JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 5, 716–728 (1988)

On Implementing Addition in VLSI Technology

VOJIN G. OKLOBDZIJA AND EARL R. BARNES

IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598

Received July 30, 1987

In this paper we discuss the rules for evaluation of arithmetic algorithms based on the speed of their VLSI implementations. We present the rules which are simple enough to be useful for quick estimates, but yet reflect basic dependencies. By applying these rules we derived a simple scheme for VLSI implementation of addition (ALU), with a near minimal number of gates and small and regular area. Despite its simplicity, this scheme outperforms carry-lookahead and recurrence solver schemes as demonstrated by simulation of the actual implementation of examples. This is because the properties of the scheme are based on the dependencies and assumptions reflecting the real conditions existing in VLSI-CMOS technology. We discuss these results and demonstrate by actual implementation of examples that the measures based on the number of logic levels are not applicable to the new VLSI technologies. © 1988 Academic Press, Inc.

1. INTRODUCTION

In many implementations of a single-chip VLSI processor, the adder (ALU) is found to be in the critical path of the machine, therefore determining the machine cycle. Since the duration of the machine cycle is directly related to the performance of the machine, the speed of the ALU (adder) is critical in achieving higher performance.

In the past, much work was done in developing fast schemes for addition, and many different schemes and their implementations were created [1]. Their development was guided by the rules of the technology used in those days (TTL, ECL) and many schemes were developed approaching the limit of the attainable speed [2].

Several papers were published, describing adders based on a recurrence solving scheme emphasizing their feasibility for VLSI implementation [11–16]. Recently a study of VLSI-oriented schemes was undertaken which implied that CLA is the scheme yielding the fastest implementation. This conclusion was supported by partially simulated sections in nMOS technology [9]. A

0743-7315/88 $3.00

similar study of "recurrence solvers" [11] was done by Han and Carlson [17, 18]. They developed a scheme which they refer to as "hybrid prefix computation" (HPC). They claim this scheme to be faster than all previously reported "recurrence solvers" [11–16] including CLA and our scheme [5, 6] based on variable sized blocks, "variable block adder" (VBA), obtained by optimizing the carry path. Their claim is supported by partial simulation using SPICE as it was done in [9].

During the course of the work reported in this paper a "brute force" approach was applied—we simply implemented those representative schemes, for the typical sizes of 16 and 32 bits [1, 5, 17, 18], and compared the speeds obtained. For implementation we considered 1.5-μm CMOS-ASIC technology coupled to a simulator developed for that purpose [21, 22]. The timing was obtained through an elaborate simulation of the entire design taking into account all the loading, including wiring. The results contradicted many of the previous findings [9, 17, 18]. Even though the recurrence solver schemes are very elaborate, they show no gain (and even a loss) in performance due to larger fan-ins (FI) and fan-outs (FO). Our implementation (VBA) turned out to be faster than "recurrence solvers" and CLA for a 32-bit implementation and a close second for a 16-bit implementation. Such an outcome was not unexpected, because we have claimed that the measures traditionally applied to the technologies, such as TTL, are not adequate for VLSI technologies, CMOS in particular [5, 6–8]. This claim has been somewhat confirmed by the practical wisdom of the industry, which does not use any of the more elaborate schemes in the microprocessors that can be found on the market. In this paper we attempt to explain the rationale. We discuss the features of the technology used to implement our scheme, describe it, and compare it with the results obtained from the actual implementations of other schemes.

2. Estimating Delay

In this discussion we used CMOS-ASIC vendor technology [21]. Even though data are derived from an ASIC-CMOS standard cell library [21], it can be fairly well generalized to the other CMOS technologies. Basic dependencies discussed here are valid even for custom design. The ASIC library consists of a collection of gates and cells designed for high speed and low gate count. There are two versions of each cell:

1. a fast version with high gate count using more power (FG)
2. a standard version designed for small gate count (SG).

Their purpose is to use the fast version where it is critical to minimize the delay. Otherwise the standard version is used to minimize the gate count and power.

