High-Speed Addition in CMOS

Nhon T. Quach and Michael J. Flynn

Abstract—**This paper describes a fully static Complementary Metal-Oxide Semiconductor (CMOS) implementation of a Ling type 32-bit adder. The implementation described herein saves up to one gate delay and always reduces the number of serial transistors in the worst-case (critical) path over the conventional carry look-ahead (CLA) approach with a negligible increase in hardware.**

Index Terms— **Carry look-ahead, conditional-sum adders, group-generate, high-speed CMOS binary adder, modified Ling addition, multiple output Domino logic.**

I. Introduction

For high-speed addition, Ling type adders [1]–[2] have been demonstrated to have advantages over conventional CLA adders in emitter-coupled logic (ECL) [3]. Ling's approach results in a drastic load reduction in the input stage circuitry, thereby allowing direct generation of group generate from the input operands. Because this approach takes advantage of the dot-or capability of ECL, it is not as suitable for CMOS adders. A straightforward application of Ling's scheme to CMOS adders can lead to an increase in hardware or delay time, or both.

In this paper, we present an implementation of a fully static CMOS adder using a modified Ling scheme. Our implementation saves up to 1 gate delay and always reduces the number of serial transistors in the critical path over the conventional CLA approach with a negligible increase in hardware. In CMOS, because the number of serial transistors from the output to the power or the ground node is one of the major speed limiting factors, reducing it in the critical path is therefore of interest.

In Section II, we show how Ling's approach can be modified for CMOS technology by way of a 32-bit adder design. Our approach, however, is independent of the number of bits in the adder. In Section III, we compare the present adder with other adders reported in the literature. Section IV contains a summary. In this paper, *AND* is denoted by juxtaposition, *OR* by $\vee$, *EXCLUSIVE-OR* by $\oplus$, negation by overbar, and $\prod_{i=1}^{n} p_i$ by p_{1-n}. Index $i \in (0, 32)$ is used for bits (carry-in is treated as g_0 and actual sums range from 1 to 32), index $j \in (0, 10)$ for groups, and index $k \in (0, 2)$ for blocks, exclusively and respectively.

II. The Adder

The adder is divided into 4 blocks, of sizes 9, 9, 9, and 6 bits, respectively. Since carry-in is treated as g_0, Block 0 has only 8 bits. Each block is subdivided into three 3-bit groups, except for the last 6-bit block, which has two 3-bit groups. Within each group and each block, the local sum logic uses the conditional-sum algorithm [4]. Fig. 1 shows the structure and numbering convention of the adder. The ECL adders reported in [1] and [3] have a 4-bit group; owing to the limited fan-in capability of fully static CMOS circuits, 3-bit groups are used in the present adder.[1]

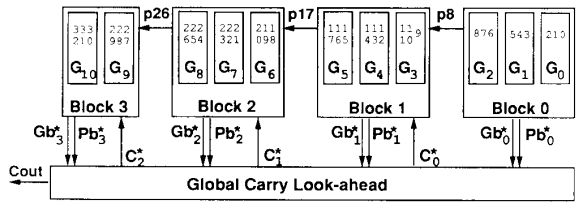


Fig. 1. Structure and naming convention of present CMOS adder. Carry-in is defined as g_0.

Some definitions: $g_i \equiv a_i b_i$, $p_i \equiv a_i \vee b_i$, $g_0 \equiv c_{\text{in}}$, and $s_i \equiv a_i \oplus b_i$, so that $S_i = s_i \oplus c_{i-1}$. In the definitions, a_i, b_i are the ith bits of the input operands and s_i, S_i are the ith bits of the local and final sum, respectively.

A. Global-Carry

For ease of discussion, we illustrate our approach on Group 8 in Block 2 of the adder (see Fig. 1). The conventional group-generate equation for the group is [4]

$$G_8 = g_{26} \vee g_{25}p_{26} \vee g_{24}p_{25}p_{26}. \tag{1}$$

Using the identity $g_i = p_i g_i$ and extracting p_{26}, we rewrite (1) as $G_8 = p_{26}G_8^*$ where $G_8^* = g_{26} \vee g_{25} \vee g_{24}p_{25}$. The essence of Ling's approach is to propagate the G_8^* term only. This is because G_8^* can be expanded as

$$G_8^* = a_{26}b_{26} \vee a_{25}b_{25} \vee a_{24}b_{24}a_{25} \vee a_{24}b_{24}b_{25}. \tag{2}$$

Equation (2) contains 4 terms and a total of 10 literals, with the largest term having 3 literals. This can be implemented in CMOS in 1 complex gate [5]. Equation (1) on the other hand, when expanded, contains 7 terms and a total of 24 literals, with the largest term containing 4 literals. In CMOS, the number of literals in a minterm corresponds to the number of N-channel serial transistors; (2) is therefore preferable to (1) for direct generation of group generate from the input operands. Generating group generate directly saves 1 gate delay in the critical path because the g_i and p_i terms are not implemented.

The reader may have realized that a straightforward implementation of (2) in fully static CMOS has 4 P-channel transistors in series, severely limiting its usefulness. But the P-channel transistor network can be simplified as the relationship of p_i and g_i, $\overline{g}_i\overline{p}_i = \overline{p}_i$, can again be put to use. Fig. 2 shows an implementation of (2); only 3 P-channel transistors are in series. The equation for group propagate in conventional CLA is $P_8 = p_{24-26}$. Ling uses a modified group generate, P_8^*, defined as $P_8^* = p_{23-25}$. G_j^* and P_j^* are, respectively, the reduced and left-shifted versions of G_j and P_j. To see why group generate is defined this way, consider the block-generate equation for Block 2:

$$Gb_2 = G_8 \vee G_7P_8 \vee G_6P_7P_8. \tag{3}$$

Manuscript received June 15, 1990; revised December 15, 1990. This work was supported by NSF Contract MIP88-22961.

The authors are with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

IEEE Log Number 9105496.

[1] Though a fan-in of 4 is often used in fully static CMOS, we have chosen to use a fan-in of 3 to avoid having 4 P-channel devices in series. Our approach is independent of the group size, however.

