Evaluation of $A + B = K$ Conditions Without Carry Propagation

Jordi Cortadella and José M. Llabería

***Abstract*—The response time of parallel adders is mainly determined by the carry propagation delay. This paper deals with the evaluation of conditions of the type $A+B=K$. Although an addition is involved in the comparison, we show that it can be evaluated without carry propagation, thus drastically reducing the computation time. Dependencies produced by branches degrade the performance of pipelined computers. The evaluation of conditions is often one of the critical paths in the execution of branch instructions. A circuit is proposed for the fast evaluation of $A + B = K$ conditions that can significantly improve processor performance.**

***Index Terms*— Addition, carry propagation, comparison, conditional branches, parallel adders, pipelined architectures.**

I. Introduction

Carry propagation determines the critical path in the response time of parallel adders. Several approaches have been proposed to reduce it [1], [2]. VLSI techniques have also been used to minimize design costs and chip area [3], [4]. All the efforts to design fast adders are focused on speeding up the carry computation. Regardless of the technique used, the minimum response time of a feasible n-bit adder is $O(\log n)$.

The problem addressed in this paper is closely related to parallel adders. A circuit for evaluating when the addition of two numbers is equal to another number ($A+B=K$) is presented. The usual way to design such a circuit involves the inclusion of an adder, which makes the computation time depend on the delay produced by the carry propagation. The approach we propose avoids the carry propagation and, therefore, drastically reduces the response time.

An immediate application of this approach can be found in the execution of conditional branches. The evaluation of conditions is one of the most frequent operations performed in computer instruction execution. Conditional branches must perform a comparison to determine whether or not a branch must be taken. This dependency may produce hazards that delay the execution of the following instructions. This problem becomes very important in pipelined computers, where the overlapped instruction execution increases the probability of generating hazards. The circuit presented in this paper can be used to alleviate the dependencies produced by conditional branches. Considering the execution frequency of this kind of instruction, processor performance can be significantly improved.

The paper is organized as follows. Section II formulates the problem addressed in the paper. Section III shows how carry propagation can be eliminated from the condition evaluation. Section IV presents the design and evaluation of the circuit and its inclusion in a conventional ALU. Section V proposes the utilization of the circuit for the efficient execution of conditional branches in pipelined computers. This utilization is illustrated with some examples in the scope of RISC architectures.

Manuscript received August 1, 1990; revised January 24, 1991.

The authors are with the Computer Architecture Department, Polytechnic University of Catalonia, Gran Capità s/n, Mòdul D4, 08071-Barcelona, Spain.

IEEE Log Number 9205054.

II. Formulation of the Problem

Given three n-bit vectors $A = (a_n, a_{n-1}, \cdots, a_1)$, $B = (b_n, b_{n-1}, \cdots, b_1)$, and $K = (k_n, k_{n-1}, \cdots, k_1)$ which represent two's complement integers, a circuit must be designed that evaluates the condition $A + B = K$. Hereafter, we will call it Fast Adder-Comparator (FAC). The conventional way to compute this condition is to carry out first the addition $A + B$ and, afterwards, to compare the result with K. The behavior of an adder computing $A + B$ can be described as follows:

$$p_i = a_i \oplus b_i \qquad \text{(Carry propagation)} \tag{9}$$

$$g_i = a_i \wedge b_i \qquad \text{(Carry generation)} \tag{10}$$

$$c_i = (p_i \wedge c_{i-1}) \vee g_i \qquad (\text{Carry. } c_0 = 0) \tag{11}$$

$$r_i = p_i \oplus c_{i-1} \qquad \text{(Addition result).} \tag{12}$$

Vector $R = (r_n, r_{n-1}, \cdots, r_1)$ is the result of the addition $A + B$. If E is the result of comparison $R = K$, E can be defined as

$$e_i = \overline{r_i \oplus k_i} \qquad (e_i = 1 \Leftrightarrow r_i = k_i) \tag{13}$$

$$E = E_n = \bigwedge_{i=1}^{n} e_i. \tag{14}$$

The response time of a circuit computing E in such a way is mainly determined by the delay produced by the recursive definition of c_i in (3), which is of order $O(n)$ for a ripple-carry adder and $O(\log n)$ for faster techniques, like a carry-look-ahead adder.