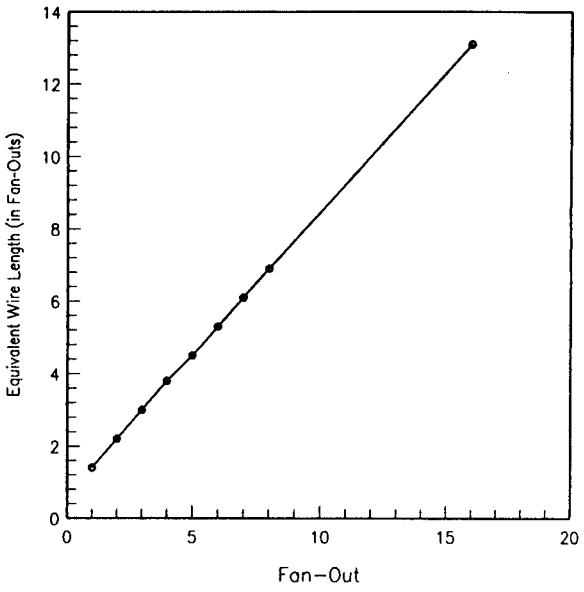

FIG. 2. Effect of wire load on delay (for 3 × 3-mm area).

is neglected by the delay estimates based on counting levels of logic. One of the approaches to alleviating the delay caused by nodes with very high fan-outs is to use gates with higher driving capabilities [10]. This helps to improve the situation somewhat, although it does increase the loading of the gates in the preceding level due to the higher loading that the input of a FG represents.

Fan-in. Forcing a reduction in the number of logic levels may also result in an increase in fan-in. This increase is reflected in the gate delay since fan-in is directly proportional (equal) to the number of transistors connected in series in an n-type transistor network (NAND gate) or p-type transistor network (NOR gate). The gate delay vs fan-in dependency for a NOR gate is shown in Fig. 3. To set the logic state in CMOS, a node must be connected to one of the terminal nodes (Gnd, Vdd) by a string of transistors equal in length to the FI. In case of a NAND gate, these transistors are of n-type (connected to ground node). In case of a NOR gate, these transistors are of p-type (connected to Vdd). Given that a p-type transistor (whose major carriers are holes) is almost twice as slow as an n-type transistor, a NOR gate is slower than an equivalent NAND gate (taking the worse of the two times, t_f, t_r, into account). The opposite is true for bipolar technology where a wired-OR implementation is used to obtain a major speed advantage factor. Also, it can be observed that in the case of a NOR gate, the rise time t_r is longer than the fall time t_f, contrary to the NAND gate where the opposite is the case. This is to be expected from the previous observation regarding the transistor paths.

Since the time required to change logic states is directly proportional to the number of transistors in the path between the terminal nodes (worst case), which is equal to the FI, the larger the fan-in, the slower the gate.

2.1. *Delay Dependency*

Commonly used speed estimates take into account the number of logic levels only [1]. However, the speed of CMOS technology shows a dependency on fan-out loading and fan-in. Wiring delays are attributed to the loading of gate outputs and signal propagation delays. The first is expressed in terms of additional fan-out loads and added to the gate output. The second is added to the gate delay.

Fan-out. CMOS gates exhibit an almost linear dependency on fan-out loading. Figure 1 shows delay of a NAND gate as a function of fan-out load. For a NOR gate, we observe a similar increase (increasing the fan-out from 2 to 8 results in approximately tripling the gate delay). An inverter delay resembles similar dependency on fan-out.

The average delay due to wire capacitance is also linearly dependent on the fan-out and its effect is in increasing linear dependency of gate delay on the fan-out. The average amount of wiring is proportional to the fan-out and it is expressed in terms of additional fan-out loading, Fig. 2. The average wire length (expressed in terms of fan-out) is a function of chip size. Both functions exhibit a linear relationship (for chips of moderate area) and the wiring is reflected as additional fan-out in the gate delay calculation.