0018–9340/92$03.00 © 1992 IEEE

Using the definition of P_j^* and G_j^*, we can rewrite (3) as

$$Gb_2 = p_{26}(G_8^* \vee G_7^* P_8^* \vee G_6^* P_7^* P_8^*) = p_{26} Gb_2^*. \quad (4)$$

Hence, the use of P_j^* affords a more efficient implementation of block generate. Equation (4) is easier to implement than (3) because G_j^* are available, but G_j are not. Also, p_{26} in (4) propagates further down into the final-carry equation:

$$C_2 = Gb_2 \vee Gb_1 Pb_2 \vee Gb_0 Pb_1 Pb_2$$

which can be rewritten as

$$C_2 = p_{26} C_2^* \quad (5)$$

where

$$C_2^* = Gb_2^* \vee Gb_1^* Pb_2^* \vee Gb_0^* Pb_1^* Pb_2^*. \quad (6)$$

The final-carry equations C_0, C_1, and C_{out} can be derived similarly as

$$C_0 = p_8 Gb_0^* \quad (7)$$

and

$$C_1 = p_{17} C_1^* \quad (8)$$

where

$$C_1^* = Gb_1^* \vee Pb_1^* Gb_0^* \quad (9)$$

and

$$C_{\text{out}} = p_{32} C_{\text{out}}^* \quad (10)$$

where

$$C_{\text{out}}^* = Gb_3^* \vee Gb_2^* Pb_3^* \vee Gb_1^* Pb_2^* Pb_3^* \vee Gb_0^* Pb_1^* Pb_2^* Pb_3^*. \quad (11)$$

The definitions of Gb_k^* and Pb_k^* in (6), (7), (9), and (11) follow those of the conventional Gb_k and Pb_k with a simple modification: P_j and G_j are replaced by P_j^* and G_j^*, respectively. For example:

$$Gb_1 = G_5 \vee G_4 P_5 \vee G_3 P_4 P_5$$
$$Gb_1^* = G_5^* \vee G_4^* P_5^* \vee G_3^* P_4^* P_5^* \quad (12)$$

and

$$Pb_1 = P_{3-5}$$
$$Pb_1^* = P_{3-5}^*.$$

Ling's scheme calls for either the implementation of C_k from Pb_k^*, Gb_k^*, and C_k^* [(5)–(11)] or the modification of the local sum logic to account for the fact that C_j^* are propagated [1]. Neither options are attractive. The former fails to reduce the number of serial transistors in the critical path and the latter adds complexity to the local sum logic, increasing hardware and delay time.

In the present implementation, only the $C_k^*, Pb_k^*, Gb_k^*, P_j^*$, and G_j^* terms are implemented in the carry look-ahead circuitry without modifying the local sum logic. The p terms in (5), (7), (8), and (10) are implemented in the local group-carry equations, both of which are noncritical paths. In the following section, we show that this is possible and in fact desirable because of reuse of the P_j^* and G_j^* terms. The ability to reuse the P_j^* and G_j^* terms is one of the salient features of the present adder.

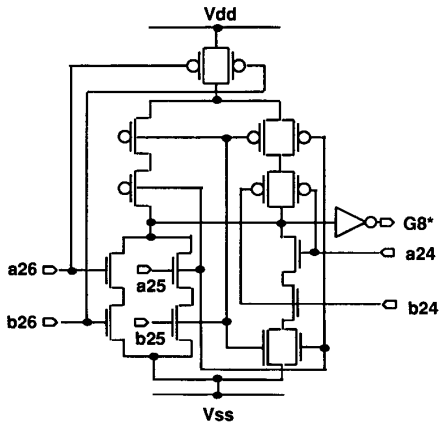


Fig. 2. Typical CMOS implementation of group-generate.

B. Local Group-Carry

We can prove for the general case that the p terms in the final-carry equations (5), (7), (8), and (10) can be implemented in the local group-carry equations for all blocks in the adder. For ease of discussion, however, we show that this is possible for Block 2. That is, we show that only C_1^* needs to be propagated to Block 2 and p_{17} needs only be propagated to Group 6. Equations for the other blocks can be derived similarly.

The final-sum equation for bit 24 in Group 8 (see Figs. 1 and 3) is

$$S_{24} = s_{24} \oplus (G_7 \vee P_7 G_6 \vee P_7 P_6 C_1)$$
$$= s_{24} \oplus \{p_{23}[G_7^* \vee P_7^*(G_6^* \vee P_6^* C_1^*)]\}. \quad (13)$$

Defining

$$gb_7 \equiv p_{23}(G_7^* \vee P_7^* G_6^*) \quad (14)$$

and

$$pb_7 \equiv p_{23}[G_7^* \vee P_7^*(G_6^* \vee P_6^*)] \quad (15)$$

and expanding (13) in terms of C_1^* using Shannon's theorem [6], we get

$$S_{24} = \overline{C}_1^*(s_{24} \oplus gb_7) \vee C_1^*(s_{24} \oplus pb_7).$$

The equations for S_{25} and S_{26} can be derived similarly. The S_{26} one is given below:

$$S_{26} = \overline{C}_1^* \{\overline{gb}_7[s_{26} \oplus (g_{25} \vee p_{25} g_{24})]$$
$$\vee\ gb_7[s_{26} \oplus (g_{25} \vee p_{25} p_{24})]\}$$
$$\vee\ C_1^* \{\overline{pb}_7[s_{26} \oplus (g_{25} \vee p_{25} g_{24})]$$
$$\vee\ pb_7[s_{26} \oplus (g_{25} \vee p_{25} p_{24})]\}. \quad (16)$$

Hence, only C_1^* in (8) needs to be propagated globally to Block 2. p_{17} can be accounted for locally in Group 6 in G_6^* and P_6^* in (14) and (15). Both equations can be implemented in 2 complex gate delays since P_j^* and G_j^* require only 1. In the present implementation, all complex gates have at most 3 serial transistors and are roughly of the same complexity as that shown in Fig. 2. One complex gate delay is roughly equal to two 2-input NAND gate delays at a load of 0.5 pF from SPICE simulation. The $G_6^* \vee P_6^*$ term in (15) is actually available from Group 7 within the same block and can be reused at a cost of 1 complex gate delay, increasing the number of gate delays of pb_7 from 2 to 3. Fig. 3 shows an implementation of Group 8.