III. Elimination of the Carry Propagation

In this section, we propose an alternative expression for evaluating E. To avoid the use of the recursive equation (3), we define the *required carry out* (v) and the *required carry in* (w) as follows:

$$v_i = \left(p_i \wedge \overline{k_i}\right) \vee g_i \qquad (v_0 = 0) \tag{15}$$

$$w_i = p_{i+1} \oplus k_{i+1} \tag{16}$$

where v_i is the carry out of the adder stage i when $k_i = r_i$. Similarly, w_i is the corresponding carry in to the adder stage $i + 1$ required for $k_i = r_i$. We define z as the equivalence of v and w:

$$z_i = \overline{v_{i-1} \oplus w_{i-1}} \tag{17}$$

and

$$Z = Z_n = \bigwedge_{i=1}^{n} z_i. \tag{18}$$

The role of Z is to substitute E for the evaluation of the condition. The computation of Z is based on the definition of the *required carry in* and *carry out*, whose expressions are not recursive. Now, we only have to prove that the definitions of Z and E are equivalent. This is the purpose of the following theorem:

Theorem: $E_n = Z_n$

Proof: By induction on n.

First, we will prove the theorem holds for $n = 1 (E_1 = Z_1)$. From (7) and (8) we have

$$Z_1 = z_1 = \overline{v_0 \oplus w_0} = \overline{p_1 \oplus k_1}.$$

From (5) and (4), and since $c_0 = 0$ we have

$$E_1 = e_1 = \overline{r_1 \oplus k_1} = \overline{p_1 \oplus k_1}$$

and therefore $E_1 = Z_1$.

0018-9340/92$03.00 © 1992 IEEE

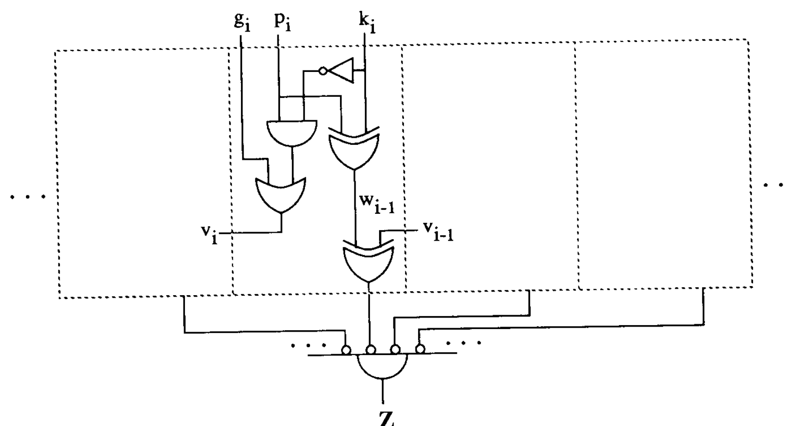

Fig. 1. Fast Adder-Comparator.

Next, we will prove that the theorem holds for $n > 1$. Let us assume it holds for $n-1$ bits ($E_{n-1} = Z_{n-1}$). Equations (6) and (10) can also be defined as

$$E_n = e_n \wedge E_{n-1} \text{ and } Z_n = z_n \wedge Z_{n-1}.$$

Thus, if $E_{n-1} = Z_{n-1} = 0$ then $E_n = Z_n = 0$ and the theorem holds. If $E_{n-1} = Z_{n-1} = 1$, we also have $e_{n-1} = z_{n-1} = 1$, $E_n = e_n$, and $Z_n = z_n$. So, to prove the theorem, we must prove that $e_n = z_n$.