The relationship between gate delay and fan-out can be approximated fairly well by a linear function. Increasing the fan-out results in more delay which

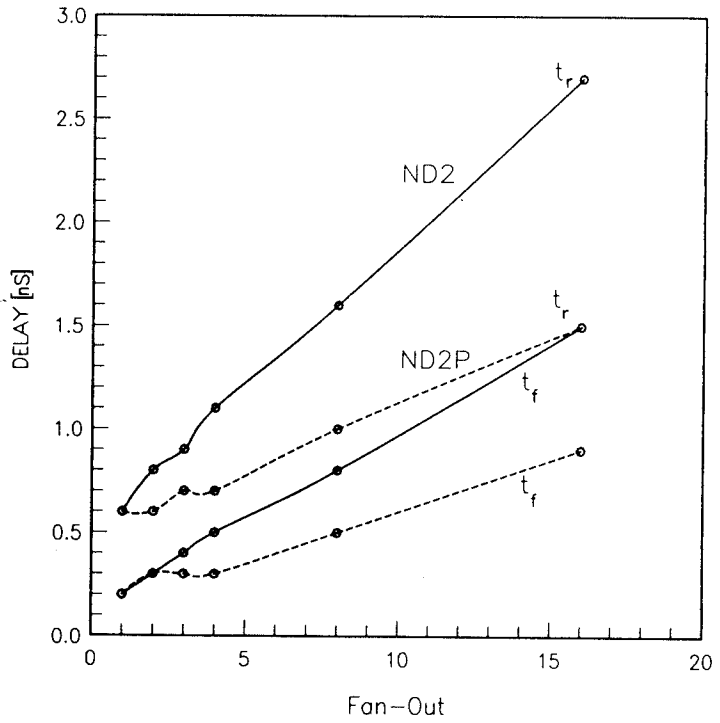

FIG. 1. NAND gate delay vs fan-out: ND2P is the powered version of a NAND gate. t_r, rise time; t_f, fall time.

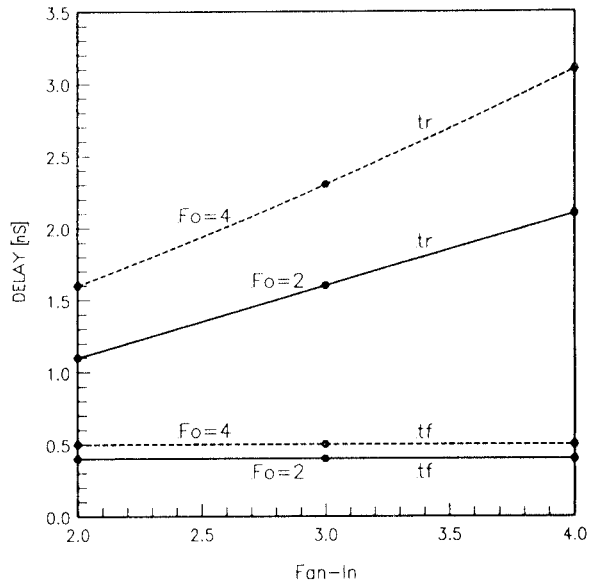

FIG. 3. Gate delay as a function of fan-in. t_r, rise time; t_f, fall time.

This simple analysis also challenges the validity of just using the number of logic levels as an adequate estimate of the speed of an implementation as has been traditionally used in development of fast computer arithmetic. However, estimates based on logic levels are still adequate for bipolar technology and other technologies where the driving capabilities of gates are such that the influence of fan-out and wire loading on speed is negligible. With CMOS becoming the dominant VLSI technology, estimates based solely on the number of logic levels are no longer valid. In summary, implementing functions in a minimal number of logic levels does not necessarily yield the fastest implementation.

2.2. *Delay Estimates*

Here we propose simple formulas for estimating the speed of a CMOS implementation. They are derived from the delay tables for the particular ASIC technology [21] and simplified to make them easier to use. All delays are normalized to delay unit of 1 (delay of an inverter). The formulas are simple to use yet they reflect the real parameters affecting the delay of a CMOS gate. As estimates of the speed, they can be used generally for other CMOS designs including custom. The parameters used to estimate the delay δ are Fo, fan-out of a given gate; (Fi − 2), fan-in in excess of 2.

The formulas for calculating estimates for NAND gate, NOR gate, and inverter are

$$\delta_{\text{NAND}} = 1 + 0.3\text{Fo} + 0.5(\text{Fi} - 2)$$

$$\delta_{\text{NOR}} = 1 + 0.5\text{Fo} + 0.5(\text{Fi} - 2)$$

$$\delta_{\text{INV}} = 0.7 + 0.3\text{Fo}.$$

We found that these estimates adequately model the main contributions to the delay. In addition these equations reflect the facts that

- in terms of a delay, NAND gate is slightly less sensitive to fan-out loading than NOR gate and
- an inverter is faster than NAND and NOR gates.