In Fig. 3, we have used s_{25} as the local propagate (globally, we used p_i), allowing a more efficient implementation of $(g_{25} \vee p_{25} g_{24})$

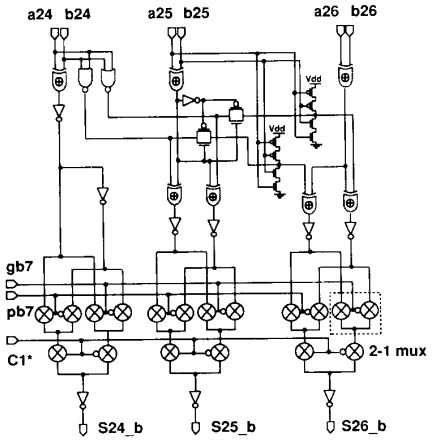


Fig. 3. Logic diagram of Group 8.

and $(g_{25} \vee p_{25}p_{24})$ in (16). Because the present approach does not modify the local sum logic, there is no increase in hardware in Fig. 3 when compared with an implementation that uses the conventional conditional-sum (CSA) algorithm.[2] In terms of number of gate delays, S_{32} is no worse than S_{26}. The equation for S_{32} is similar to (16):

$$\begin{aligned} S_{32} = &\overline{C}_2^*\{\overline{gb}_9[\overline{g}_{30}(s_{32} \oplus g_{31}) \vee g_{30}(s_{32} \oplus p_{31})] \\ &\vee gb_9[\overline{g}_{30}(s_{32} \oplus g_{31}) \vee g_{30}(s_{32} \oplus p_{31})]\} \\ &\vee C_2^*\{\overline{pb}_9[\overline{p}_{30}(s_{32} \oplus g_{31}) \vee p_{30}(s_{32} \oplus p_{31})] \\ &\vee pb_9[\overline{p}_{30}(s_{32} \oplus g_{31}) \vee p_{30}(s_{32} \oplus p_{31})]\} \quad (17) \end{aligned}$$

where $gb_9 = p_{29}G_9^*$ and $pb_9 = p_{29}(G_9^* \vee P_9^*)$. From the previous discussion, P_j^* and G_j^* are available in 1 complex gate delay, Pb_k^* and Gb_k^* in 2, and C_k^* in 3. The final sum selection multiplexor is counted as 1 gate delay. Hence, the present adder has a total of 4 complex gate delays.

III. Comparison With Other Adders

Table I compares the present adder with conventional CLA, conditional-sum, carry-select, and Multiple-Output Domino Logic (MODL) adders [7] in terms of complex gate delays and number of serial transistors in the critical path. The present adder has fewer complex gate delays than other adders. These adders were chosen because their delay times have the same order of growth, $O(\log n)$, as the present adder. In the comparison, we have assumed that the CLA adder is implemented in a complex gate oriented media (i.e., MOS LSI or VLSI) and the carry-select adder uses a 4-bit group and conventional carry look-ahead to propagate the global carry. To be fair, we have further assumed that the conditional-sum adder has a similar organization as the present adder but without using the modified Ling approach and that the MODL adder uses conditional-sum logic locally.

Because CLA, conditional-sum adder, and carry-select adder do not generate group generate directly, they have one more gate delay than the present adder. CLA requires another gate delay to generate the local sum, increasing its gate delay from 5 to 6. The MODL adder though generates group generate directly, it does so by using a small 2-bit group [7], requiring more levels in the global carry generation process than the present adder.

Comparison of complex gate delays in CMOS adders can be misleading because they depend on both fan-in and fan-out. A better measure is the number of serial transistors which a signal must traverse in the critical path. This means that for fully static CMOS circuits, we evaluate both P-channel and N-channel transistors for critical paths. For dynamic CMOS circuits [5], which include DOMINO circuits, we only evaluate the N-channel transistors. Hence, this comparison scheme is slightly biased against fully static CMOS circuits. Admittedly, comparing CMOS adders in terms of serial transistors in the critical path is crude but it does allow us to quickly evaluate the potential performance of an algorithm for further study. A fair comparison scheme should consider area, power consumption, speed, and design turnaround time.

[2] Since we used CSA for the local sum logic, it is fair to compare our adder with CSA, knowing that CSA consumes more hardware than CLA [4].

TABLE I
Comparison of CMOS Adders in Terms of Gate Delays and Number of Serial Transistors in Critical Path

Adders	Gate Delay* from c_{in} to S_{32}	Number of Serial Transistors from c_{in} to S_{32}
CLA	6	21
Condition-Sum Adder	5	16
Carry-Select Adder	5	18
MODL Adder	5	18
Present Adder	4	14

*Complex gate delay. As explained in the text, this is not a good measure of adder speed. Number of serial transistors is a better measure.

In counting the number of transistors in series, the discharge N-channel transistors in DOMINO logic are not included. Inverters are counted as 1 transistor and *XOR* gates as 2.[3] The number of serial transistor count for *NAND*, *NOR*, and complex gates is the number of transistors in the longest N-channel or P-channel chain for static and dynamic CMOS circuits. When there are pass gates [5] involved, the situation is a little more complicated. The input-output (source-drain) path of a pass gate is counted as 0.5 transistors[4] and the control-output (gate-drain) path as 1 transistor. By the same token, 2-1 multiplexors are counted as 2 transistors from the select-output path, but as 0.5 from the input-output path. A similar comparison scheme has been suggested by Oklobdzija and Barnes [8], but the accounting details were not given.