From (9), (7), and (8) we have

$$z_n = \overline{v_{n-1} \oplus w_{n-1}} = \overline{\left((p_{n-1} \wedge \overline{k_{n-1}}) \vee g_{n-1}\right) \oplus (p_n \oplus k_n)}.$$

Since $e_{n-1} = 1$ then $r_{n-1} = k_{n-1}$, and from (4) we have

$$k_{n-1} = p_{n-1} \oplus c_{n-2}.$$

By substituting k_{n-1} and using (3) and (4) we obtain

$$z_n = \overline{((p_{n-1} \wedge c_{n-2}) \vee g_{n-1}) \oplus (p_n \oplus k_n)} = \overline{(c_{n-1} \oplus p_n) \oplus k_n}$$
$$= \overline{r_n \oplus k_n} = e_n.$$

Therefore, the theorem also holds when $E_{n-1} = Z_{n-1} = 1$. ■

The basic contribution of this theorem is the substitution of the carry defined in (3) by the *required carry in* and *carry out* defined in (7) and (8). Since these definitions are not recursive, the required carries can be computed in parallel for all the stages of the circuit, thus drastically reducing the computation delay. While the delay time for e_i is $O(i)$ by using a ripple carry adder, or $O(\log i)$ by using a carry-look-ahead adder, the delay time for z_i is $O(1)$ by using the Fast Adder-Comparator. The computation of E and Z from e_i and z_i, respectively, requires an n-input AND gate, which can be implemented as a tree of constant fan-in AND gates, thus resulting in a delay time of $O(\log n)$.

IV. Circuit Design and Performance Evaluation

Fig. 1 shows a realization of the circuit described in the previous section. Two main parts can be distinguished: an array of cells computing z_i and an n-input AND gate that yields the condition result. The area complexity of the circuit is $O(n)$. The delay time of the array of cells is constant since there is no signal propagation through the array.

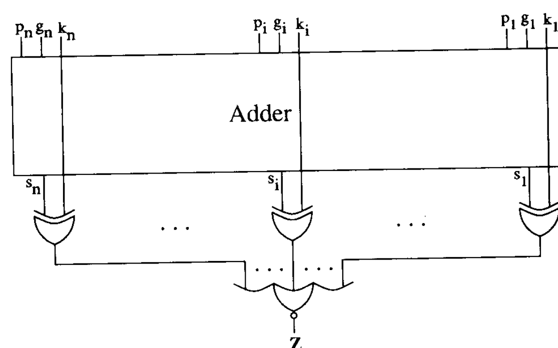

Fig. 2. Adder and comparator.

TABLE I
Delay and Silicon Area for Several Adder-Comparator Circuits. (Implementation With a 1.5 μm Standard Cell Library)

n	RCA Delay (ns)	RCA Area (mm²)	CLA Delay (ns)	CLA Area (mm²)	FAC Delay (ns)	FAC Area (mm²)
8	26.95	0.28	23.35	0.45	13.53	0.30
16	45.75	0.58	29.22	1.16	14.56	0.64
32	85.47	1.53	37.51	3.59	15.63	1.53
64	164.17	3.79	44.80	8.71	17.80	3.55
128	320.30	10.80	53.54	24.22	18.93	9.80

A. Delay Time Evaluation

The delay time improvement of this approach compared to other conventional ones has been evaluated by means of the simulation of several VLSI implementations. The evaluated circuits have the following structure:

- A ripple-carry adder, n bit-comparators (EXOR gates), and an n-input NOR gate, as shown in Fig. 2 (RCA in Table I).
- The same structure with a carry-look-ahead adder instead of a RCA. The adder is built as a tree of 4-bit carry-look-ahead circuits as described in [1] (CLA in Table I).
- The *Fast Adder-Comparator*, depicted in Fig. 1 (FAC in Table I).