For the fast version of the gates (FG) which uses a more powerful driver to drive a large fan-out load, we similarly obtained

$$\delta_{\text{NAND-F}} = 1 + 0.125\text{Fo} + 0.5(\text{Fi} - 2)$$

$$\delta_{\text{NOR-F}} = 1 + 0.25\text{Fo} + 0.5(\text{Fi} - 2)$$

$$\delta_{\text{INV-F}} = 0.3 + 0.125\text{Fo}.$$

We used these equations to obtain some meaningful comparison of the alternative implementations. This resulted in much better estimates than those obtained using logic levels and the agreement with measured speeds was much closer (as shown in Fig. 7). These relations are used as a base for derivation of a new scheme.

In the next section a suggested scheme for ALU implementation is described. We compare it to the results of actual CMOS implementation of other schemes and present results supporting our claims about speed.

3. Suggested Scheme

Our scheme is based on using variable sizes of carry blocks in the carry chain implemented as carry-skip adder [3, 4]. The sizes we use are chosen to optimize the speed of the carry path. We refer to our scheme as VBA (variable block addition) [5, 6–8].

3.1. *Derivation of the Scheme*

For a 32-bit adder, and the VBA scheme, we divided the carry chain into blocks of sizes 1, 3, 5, 7, 7, 5, 3, 1. We will now explain why this division is optimal. Let t denote the time required for a carry signal to ripple across a bit in the carry chain, and let T denote the time required for the signal to skip over a group of bits. By simulation of the blocks, we have found that $t = 0.8$ ns and $T = 1.6$ ns. To simplify our analysis, we normalize them so that $t = 1$ and $T = 2$. Then we apply the theory developed in [5] for finding the optimal division of a carry chain.

Let m denote the optimal number of groups for an n-bit carry chain. By Lemma 1 in [5], m is the smallest positive integer satisfying

$$n \leq m + 1/2mT + 1/4m^2T + (1 - (-1)^m)T/8.$$

Given m, an optimal division of the carry chain into groups can be obtained as follows. Let

$$y_i = \min\{1 + iT, 1 + (m + 1 - i)T\}, \qquad i = 1, \ldots, m.$$

Given $y_1, \ldots, y_m$, solve the minimization problem

$$\min_x \max\{x_1, \ldots, x_n\}$$

subject to

$$0 \leq x_i \leq y_i, \qquad i = 1, \ldots, m,$$

and

$$\sum_{i=1}^{m} x_i = n.$$

Any solution $x_1, \ldots, x_m$ gives optimal group sizes for a division of the carry chain.

The x's can be computed iteratively as follows: Initially take $x_1 = \cdots = x_m = 0$. At each iteration, increase as many of the x's as possible by one unit, without violating the constraints $0 \leq x_i \leq y_i$, $i = 1, \ldots, m$, $\sum_{i=1}^{m} x_i \leq n$. An easy calculation shows that

$$\sum_{i=1}^{m} y_i = m + 1/2mT + 1/4m^2T + (1 - (-1)^m)T/8 \geq n.$$

Thus, at some iteration, we have $\sum_{i=1}^{m} x_i = n$ and the algorithm terminates.

For $n = 32$ we have $m = 7$, $y_1 = 3$, $y_2 = 5$, $y_3 = 7$, $y_4 = 9$, $y_5 = 7$, $y_6 = 5$, $y_7 = 3$. The above algorithm gives $x_1 = 3$, $x_2 = 5$, $x_3 = 5$, $x_4 = 6$, $x_5 = 5$, $x_6 = 5$, $x_7 = 3$. A carry chain divided in this way has maximum delay $\Delta = mT = 14$. Since one unit of delay in our system is 0.8 ns, the maximum delay for our 32-bit carry chain is $\Delta = 14 \times 0.8$ ns $= 11.2$ ns. This time involves only the delay in the carry chain. It is easy to check that this is also the delay for a chain divided into groups of sizes 1, 3, 5, 7, 7, 5, 3, 1. Thus, this is also an optimal subdivision.