From Table I, the present adder has a fewer number of serial transistors in the critical path than others. To count the number of serial transistors in the critical path, (6), (12), and (17) can be used. The input signal must traverse G_j^* in 4 transistors (Fig. 2), Gb_k^* in another 4 transistors [(12) plus an inverting buffer], and C_2^* in yet another 4 transistors [(6) plus an inverting buffer], giving a total of 12 transistors. All other terms in (17) arrive sooner than C_2^*. The final sum selection multiplexors contribute 2 more transistors. Hence, the critical path from c_{in} to S_{32} in the adder has 14 transistors, as indicated in Table I. As with other adders, the inverting output buffers in Fig. 3 are not counted because they are only included for driving considerations. The path from c_{in} to C_{out} has the same number of serial transistors as the path from c_{in} to S_{32}. This can be seen by rewriting (11) as $C_{out}^* = p_{32}[Gb_3^* \vee Pb_3^*(Gb_2^* \vee Gb_1^*Pb_2^* \vee Gb_0^*Pb_1^*Pb_2^*)]$. This c_{in} to C_{out} path, however, is not the critical path because it has a much smaller capacitive loading than that from c_{in} to S_{32}.

We have laid out an 11-bit adder in an advanced bipolar/CMOS process with 1.0 μm drawn channel length. This 11-bit adder was used as an exponent adder in a 53 × 53 bit multiplier. The bipolar

[3] If both the inputs and their complements are available, the *XOR* gate delay is 1.

[4] When there are several pass transistors in series, one has to take capacitive loading into account as suggested in [8]. This situation never arose in the present study.

devices were exclusively used in the Wallace tree and the final carry-propagate adder. The 11-bit CMOS adder was laid out in a standard-cell fashion and occupied roughly 700×550 μm^2. No automatic compaction was performed on the layout. Based on this adder, the delay of the whole 32-bit adder operated at 5 V driving a 0.3 pF load at room temperature is estimated using SPICE to be around 4.0 ns.

IV. Summary

In summary, we have presented a fully static CMOS implementation of a Ling type adder. The implementation has a fewer number of complex gate delays and a fewer number of serial transistors in the critical path than other conventional adders of the same order of growth complexity (i.e., conditional-sum adder, CLA, carry-select adder, and multiple-output domino logic adder). Compared with a conventional conditional-sum adder, the increase in hardware in the present implementation is negligible.

The key idea presented in this paper that allows Ling's scheme to be used for CMOS adders is that the factored p term in Ling's equation can be propagated locally, reducing the number of serial transistors in the critical carry propagation path. Another minor observation is that by using s_i as the local propagate, the present implementation saves hardware in the local sum logic over the conventional conditional-sum adder (Fig. 3).

Acknowledgment

The authors wish to thank M. Horowitz, A. R. Todesco, and the referees for their valuable comments on the paper. The comments from one of the referees were particularly incisive and helpful.

References

[1] H. Ling, "High speed binary adder," *IBM J. Res. Develop.*, vol. 25, no. 3, pp. 156–166, May 1981.
[2] R. W. Doran, "Variants of an improved carry-look-ahead-sum adder," *IEEE Trans. Comput.*, vol. 37, no. 9, pp. 1110–1113, Sept. 1988.
[3] G. Bewick, P. Song, G. DeMichel, and M. J. Flynn, "Approaching a nanosecond: A 32-bit adder," in *Proc. Int. Conf. Comput. Design*, 1988, pp. 221–224
[4] S. Waser and M. J. Flynn, *Introduction to Arithmetic for Digital Systems Designers.* New York: Holt, Rinehart and Winston, 1982.
[5] J. P. Uyemura, *Fundamentals of MOS Digital Integrated Circuit.* Reading, MA: Addison-Wesley, 1988, ch. 6–9.
[6] E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits.* Englewood Cliffs, NJ: Prentice-Hall, 1986, ch. 2.
[7] I. S. Hwang and A. L. Fisher. "A 3.1 ns 32 b cmos adder in multiple output domino logic," in *Proc. IEEE Int. Solid-State Circuit Conf. Rec.*, 1988, pp. 140–141.
[8] V. G. Oklobdzija and E. R. Barnes. "Some optimal schemes for alu implementation in vlsi technology," in *Proc. of the 7th Symp. Comput. Arithmetic,* June 1985, pp. 2–8.

Concurrent Error Detection and Correction in Real-Time Systolic Sorting Arrays

Sy-Yen Kuo and Sheng-Chiech Liang

***Abstract*—A novel approach to on-line error detection and correction for high throughput VLSI sorting arrays is presented. The error model is defined at the sorting element level and both functional errors and data errors are considered. Functional errors are detected and corrected by exploiting inherent properties as well as newly discovered special properties of the sorting array. Coding techniques are used to locate data errors. All the checkers are designed to be totally self-checking and hence the sorting array is highly reliable. Two-level pipelining is employed which makes the design very efficient and suitable for real-time application. The structure is very regular and therefore, is very attractive for VLSI or WSI implementation.**

***Index Terms*—Coding, concurrent error detection and correction, diagnosis, systolic sorting, totally self-checking.**

I. Introduction

Many applications in real-time digital signal and image processing need a high performance and special purpose architecture for parallel sorting on a huge amount of input data. Sorting arrays which consist of a number of identical processing elements with regular interconnections and high concurrency factors [1], such as the odd-even transposition sort [2], the bitonic sort [3], and the perfect shuffle sort [4], are good candidates for real-time applications. Studies by Kung [5] indicate that both regular cell structures and simple interconnections will dominate the cost in VLSI or WSI implementations. Although both the perfect shuffle sort and the bitonic sort use less sorting elements ($O(N \log_2{}^2 N)$) than the odd-even transposition sort ($O(N^2)$), the wiring complexities of the first two sorters make them more costly to implement than the odd-even sort since, for large N, the wiring space will dominate the silicon area.

Reliability, availability, and continuous operation are also very important in real-time applications. On-line error detection is the first step to increase the reliability. In order to increase the system availability, off-line diagnosis after on-line error detection should be avoided and the system should be able to automate the recovery process. In this paper, we present a highly reliable sorting array which can detect multiple errors and correct a single error for on-line applications. Since the array is based on the odd-even transposition sort, it has a regular structure and simple interconnection links. Both the regularity and the simplicity are preserved by the presented fault tolerance technique so that redundancy can be included into the system easily. Also, it can be reconfigured easily to tolerate the faulty sorting elements located by the on-line fault diagnosis procedure and can be degraded gracefully after redundancy is exhausted.