Although the delay times of the FAC and the circuit with the CLA are of the same order ($O(\log n)$), their constant factor is significantly different. The logarithmic delay of the FAC is only determined by the n-input AND gate. The delay of the CLA structure is mainly

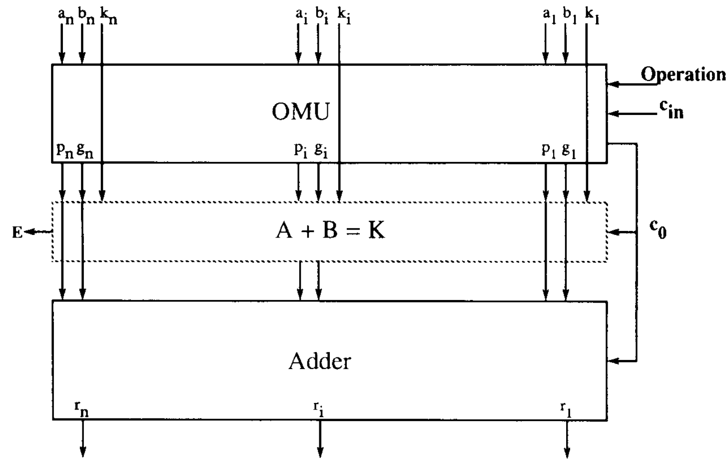

Fig. 3. ALU with the "$A + B = K$" evaluator.

TABLE II
p_i AND g_i EXPRESSIONS FOR AN OPERAND MODIFIER UNIT.
(*logop**: ANY LOGICAL OPERATION (AND, OR, XOR, ...))

Operation	p_i	g_i	c_0
add	$a_i \oplus b_i$	$a_i b_i$	0
sub	$a_i \oplus \overline{b_i}$	$a_i \overline{b_i}$	1
*logop**	a_i *logop* b_i	0	0

determined by the computation of the carry. For this reason, an important time saving can be obtained by using the FAC. The area of the FAC is similar to that of the circuit with a Ripple-Carry Adder.

B. ALU and Fast Comparator

The FAC can also be used in combination with an adder-based ALU, such as the one described in [5] and depicted in Fig. 3. This type of ALU can perform additions, subtractions and logical operations. It consists of two modules: the Operand Modifier Unit (OMU) and the adder. The OMU computes the carry propagation and generation signals (p_i and g_i) from the input data and the operation to be performed. Table II details the expressions for p_i, g_i, and c_0 depending on the ALU operation. The adder behaves according to expressions (3) and (4) in Section II.

This model of ALU is suitable for use in RISC-like processors. Since one of the most important features of these processors is the simplicity of its pipeline, multiple-cycle operations, such as multiplication and division, are not directly supported in hardware [6]. With this scheme, not only the conditions of the type $A+B = K$ can be detected, but also any condition of the type A *op* $B = K$, where *op* is any of the operations the ALU can perform. In this way, the comparison result can be detected long before the ALU result is known. A particular application of this circuit is the advanced computation of the flag Z of the ALU (before the ALU result is known). In this case, the circuit must detect the condition A *op* $B = 0$ and its design becomes much simpler than the one presented in Fig. 1. A similar approach for the case of zero-sum detection has been presented in [7].

The design of an ALU with a fast computation of conditions is the basis of the technique proposed in the next section for the efficient execution of conditional branches in pipelined processors.

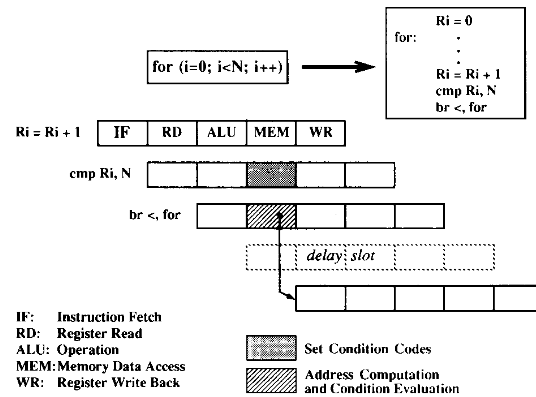

Fig. 4 Branches on condition codes.

V. A CASE STUDY: FAST CONDITIONAL BRANCHES

The dependencies produced by the execution of conditional branches have been extensively studied by many authors [8], [9]. The operation that mainly restricts the potential performance a pipelined processor can reach is the condition evaluation. Next, different approaches for conditional branches in pipelined processors are presented: branches on condition codes, compare&branches, and fast conditional branches. The latter is the mechanism proposed in this paper to execute branches efficiently, based on the *Fast Adder-Comparator* described previously.