The worst case delay includes the time needed to generate p_i and g_i signals, delay of the carry chain, and the time for producing last sum bit s_n.

3.2. *Implementation*

The implementation of our scheme will be described for a 32-bit adder (easily modified to an ALU). The size of the blocks varies with the size of the adder and is different in the 16-bit case. The critical path is the carry signal

path. We implemented it with a string of multiplexers, as shown in Fig. 4, taking advantage of the fact that the multiplexer cell is designed to be very fast. Another reason for using multiplexers is that they are designed as very fast structures using buffered pass gates and in this sense are similar to the "Manchester carry chain" [3], [23] which has been shown to be the most effective implementation of a carry chain [5, 23].

The implementation of a single carry block is done by mixing a 4 to 1 multiplexer (actually used as a 3 to 1) in the last stage with a string of 2 to 1 multiplexers. A carry bypass is connected to inputs 3 and 4 of the 4:1 multiplexer (group carry multiplexer) and the selection of the carry bypass is activated by the NAND gate signaling when the condition for "group propagate" is reached and activating the group multiplexer in turn. It is shown in Fig. 4.

The 32-bit implementation of the VBA adder is obtained by connecting the groups of the sizes calculated in Section 3.1 for the full length of $n = 32$ bits. To increase the speed further, we used a faster inverting version of the multiplexer, alternating between C_i and $\bar{C}_i$ signals.

Design was simulated for a number of input combinations (test cases), each involving a potentially critical path. The simulator used was LDS-III [22], which takes into account the average wire length, loading imposed by input transistors, intrinsic gate delay, and rise/fall time. Experience with a very large number of cases has confirmed the accuracy of the simulator.

For our 32-bit implementation the worst case obtained had a delay of $t_{cr} = 14.2$ ns.

4. Comparison with Implementations of Representative Schemes

Given that the speed of a particular scheme can be judged only by its implementation, we decided to measure the speed of the best possible implementations of their representatives. In the course of designing them, we took great care that the implementations are truly the best possible for a given technology and cell library.

A preliminary pass through all the blocks was applied to identify heavily loaded nodes (larger fan-outs or driving longer wires). At these nodes, gates with stronger driving capabilities were used. Also, in every block a combination of gates (NAND–NAND, NAND–NOR, etc.) yielding the fastest implementation of a given function was applied. This was estimated from the gate delay vs fan-out load tables [21] for each gate. After the network compilation and first simulation pass, all the rise/fall times on all the nodes were checked, and those with rise/fall times longer than 3.0 ns were flagged [22]. At this point, the design was sped-up by introducing the fast version gates and splitting the fan-outs into separately driven logic.