Extra cost incurred by bringing in fault tolerance features is minimized by exploiting the inherent properties of the embedded sorting algorithm. Properties such as nondecreasingly or nonincreasingly ordered output sequence are used to check the functional correctness of the sorting array. In contrast with assuming that a faulty sorting element will transmit its inputs to the outputs unchanged or a faulty element can be located by some external circuits and then bypassed

Manuscript received July 26, 1990; revised November 11, 1991.
The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, R.O.C.
IEEE Log Number 9200319.

0018–9340/92$03.00 © 1992 IEEE

as in [6] and [7], a faulty sorting element in our error model can either pass or swap data incorrectly. Also, we discovered an important robust property of the odd-even transposition sorting array in which a single error can be masked automatically and multiple errors can be detected concurrently without disturbing the normal circuit operation.

In addition to checking the order of the outputs, the code-preserving property in data manipulation is employed to check whether the output data has been modified. Errors which violate the code-preserving property can be detected by using an appropriate coding technique. Depending on how critical the applications are, the requirements of fault coverage as well as the corresponding coding techniques will be different. Three example coding techniques are evaluated and the results are shown in Section VI. The total overhead of the proposed approach based on our analysis is much lower than previous fault tolerance techniques for other pipelined array processors [8], even if the checkers in the array are designed to be totally self-checking to increase the reliability.

II. Array Architecture and Cell Realization

In order to have a high performance system, the two-level pipelining technique [9] which is frequently used in sorting arrays to achieve very high throughput [10] is employed in the design here. In addition to the use of the pipelined odd-even transposition sort as the word-level structure (Section II-A), the systolic data flow concept [5] is used for the bit-level pipelining (Section II-B). In this paper, we use the term CS *element* to represent a word-level compare-swap element and the term *cell* to represent a bit-level compare-swap element. Also, without loss of generality, we will assume that the sorted output sequence is in nonincreasing order.

A. Array Architecture

The word-level pipelines can be achieved by one of the parallel sorting algorithms such as the odd-even transposition sort, the bitonic sort, the perfect shuffle sort, or the balanced sort. An example odd-even transposition sorter with $N = 5$ (without loss of generality, N is assumed to be an odd number) is shown in Fig. 1. The parallel odd-even transposition sorting array consists of a cascade of N stages with $N(N-1)/2$ CS elements in each stage to sort N input data elements [2]. Each CS element in the sorting array compares two n-bit input numbers x and y and swaps these two values if $x < y$. Data registers (D) in Fig. 1 are used as delay buffers so that input data sets can be synchronized by the system clock and pipelined through stages of the sorting array.

B. Cell Realization

In the word-level, there are only two types of elements in the sorting array: data registers (D) and compare-swap elements (CS). For each CS element, n bit-level comparisons are required to compare two n-bit binary numbers. Since the goal here is to have high throughput systems, the systolic data flow concept is also applied in the bit-level pipelines. A matrix of single-bit data registers (d) is cascaded before the input stage to synchronize the data flow as shown in Fig. 2. (Note that this matrix of d registers is required only before the CS elements of the first stage.) These data registers are arranged as a lower right triangular matrix such that input data bits can enter the systolic sorting array in a skewed fashion. That is, for a CS element, when c_n has finished processing x_n and y_n the comparison results can pass to cell c_{n-1} together with the two inputs x_{n-1} and y_{n-1} at the same time. Therefore, c_n can process the next inputs a_n and b_n when c_{n-1} is processing x_{n-1} and y_{n-1}. The n cells of each CS element are chained together by the swap control lines r and s.

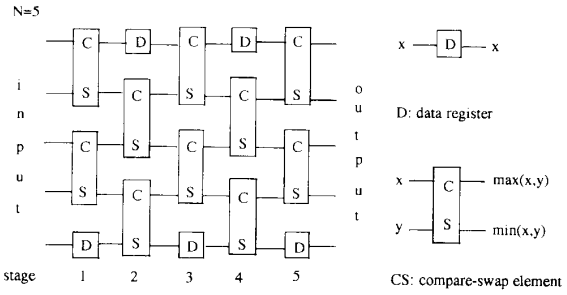


Fig. 1. Odd-even transposition sort: word-level structure.

Logical operations of a cell c_i in a CS element are described in the following: 1) Signals s_{i+1} and r_{i+1} from cell c_{i+1} to cell c_i indicate whether any of the more significant bits than bit i has been swapped or not. 2) Inputs x_i and y_i are the ith bits of the two input words x and y to a CS element, respectively. 3) The signal r_i indicates whether $(x_n, \cdots, x_i) = (y_n, \cdots, y_i)$ or not ($r_i = 0$ or 1, respectively) and s_i indicates whether $(x_n, \cdots, x_i) < (y_n, \cdots, y_i)$ or not ($s_i = 1$ or 0, respectively). 4) α_i and β_i are two output data bits from cell c_i with $\alpha_i \geq \beta_i$. Therefore, we have the bit-level cell structure and logical equations as shown in Fig. 3.

The swap control signals s_1, r_1 from cell c_1 indicate whether the two inputs, x and y, have been swapped ($s_1 = 1$, $r_1 = 1$) or not ($s_1 = 0$). We call s_1 the *swap-indicator* since it alone can tell us if there is any swap operation performed in the corresponding stage.

III. Properties of the Sorting Array

A. Error Model

The error model is defined at the CS element level. A CS element which contains physical faults can generate errors such as swapping its inputs incorrectly, modifying the data values, or both, and can be classified as a *functional error*, a *data error*, or a *hybrid error*, respectively. For example, stuck-at faults on the two swap control lines can cause functional errors and stuck-at faults on the communication links can cause data errors. Effect of faults on links between stages i and $i+1$ is lumped into stage $i+1$ such that errors in communication links are also representable in this word-level error model. Faults in communication links are less common [11] but more severe since, in a sorting array, a faulty communication link will cause the entire output data useless unless a reconfiguration process followed by a recovery process is applied.