A. Branches on Condition Codes

Fig. 4 depicts the execution of a conditional branch in a RISC-like pipeline structure. The control flow code shown in this figure detects the stop condition of a loop. A compare instruction sets the values of the condition codes, while the branch instruction evaluates the condition based on the values of the condition codes. In the pipeline model used in this example, the condition codes are set at the same stage as the condition is evaluated.

The arrow depicted in Fig. 4 represents the control dependency produced by the condition evaluation. The branch target address is also computed during the second stage of the branch execution. These dependencies delay the execution of the instruction that follows the

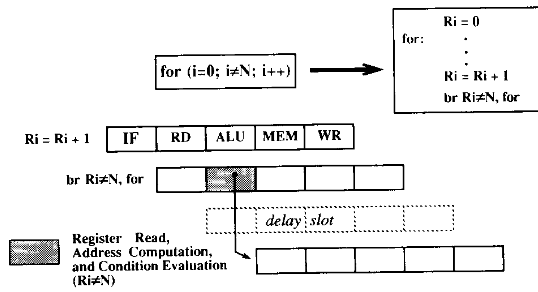

Fig. 5. Compare&Branch instruction.

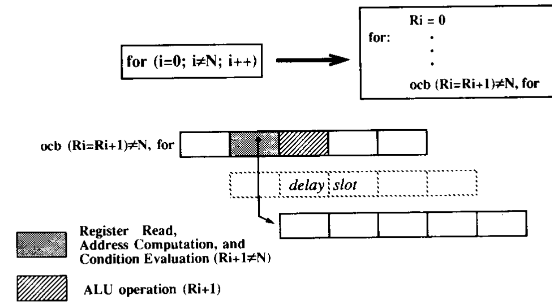

Fig. 6. Fast conditional branch.

branch. To reduce the negative effect of this delay, some techniques have been proposed. The one most used in RISC processors is the *delayed branch* [10], which consists of the execution of a fixed number of instructions following the branch—the delay slots—regardless of the result of the branch. The compiler must fill the slots with useful instructions that have no dependencies with the branch. In case not enough instructions are found, NOP's must be inserted. In our model, there is one delay slot after the branch. The cost (in cycles) of a control flow instruction sequence is the following:

$$C = 2 + u$$

where u is the number of useless instructions inserted in the delay slots. The two cycles correspond to the comparison and branch instructions.

B. Compare&Branch Instructions

A conditional branch is usually preceded by a comparison that sets the condition codes. The frequency of joint execution of both instructions has given rise to designing architectures with *compare&branch* instructions. Katevenis also observed that most conditional branches evaluate conditions for equality $(=, \neq)$ and any relation with zero $(a > 0, a \geq 0, \cdots)$ [11]. These comparisons are called fast comparisons, since their computation requires no carry propagation. Further studies have shown that the frequency of this type of comparisons within all the comparisons can be higher than 90% [9].

Compare&branch instructions can be combined with fast comparisons allowing, first, one cycle saving in the cost of the control flow sequence, and second, evaluation of the condition in the second stage of the pipeline (Register Read), thus not increasing the number of delay slots. The cost of a control flow instruction sequence is now

$$C = 1 + u.$$

Fig. 5 depicts a timing diagram of the execution of a *compare&branch* instruction. This mechanism has been used, for instance, in the MIPS R2000 architecture [12].

C. Fast Conditional Branches

The ALU scheme with a *Fast Adder-Comparator* presented in Section IV can be used to improve the efficiency of the execution of conditional branches. The new proposed approach consists of including a new instruction that combines an operation, a fast comparison, and a branch. We call it *operate&compare&branch (ocb)*. The functional description of this instruction is the following:

```
dst = src1 op src2
branch if (not) dst = src3
```

This instruction is a general case for other simpler instructions that have been already included in some architectures. This is the case of the ACB instruction (Add Compare and Branch) in the DEC VAX [13] or the DBcc instruction (Test Condition, Decrement and Branch) in the MC68020 [14]. In case of including *ocb* in a particular instruction set, *ocb* should be simplified or adapted to properly fit the implementation constraints of the architecture (i.e., instruction format).