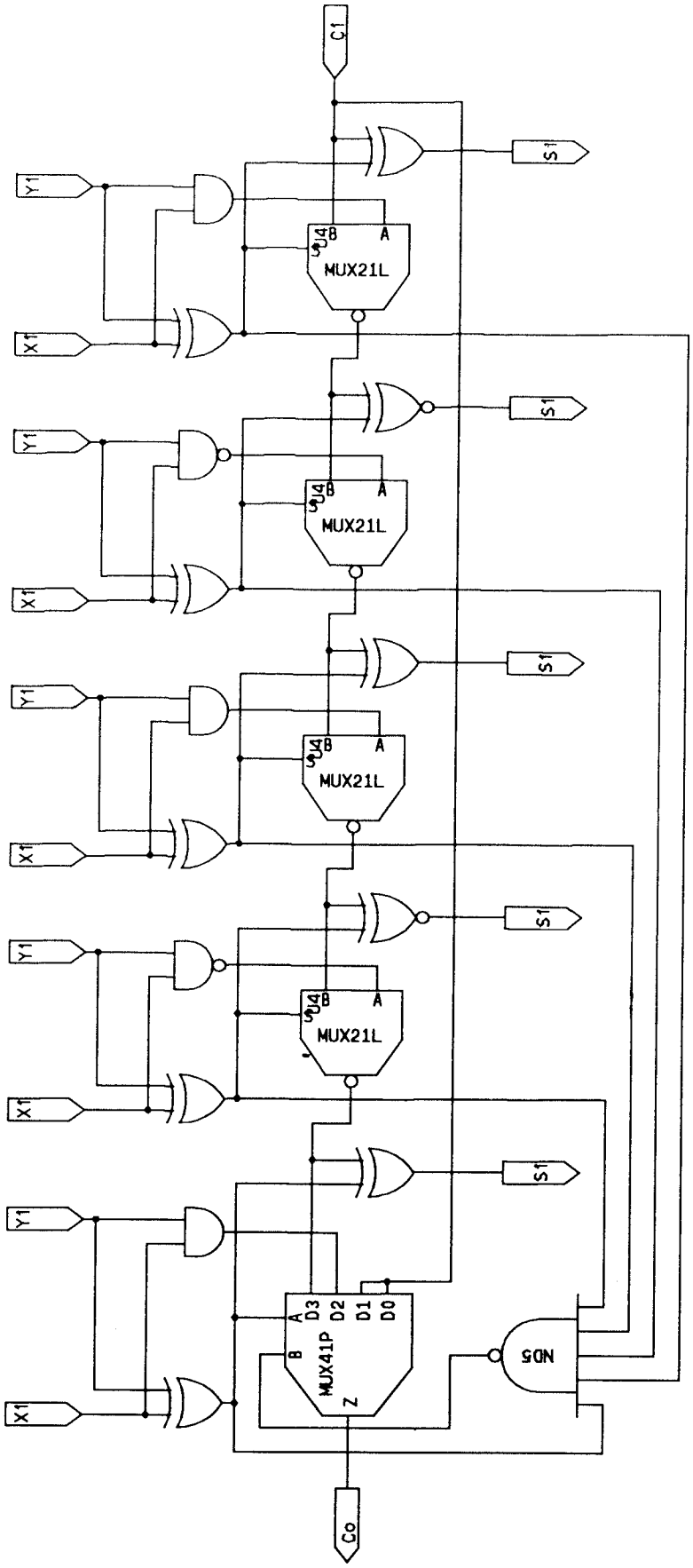

FIG. 4. Implementation of the carry path.

In the final pass, the critical path was examined, replacing gates with their faster version, FG, where applicable, and reducing the loading. After several iterations through the design and simulator this process converged. The results reported were obtained by running the entire designs through a very accurate simulator [22].

We decided on implementing the following schemes:

1. ripple carry (RCA)
2. carry look-ahead (CLA)
3. recurrence solver (HPA)
4. variable block carry (VPA).

All of the examples were implemented for two sizes, 16 and 32 bits, taking care that each implementation was optimized for speed. In the case of the HPA we implemented the 16-bit scheme reported in [17] (claimed to be the fastest compared to other 16-bit implementations), while for 32 bits we used the 32-bit implementation reported in [18], by the same authors. It is different than their 16-bit scheme. Their 16-bit scheme has fan-out restriction to FO = 2, while the 32-bit scheme has no fan-out restriction.

In order to ensure that a given scheme was implemented in the best possible way, we made several iterations through simulation and back through design. The fast version FG (power gate) was applied at any node where the substitution yielded a performance improvement over the use of a normal version SG. This substitution procedure eliminated the possibility of overloading the design by using too many of the FGs. In addition, we used the combination of gates (NAND–NOR) which resulted in the fastest implementation. This was done in accordance with the measures described in Section 2 of this paper.

4.1. *Results*

The respective speed comparison obtained for the selected schemes is shown in Fig. 5.

The complexity of each design is measured as a "gate count," which is a number of equivalent two-input gates (one gate consists of four transistors). The gate count for the implementation of each representative scheme is shown in Fig. 6.

From Figs. 5 and 6 we can observe that the implementation of the VBA scheme yields the fastest adder for the size of 32 bits and it is second best in terms of complexity measured in the number of gates used. In terms of complexity, measured by the equivalent two-input gates used, HPC uses 458 equivalent gates versus 348 used by VGA. That means that 31% more gates are used to achieve no advantage at all. It would be reasonable to expect that if wiring complexity is added to this complexity measure, the VBA imple-