B. Properties

Due to the limitation of space, all the proofs of theorems are omitted and can be found in [12]. The first well known property is that the systolic sorting array based on the odd-even transposition sort with N stages and $(N-1)/2$ elements in each stage is a valid sorting array and a random input sequence will be correctly ordered at the outputs. The second property is that the sorting array is a code-preserving sorter. This is due to the fact that the sorting array consists of CS elements and data registers only, no logical or arithmetic operation which can modify data values is performed during normal circuit operations. Therefore, the order of input sequence may be modified at the outputs but the coded input values should be preserved.

These two properties inherently exist in all sorting arrays and any sorter can be examined functionally according to these two properties. In addition to these two general properties, we have derived a special property for functional error checking which can be applied to all

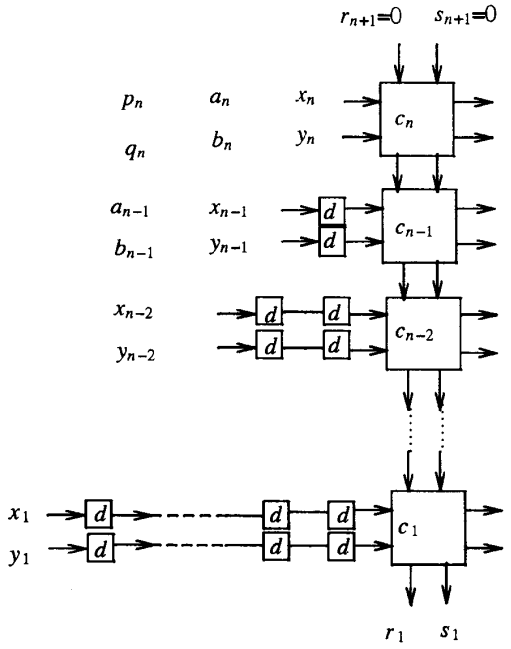


Fig. 2. Structure of a CS element and the matrix of data registers.

pipelined sorting algorithms and a robust property for the odd-even transposition sort only, in which any single functional error can be recovered automatically.

It should be noted that as was discussed in Section III-A, it is possible for a CS element to swap its inputs incorrectly such that the entire output sequence from a sorting array is not nonincreasingly ordered. From the odd-even transposition sort (Fig. 1), we can see that two neighbor stages in the odd-even transposition sorting array completely compare all pairs of adjacent inputs in two clock cycles. Therefore, if two additional neighbor stages which include an odd-numbered stage and an even-numbered stage are added after the last stage of any sorting array, they can be used as a checker to check whether the outputs from the sorting array are ordered or not. If the output sequence is correctly ordered, no swap operation will be executed in any CS element of these two additional stages, otherwise, some of these CS elements will perform swap operations and it represents that the output sequence from the sorting array is not correctly ordered. We call these two stages the *Nonincreasing-Order-Checker* (NOC).

Theorem 1: The nonincreasing-order-checker (NOC) can determine whether the output sequence from the sorting array is correctly ordered or not. □

The second property is the robust property of the odd-even transposition sort. This robust property is very important in on-line real-time applications. For on-line applications, the probability of a single error is much higher than multiple errors. If a single error can be recovered automatically without interrupting the entire system, the system availability will be increased significantly.

Theorem 2: The systolic sorting array for N inputs based on the odd-even transposition sort with $N + 2$ stages can recover from a single functional error in the first N stages automatically. □

IV. Fault Tolerance

By checking the two general invariant properties, the sorting array has the capability of concurrent error detection. By exploiting the special property we derived in Theorem 2, the sorting array can correct a single functional error during the normal operation. It is difficult to correct data errors in a sorting array during the normal operation. Even if the faulty bits can be detected and corrected by some coding techniques such as Hamming code, the output sequence is no longer correctly ordered. Therefore, with the assumption that the hardware used for off-line diagnosis and yield enhancement such as the multiplexers and the bypass registers in each CS element are fault-free, we will present a fast on-line fault diagnosis procedure in Section V-B to locate the faulty sorting elements.

As shown in Theorem 1, whether the output sequence is in nonincreasing order or not can be detected by the NOC. Error correction for a single functional error is done automatically as shown in Theorem 2. Therefore, the two stages added to a sorting array can be either a checker or a single error corrector. These two stages are sufficient for a single error. But for multiple errors, two more stages are required to detect other errors after the first error has been corrected by the first two added stages.

The problem of who will check the checkers is very important in mission critical applications. The two additional stages used to recover a single error in the array will not be able to recover errors in themselves. The errors in the NOC itself will generate a useless result if the NOC does not have a self-checking capability to check its own outputs. Therefore, from Theorems 1 and 2, the sorting array for N inputs can be implemented with $N+4$ stages for error detection and correction. The first N stages are for normal sorting functions. Stages $N+3$ and $N+4$ are used as the NOC and will be designed to be *totally self-checking* (TSC) [13]. (The details on implementing a TSC checker will be discussed in Section V.) Stages $N + 1$ and $N + 2$ which are used to correct a single functional error do not need to be TSC circuits since their outputs are checked by stages $N + 3$ and $N + 4$ (NOC). However, in the following theorem, we will show that stage $N + 4$ can be omitted from the sorting array if stage $N + 2$ is implemented by TSC circuits.

Theorem 3: The systolic sorting array with a total of three additional stages, stages $N + 1$, $N + 2$, and $N + 3$, where stages $N+2$ and $N+3$ are implemented by TSC circuits can tolerate one incorrect swap operation and check whether the output sequence is nonincreasingly ordered or not. □

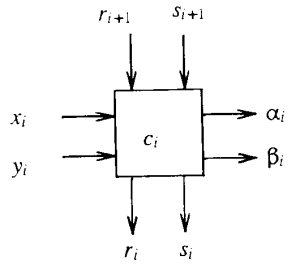
In addition to checking the correctness of the output order, we can check whether the input data values are preserved during the normal operation or not by using appropriate coding techniques. The choice of a data error detection method is very flexible depending on the properties of the sorting array, the type of errors to be detected, and the fault coverage requirement.