Fig. 6 illustrates the use of the *ocb* instruction in our pipeline model. By using the $A\ op\ B = K$ comparator, the condition evaluation $(src1\ op\ src2 = src3)$, which does not require carry propagation, can be scheduled before the ALU yields the result (dst), such as in the case of fast comparisons explained in the previous technique. Now, the cost of a control flow instruction sequence is

$$C = u.$$

The execution frequency of conditional branches (15–30% of all the executed instructions) and equality comparisons make this approach attractive for significantly improving the performance of pipelined architectures.

VI. Conclusions

Carry propagation is the most limiting factor in the response time of parallel adders. The computation of conditions of the type $A + B = K$ has been conventionally performed by means of a full n-bit addition and equality comparison. This paper has proposed an alternative way to avoid the carry propagation involved in the addition. The elimination of the carry propagation drastically reduces the computation time.

This contribution introduces new possibilities for reducing the negative effect of conditional branches in pipelined architectures. An example has been presented in the scope of RISC architectures. Taking into account the execution frequency of branches, the inclusion of the *Fast Adder-Comparator* in a pipelined execution unit can significantly improve processor performance.

Acknowledgment

The authors would like to thank T. Lang and the referees for their technical contribution to the improvement of the paper. They would also like to thank the financial support provided by the Ministry of Education of Spain (CICYT, TIC 91-1036).

References

[1] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design.* New York: Wiley, 1979.

[2] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

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

[4] B. W. Y. Wei and C. D. Thompson, "Area–time optimal adder design," *IEEE Trans. Comput.*, vol. 39, no. 5, pp. 666–675, May 1990.
[5] M. Pomper *et al.*, "A 32-bit execution unit in an advanced nMOS technology," *IEEE J. Solid State Circuits*, vol. SC-17, no. 3, pp. 533–538, June 1982.
[6] J. L. Hennessy, "VLSI processor architecture," *IEEE Trans. Comput.*, vol. C-33, no. 12, pp. 1221–1246, Dec. 1984.
[7] A. Weinberger, "High-speed zero-sum detection," in *Proc. 4th. Symp. Comput. Arithmetic*, Oct. 1978, pp. 200–207.
[8] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Trans. Comput.*, vol. C-21, no. 12, pp. 1405–1411, Dec. 1972.
[9] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," in *Proc. 13th Annu. Symp. Comput. Architecture*, June 1986, pp. 396–403.
[10] D. A. Patterson and C. H. Séquin, "RISC I: A reduced instruction set VLSI computer," in *Proc. 8th Annu. Symp. Comput. Architecture*, May 1981, pp. 443–457.
[11] M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI.* Cambridge, MA: M.I.T. Press, 1985.
[12] J. Moussouris *et al.*, "A CMOS RISC processor with integrated system functions," in *Proc. COMPCON Spring 86*, Mar. 1986, pp. 126–131.
[13] Digital Equipment Corp., *VAX-11 Architecture Handbook*, 1981.
[14] Motorola Inc., *MC68020 32-Bit Microprocessor User's Manual.* Englewood Cliffs, NJ: Prentice-Hall, 1985.