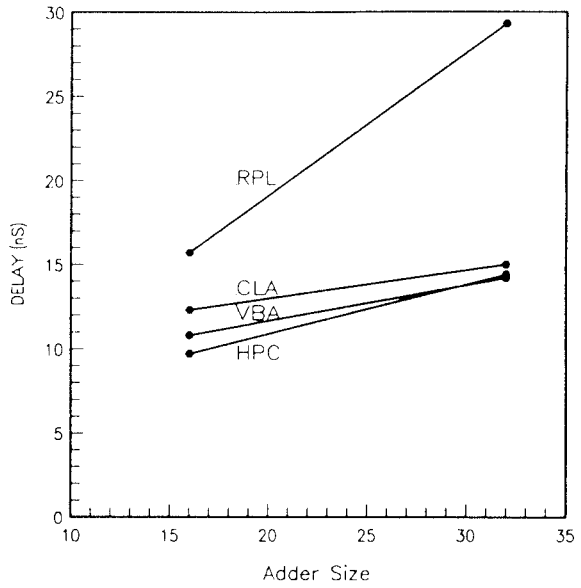

FIG. 5. Speed of implemented schemes. Delay of various adders vs size.

mentation would gain even more of an advantage given its regular structure and relatively low fan-outs.

4.2. *Comparing Estimates*

We estimated the speed of each scheme by using the "number of logic levels" estimate (columns 1, 2 in Fig. 7). Based on this estimate for the size of 32 bits, the HPC scheme seems to be the fastest as concluded in [18], while in practice it was a close second to VBA. It should be noted that the 32-bit HPC scheme has no fan-out restriction and therefore has fewer levels of logic

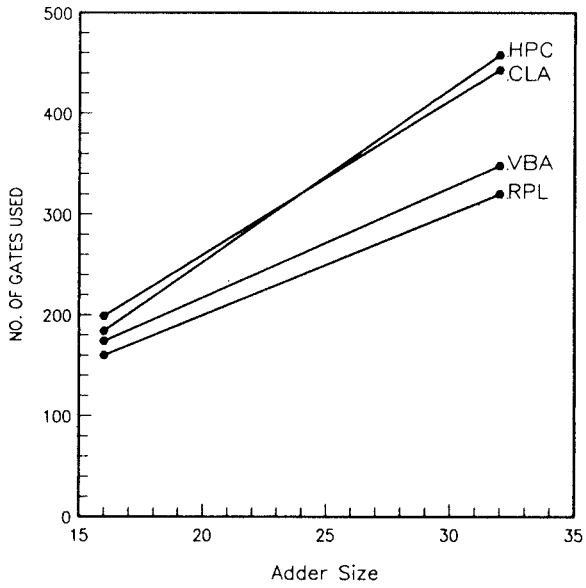

FIG. 6. Complexity of the implementation (various adders vs size).

Delay	Estimated by the # of Logic Levels		Estimated using the new meassures		Simulated	
Adder type/size	16	32	16	32	16	32
RPL	16	32	20	40	15.7	29.3
CLA	8	11	10.9	16.9	12.3	15.0
HPC	13	10	10.2	15.5	9.7	14.4
VBA	10	14	9.7	12.2	10.8	14.2

FIG. 7. Estimated and simulated delay of various adder implementations.

than its 16-bit version. Therefore this estimate also shows 32-bit HPC to be even faster than the 16-bit one.

Finally, we applied the formulas developed in Section 2 to compare the speeds of the test schemes. The results obtained are presented in columns 3 and 4 of Fig. 7. The delay estimates, obtained by applying estimates proposed in Section 2.2 (rather than "levels of logic"), are showing much closer resemblance to the delays actually obtained. This is true in terms of both relative speed and ranking.

5. CONCLUSION

In this paper we argue that fundamental measures (e.g., number "levels of logic") used in the course of much of the development of computer arithmetic schemes and their hardware implementations are not valid when applied to new VLSI technologies (such as CMOS). Therefore it would be a mistake to simply map the designs of these earlier schemes directly into their VLSI-chip implementations.

The measures of speed need reevaluation and new measures are proposed in this paper. Our measures reflect basic speed dependencies and therefore represent more realistic measures for estimating the speed and efficiency of a particular algorithm in terms of its VLSI implementation. We feel that the arithmetic algorithms, developed historically, should be used judiciously and reevaluated prior to their implementation rather than being mapped directly into VLSI hardware. The way in which they were evaluated and ranked in the past may no longer be valid and other algorithms, sometimes overlooked and neglected, might yield an implementation with superior performance.