V. Design of Totally Self-Checking Checkers

We have designed a data error detector and an NOC to detect data errors and functional errors, respectively. It is always desirable to design checkers which can detect errors in the checker itself as well as in its inputs. This leads us to design checkers which are totally self-checking (TSC). The concept of a totally self-checking checker has been formalized in [14] as a circuit which is fault secure, self-testing, and code disjoint [13].

A. Design of a Totally Self-Checking Data Error Detector

A general structure of the totally self-checking data error detector for the systolic sorting array is shown in Fig. 4. Check bits from the *check symbol generator* (CSG) are generated based on the coding technique used. They are attached to the corresponding data (information) and propagated through the array but are not processed by the systolic sorting array before arriving the *two-rail checker*



$$r_i = r_{i+1} + (x_i + y_i)$$

$$s_i = s_{i+1} + \bar{r}_{i+1}(\bar{x}_i \cdot y_i)$$

$$\beta_i = \bar{s}_i \cdot y_i + s_i \cdot x_i$$

$$\alpha_i = \bar{s}_i \cdot x_i + s_i \cdot y_i$$

Fig. 3. Structure of a compare-swap cell and its logical functions.

(TRC) (TRC is a two-level *AND-OR* circuit as in [13] and will be described latter). At outputs, these input check symbols are compared with the outputs from the *complement check symbol generator* ($\overline{CSG}$ is a combination of CSG which generates check bits for the received data and an inverter at the output of each check bit) through a tree of two-rail checkers. As discussed in the previous section that any input data should not be modified by the systolic sorting array, so the check symbols generated by the $\overline{CSG}$ should be complementary to the check symbols generated by the CSG if both the checker and the sorting array are fault-free.

In the case that only one check bit is generated for each codeword (for example, by using the single parity code to detect data error), since N inputs will be processed in parallel, N 1-out-of-2 code outputs [(01) or (10) for codeword outputs and (00) or (11) for noncodeword outputs] will be generated in parallel during normal operations. Therefore, a tree of two-rail checkers which maps N input pairs into one output pair can be used to combine these information together and generate a single output (10) or (01) in the normal operation and (00) or (11) as an error message. In order to have a high fault coverage, usually more than one check bit of each data will be generated by the CSG and $\overline{CSG}$ (for example, by using either the Berger code, the modified Berger code, or the low-cost code) and therefore, an intermediate-level two-rail checker is required for each codeword to map the outputs of check symbols and complement check symbols into a single output pair. Sometimes, the combination of inverters of the $\overline{CSG}$ and the intermediate-level two-rail checkers are called an *equality checker* [13] because it can check whether the input check symbols are the same as the output check symbols or not.

Design of a TSC checker for single bit parity code is quite simple [14]. Designs of a totally self-checking checker for the modified Berger code and a self-checking checker for the Berger code were presented in [15] and [16], respectively. To avoid the problem of two legal representations of zero during the calculation of residues, either special definitions are required for the modulo $2^m - 1$ adder in the check symbol generator [17] (where $2^m - 1$ is the check base of the residue code) or a code translator is added between the equality checkers and the two-rail checkers to design an efficient TSC checker for the low-cost code.

It has been shown that the two-rail checker is a totally self-checking checker [13]. The combination of the CSG and the $\overline{CSG}$ can be a totally self-checking checker for different coding techniques such as for the simple parity code [14], the modified Berger code [15], the Berger code [16], and the low-cost code [17]. Since the output pair from the CSG and $\overline{CSG}$ can generate all 0, 1 sequences needed to test the two-rail checker tree, the combination of these two circuits preserves properties of TSC [13].

B. Design of a Totally Self-Checking Order Checker

To design a TSC CS element, the concept of duplication with comparison is used to generate m-variable ($m = (N-1)/2$) two-rail code (or 1-out-of code). Every Boolean function $f(x)$ has a corresponding dual function $f_d(x)$ such that $f_d(\bar{x}) = \bar{f}(x)$. If we apply x to the function f and $\bar{x}$ to the function f_d, the resulting output should be complementary to each other and can be used as inputs to a TSC two-rail checker. Since all the cell elements are simple combinational circuits, it is possible to duplicate all the cells in the last two stages with complementary circuitry. This can be further simplified since outputs x_i and y_i are checked by the data error detector. Therefore, only the output information s_i and r_i which indicate whether the cell c_i performs swap or not should be duplicated in order to design the TSC CS elements. These CS elements which are implemented according to the above method of designing TSC circuit will generate paired *swap-indicators* in the form of the 1-out-of-2 code. That is, if a CS element has a functional error, its output pair $(s_1, \bar{s}_1)$ will be either (00) or (11) and will be (01) or (10) if it is fault-free.

The stage $N+2$ which is used to correct a functional error should be designed as TSC checkers as shown in Theorem 3. All output pairs of $(s_1, \bar{s}_1)$ from word-level TSC CS elements in this stage will be either (01) if there is no swap operation or (10) if there is any swap operation performed during normal operations and (00) or (11) if there is an error in a CS element. Since these 0,1 sequences can completely test the two-rail checker (TRC) tree which are used to map N output pairs to form a single output pair, the combination of TSC CS elements with TSC two-rail checker constitutes a TSC checker. The output pair from the two-rail checker indicates whether there are functional errors [output pair is (11) or (00)] in this stage or not [output pair is (01) or (10)].

In addition to stage $N+2$, CS elements in stage $N+3$ are also designed as TSC circuits to generate m-variable two-rail code such that if there is no functional error in this stage, then the paired output $(s_1, \bar{s}_1)$ of each CS element is either (01) or (10). In addition, if the input sequence to this stage has been ordered correctly, then the *swap-indicators* of all CS elements in this stage should be all 0's and their complement signals are then all 1's, i.e., the paired output $(s_1, \bar{s}_1)$ for all CS elements are (01).