Comments on "Tolerance of Double-Loop Computer Networks to Multinode Failures"

Jon M. Peha and Fouad A. Tobagi

***Abstract*— A previous Transactions on Computers paper[1] quantitatively evaluated the ability of a class of double-loop networks called forward-loop-backward-hop (FLBH) networks to tolerate node failures. Their approach was based on the enumeration of all possible sets of failures. In this correspondence, we prove that their results are incorrect, and demonstrate that it is difficult to solve this problem using an approach based on the enumeration of failure sets, suggesting other approaches may be preferable.**

***Index Terms*— Double-Loop network, fault tolerance, forward-loop-backward-hop (FLBH) network, network topology, reliability.**

I. Introduction

Because of its simplicity, the single-loop topology has been widely used as a basis for network architectures [1]. However, one liability of this topology is that any link or node failure disrupts communication. This problem can be alleviated by the addition of redundant links to a single-loop network. One such topology of particular interest is the forward-loop-backward-hop (FLBH) network [2]–[5]. In this topology, each node has two incoming and two outgoing links. If the nodes are assigned the numbers $\in \{0, 1, 2, \cdots, N-1\}$, node i has a "forward" outgoing link to node $(i+1) \bmod N$, and a "backward" outgoing link to node $(i-S) \bmod N$ for some $S \in \{1, 2, 3, \cdots, N-2\}$. An example is shown in Fig. 1. The subject paper[1] attempts to quantify the tolerance of an FLBH network to node failure, given N and S. This allows a network designer to determine whether an FLBH network can meet reliability requirements, and if so, for which values of S.

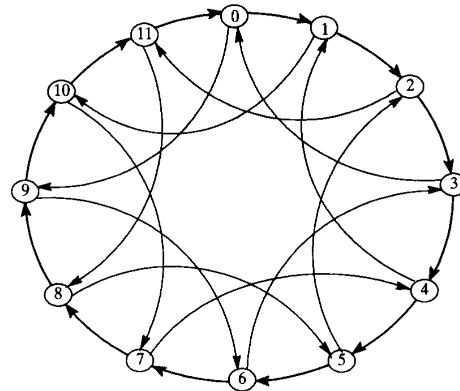

Fig. 1. An FLBH network, $N = 12, S = 3$.

As the measure of fault tolerance, the subject paper uses the "system working ratio" O^m, which is defined as follows. Assume the probability of node failure is equal and independent for all nodes. If a and b are randomly selected nodes ($a \neq b$), then O^m is the probability that a path exists from Node a to Node b given that Node a is up, Node b is up, and m nodes are down. Links are assumed to be reliable. Their approach is to group all N choose m unordered sets of node failures (i.e., failure sets) into classes, and, for each class, determine the number of source-destination pairs for which functioning paths exist. For example, if exactly two nodes, i and j, have failed, all failure sets for which the distance between failures is D can be grouped together because the number of communicating source-destination pairs is the same, where this distance between failures is D if $D = |i-j|$ or $D = N - |i-j|$. Tables are presented for both $m = 2$ and for $m = 3$ stating the number of communicating source-destination pairs for each class, where the classes are disjoint and their union comprises all possibilities. Based on these results, the authors of the subject paper extrapolate to evaluate O^m for $m > 3$.

These tables for $m = 2$ and $m = 3$, as well as the resulting conjecture for $m > 3$, are incorrect. In the next section we present two classes of counterexamples that directly contradict stated results even for the relatively simple cases in which $m = 2$ and $m = 3$, and the significance of these counterexamples is likely to increase with m. We then explicitly demonstrate that the conjecture for $m > 3$ is incorrect. We conclude with a discussion of the difficulties inherent to an approach to measuring fault tolerance based on enumeration, suggesting that other approaches may be preferable.

II. Counterexamples to the Subject Paper's Results

For $m = 2$, the subject paper rightfully concentrates on the case where the distance between node failures is $S + 1$, because any such failure set disrupts communication between some functioning

Manuscript received April 30, 1990; revised April 10, 1991.

J. M. Peha performed this research while at SRI International, and completed the writing while at Stanford University with support from a National Science Foundation Graduate Fellowship. He is now with Carnegie Mellon University, Department of Electrical and Computer Engineering, Pittsburgh, PA 15213.

F. A. Tobagi is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305. He was supported in part by the National Aeronautics and Space Administration under Grant NAGW-419.

IEEE Log Number 9105053.

[1] H. Masuyama and T. Ichimori, *IEEE Trans. Comput.*, vol. 38, no. 5, pp. 738–741, May 1989.

0018–9340/92$03.00 © 1992 IEEE