By examples, we have shown that some schemes, thought and claimed to be superior, actually fall short. Based on theoretical estimates, we claimed in [5] that VBA would be slightly slower than CLA. The examples showed VBA implementation to be actually faster which confirms the validity of our arguments.

Finally, the practical wisdom of this analysis would be to recommend simplicity and regularity in the course of VLSI design.

REFERENCES

1. Hwang, Kai. *Computer Arithmetic: Principles Architecture and Design.* Wiley, New York, 1979.
2. Winograd, S. On the time required to perform addition. *J. Assoc. Comput. Math.* **12,** 2 (1967), 277–285.
3. Kilburn, T., Edwards, D. B. G., and Aspinall, D. Parallel addition in digital computers: A new fast "carry" circuit. *Proc. IEE,* Vol. 106, Pt.B. P.464, Sept. 1959.
4. Lehman, M., and Burla, N. Skip techniques for high-speed carry-propagation in binary arithmetic units. *IRE Transactions on Electronic Computers,* Dec. 1961, p. 691.
5. Oklobdzija, V. G., and Barnes, E. R. Some optimal schemes for ALU implementation in VLSI technology. *Proc. of 7th Symposium on Computer Arithmetic,* University of Illinois, Urbana, IL, June 4–6, 1985.
6. Barnes, E. R., and Oklobdzija, V. G. *Method for Fast Carry-Propagation for VLSI Adders.* Submitted March 22, 1983, IBM Technical Disclosure Bulletin, Vol. 28, No. 4, Sept. 1985.
7. Barnes, E. R., and Oklobdzija, V. G. *New Scheme for VLSI Implementation of Fast ALU.* Submitted Jan. 24, 1984, IBM Technical Disclosure Bulletin, Vol. 28, No. 3, Aug. 1985.
8. Barnes, E. R., and Oklobdzija, V. G. *New Multilevel Scheme for Fast Carry-Skip Addition.* Submitted March 22, 1984, IBM Technical Disclosure Bulletin, Vol. 27, No. 11, Apr. 1985.
9. Ong, S., and Atkins, D. E. A comparison of ALU structures for VLSI technology. *Proc. of the 6th Symposium on Computer Arithmetic,* Aarhus University, Aarhus, Denmark, June 20–22, 1983.
10. Montoye, R. K., and Cook, P. W. Automatically generated area, power, and delay optimized ALUs. *IEEE ISSCC Digest of Technical Papers,* New York, February 23–24, 1983.
11. Kuck, David J. *The Structure of Computers and Computation.* Wiley, New York, 1978.
12. Bilgory, A., and Gajski, D. D. Automatic generation of cells for recurrence structures. *Proc. of 18th Design Automation Conference,* Nashville, TN, 1981.
13. Bilgory, A., and Gajski, D. D. A heuristic for suffix solutions. *IEEE Trans. Comput.* **C-35,** 1 (Jan. 1986).
14. Brent, R. P., and Kung, H. T. A regular layout for parallel adders. *IEEE Trans. Comput.* **C-31,** 3 (March 1982).
15. Kogge, P. M., and Stone, H. S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.* **C-22,** 8 (Aug. 1973).
16. Ngai, T. F., Irwin, M. J. Regular, area-time efficient carry-lookahead adders. *Proc. of 7th Symposium on Computer Arithmetic,* University of Illinois, Urbana, IL, June 4–6, 1985.
17. Han, T., and Carlson, D. A. Fast area-efficient VLSI adders. *Proc. of 8th Symposium on Computer Arithmetic,* Como, Italy, May 19–21, 1987.
18. Han, T., Carlson, D. A., and Levitan, S. P. VLSI design of high-speed low-area addition circuitry. *Proc. of ICCD,* Rye, New York, October 5–8, 1987.
19. Han, T. *Theory of Hybrid Prefix Algorithm.* Ph.D. proposal, Feb. 1987.
20. Han, T. Private communications, Feb. 1987.
21. *Compacted Array Product Databook.* LSI Logic Corp., Feb. 1987.
22. *LDS Software Guidelines.* LSI Logic Corp., 1985, 1986.
23. Mead, C., and Conway, L. *Introduction to VLSI Systems.* Addison–Wesley, Reading, MA, 1980.