver. Any additional driver comes in the form of even stages to preserve the noninverting property of domino logic circuits. These design considerations can be incorporated into the load function, either in the form of $f(i, m, j)\tau$ in (5.1) or DelayofSelected Driver( ) of the design routine in Section VI.

IX. Discussions and Conclusions

We have formulated the adder design as a dynamic programming problem with latency and/or area as the performance variables to be optimized. We illustrated our design algorithm with an example of CMOS VLSI implementations. The algorithm incorporates factors crucial to VLSI design: modularity, routing, speed, area, and driver design. It takes as inputs timing parameters associated with component modules, and generates an area-time optimal adder as the output. Not limited to static CMOS implementation, our formulation has been shown to be general enough to cover a wide range of implementation technologies: MOS and TTL, as well as design styles: dynamic and static circuits.

A natural extension to our formulation is to include increased cell "fan-in" and to use a more elaborate timing model. The first issue arises from the fact that the black cells we have used feature a cell "fan-in" of two as they implement the didactic associative operator o. An increase in their "fan-in" may result in a reduced number of logic levels and corresponding improvement of circuit latency. Nevertheless, the advantage of increased "fan-in" has to be addressed in the context of a specific implementation technology and circuit style.

The second issue relates to the timing model we have used to introduce our dynamic programming formulation. There we have assumed negligible parasitic capacitances and used a simplistic τ model. We can use a more sophisticated timing model as long as the adder latency is monotonically nonincreasing with respect to the data width, thus satisfying the "principle of optimality" of a dynamic programming formulation.

Our structured approach lends itself to design automation. The solution to our algorithm, specifying an arrangement of modular tiles (Fig. 10), can feed an automatic VLSI layout tool such as MQUILT [16]. MQUILT, along with user-supplied modular tiles, generates the adder layout in MAGIC [17] format, and this is how we obtained our layout discussed in Section VII.

TABLE III
OPTIMAL n-BIT TTL ADDERS WITH FAN-OUT = 10, $1 \leq n \leq 32$

data width	left	right	delay(τ)
1	0	0	0
2	1	1	2
3	2	1	3
4	2	2	4
5	3	2	4
6	3	3	5
7	4	3	5
8	5	3	5
9	4	5	6
10	5	5	6
11	6	5	6
12	7	5	6
13	8	5	6
14	7	7	7
15	8	7	7
16	9	7	7
17	10	7	7
18	10	8	7
19	10	9	8
20	10	10	8
21	10	11	8
22	10	12	8
23	10	13	8
24	10	14	9
25	10	15	9
26	10	16	9
27	9	18	9
28	10	18	9
29	6	23	10
30	7	23	10
31	8	23	10
32	9	23	10

We have proposed a realistic and practical approach to the design of optimal adders. The implementation result shows our approach is extremely competitive to alternative implementation schemes such as variable-block carry-skip adders [18], [19]. This is especially true for large data width additions.

REFERENCES

[1] A. M. Despain, "Notes on computer architecture for high performance," in *New Computer Architectures,* J. Tiberghien, Ed. London, England: Academic, 1984.
[2] D. J. Kuck, *The Structure of Computers and Computation,* vol. I, 1978, pp. 54-56.
[3] F. E. Fich, "New bounds for parallel prefix circuits," in *Proc. 15th ACM Symp. Theory Comput.*, Apr. 1983, pp. 100-109.
[4] H. C. Lai and S. Muroga, "Minimum parallel binary adders with NOR(NAND) gates," *IEEE Trans. Compat.*, vol. C-28, no. 9, 1979.
[5] B. W. Y. Wei and Y.-F. Chen, "QAC: A CMOS implementation of the 32-bit *Q* adder," in *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput.*, Port Chester, NY, Oct. 1985.
[6] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, Mar. 1982.

[7] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proc. 8th Symp. Comput. Arithmetic,* May 1987, pp. 49-55.

[8] T.-F. Ngai, M. J. Irwin, and S. Rawat, "Regular, area-time efficient carry-lookahead adders," *J. Parallel Distributed Comput.,* 1986, pp. 92-105.

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

[10] T. Hu, "Circuit design techniques for a floating point unit," Rep. UCB/CSD 87/372, Comput. Sci. Division, U.C. Berkeley, Sept. 1987.

[11] B. K. Bose, L. Pei, G. S. Taylor, and D. A. Patterson, "Fast multiply and divide for a VLSI floating point unit," in *Proc. 8th Symp. Comput. Arithmetic,* May 1987, pp. 87-94.

[12] M. Hill *et al.*, "Design decisions in SPUR," *IEEE Comput. Mag.,* vol. 19, no. 10, pp. 8-22, Nov. 1986.

[13] C. Mead and L. Conway, *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley Series in Computer Science, 1980.

[14] P. C. Flake, G. Musgrave, and M. Shortland, "The HILO logic simulation language," in *Int. Symp. Comput. Hardware Description Languages and Their Appl.,* IEEE, New York, 1975, pp. 134-142.

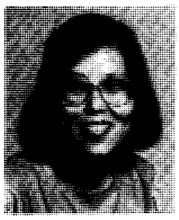
[15] A. Vladimirescu *et al.*, "SPICE version 2G user's guide," CS Division, EECS Dep., Berkeley, Aug. 1981.

[16] R. N. Mayo and W. S. Scott, "Berkeley VLSI tools manual," C.S. Division, EECS Dep., Berkeley, 1983.

[17] J. K. Ousterhout *et al.,* "A collection of papers on magic," UCB CSD Tech. Rep., Sept. 1983.

[18] V. G. Oklobdzjia and E. R. Barnes, "Some optimal schemes for ALU implementation in VLSI technology," in *Proc. 7th Symp. Comput. Arithmetic,* June 1985, pp. 2-8.

[19] A. Guyot, B. Hochet, and J. M. Muller, "A way to build efficient carry-skip adders," *IEEE Trans. Comput.,* vol. C-36, no. 10, Oct. 1987.



Belle W. Y. Wei (S'82-M'87) received the A.B. degree form the University of California, Berkeley in 1977, the M.S. degree from Harvard in 1980, and the Ph.D. degree in electrical engineering from U.C. Berkeley in 1987.

She is currently an Assistant Professor in the Department of Electrical Engineering, San Jose State University, San Jose, CA. Her research interests include VLSI theories, computer arithmetic circuits, and asynchronous circuit design.

Clark D. Thomborson (a.k.a. **Thompson**) received the B.S. degree in chemistry and the M.S. degree in computer science/computer engineering from Stanford University, Stanford, CA, in 1975 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1980. Upon marriage to Barbara Borske in 1983, Clark merged last names with his wife, becoming a Thomborson.

Clark Thomborson is now an Associate Professor of both Computer Science and Computer Engineering at University Minnesota, Duluth. His teaching and research interests center on the algorithmic aspects of computer hardware design.