During normal operation, the input sequence to stage $N+3$ will be in correct order if there is no functional error. Therefore, the inputs to the *AND-OR* pair which is used to map m-variable two-rail code to a single output pair as an error indicator will be all 0's for the *OR* gate and all 1's for the *AND* gate (these two gates can be viewed as a tree of two input gates if $m > 2$). The output pair $(S, \bar{S})$ from the *AND-OR* circuit should then be (10). This *AND-OR* circuit can be shown to be code disjoint (this can be proved easily by expanding the truth table to include all possible inputs) and fault secure. The reason that it is fault secure is described in the following. Suppose that a fault has occurred in the *OR* gate (or *AND* gate). Depending on the input, a single fault in it may not produce an error or produce

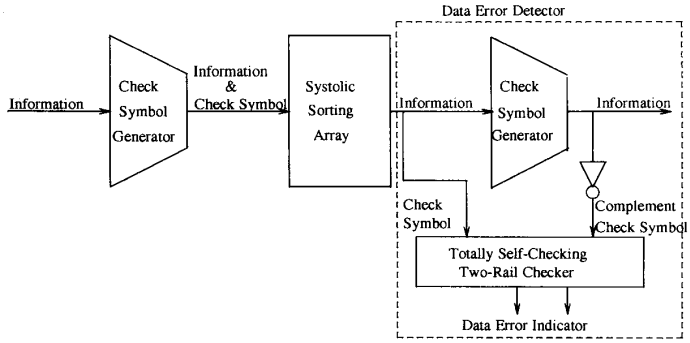


Fig. 4. General structure of a totally self-checking data error detector.

an error value which is the 1's complement of the correct value. In the first case, the fault will not affect the output of the gate. In the second case, a fault in the *OR* gate will not affect the output from the *AND* gate and a codeword will not be produced. Therefore, for single faults the output of the *AND-OR* pair is either the correct output or a none codeword and consequently, it is fault secure.

It is impossible for this paired *AND-OR* circuit to be self-testing under the condition that there is only one code input during normal operation. Therefore, the NOC which includes both the $TSCCS$ elements and the *AND-OR* circuit will not be a totally self-checking checker because the *swap-indicators* and their complements from CS elements in stage $N+3$ can not generate all the input sets to test the *AND-OR* circuit during normal operation. But it is indeed fault secure and code disjoint which will increase the system reliability.

A complete word-level structure of the fault-tolerant sorting array with $N = 5$ is presented in Fig. 5. Input data can be encoded with a parity code, a Berger code, a modified Berger code, or a low-cost code by the check symbol generator (CSG) before entering the sorting array. The output sequence is then checked by the TSC checkers which include a DED to detect data errors in the output and an NOC to check whether the output is nonincreasingly ordered. Stage $N+2$ is implemented as totally self-checking circuits to check if all the compare-and-swap functions performed by the CS elements in this stage are correct. The swap error signals from stage $N+2$ will generate an output pair 11 or 00 if there is an incorrect swapping in this stage.

VI. Evaluation And Discussion

The impact of the proposed fault tolerance techniques on fault coverage, area and time overhead will be evaluated. Multiple functional errors can be detected by the NOC and any single functional error is masked by the first two additional stages as shown in Theorem 1 and Theorem 2, respectively. Faults in the NOC will be either masked or detected by the NOC itself due to the fault secure and code disjoint properties. Coverage of data errors in the proposed sorting array will depend on the complexity of the coding technique employed to detect data errors. Since there is no arithmetic operation involved in the sorting array and it is well known that some physical defects in the VLSI circuits tend to generate unidirectional errors, only the simple parity check code, the Berger code, and the modified Berger code will be considered as potential coding techniques for data error detection. For the simple parity code, only single bit error in each data word will be detected. However, it incurs the least hardware overhead. All unidirectional errors can be detected by the Berger code but the overhead is at least 22% more than the modified Berger code which

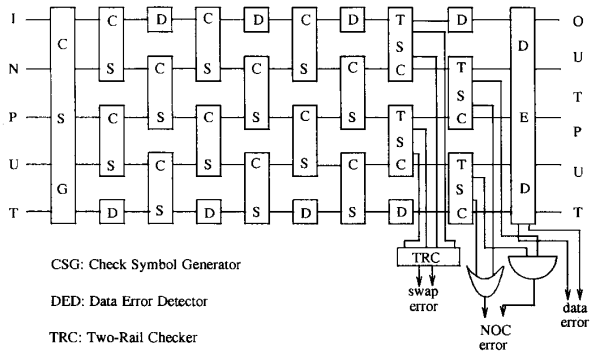


Fig. 5. A complete fault-tolerant sorting array with $N+3$ stages.

has a 93% fault coverage [15]. Error detection for other types of errors can be achieved by using more complicated codes such as the AN code, the check sum code, and the low-cost code. However, even they may have higher fault coverage and lower fault masking effect, the requirement of multipliers, adders, or dividers makes them inefficient for VLSI implementation. For example, with the same number of check bits, the number of full adders required by the low-cost code to detect undirectional multiple errors as well as errors produced by arithmetic processors is almost twice of that required by the modified Berger code [15].

In the following hardware overhead analysis, the determination of overhead ratio will be on the gate level. Since comparisons of the TSC Berger checker, the modified Berger checker, and the TSC low-cost code checker have been discussed in [15], we will only discuss the overhead ratio for the parity and the modified Berger codes. The number of check bits in the modified Berger code is assumed to be 2. The number of gates in a 1-bit full adder and a half adder in the check symbol generator of the modified Berger code is 5 and 2, respectively.

Based on the analysis in [12], examples of overhead ratios on different values of n and N for the simple parity code are shown in Table I where n is the number of bits in each word and P and A are the numbers of gates of the extra circuits and the original array, respectively.

From the table, it is observed that the difference between overhead ratios for arrays with 8-bit words and 16-bit words is very small. The overhead ratio drops in proportion to $1/N$ and therefore, the overhead ratio is smaller for an array with larger inputs.