THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 62, No. 10, December 1983
Printed in U.S.A.

Parallel Fault Simulation Using Distributed Processing*

By Y. H. LEVENDEL,[†] P. R. MENON,[†] and S. H. PATEL[†]

(Manuscript received June 3, 1983)

This paper presents a method of performing fault simulation of digital logic circuits using a special-purpose computer with distributed processing. The architecture for true value simulation presented in an earlier paper can also be used for parallel fault simulation. The special-purpose computer consists of inexpensive microprocessors interconnected by either a time-shared parallel bus or a cross-point matrix. The cross-point matrix provides higher performance than the time-shared parallel bus. The performance of the proposed simulator is better by over two orders of magnitude than traditional logic fault simulation performed on a general-purpose computer. The power of the simulator is proportional to the number of microprocessors over a certain range.

I. INTRODUCTION

Fault simulation is an important part of the logic circuit design process. It is a means of determining the behavior of a logic circuit in the presence of each one of a predefined set of faults.

One of the most common uses of fault simulation is in determining the set of faults detected by a proposed test sequence, i.e., its fault coverage. Adequate fault coverage (usually greater than 90 percent of single stuck faults) is necessary to guarantee that the test sequence will detect most of the manufacturing defects.

---

* This paper is based upon material to be submitted by S. H. Patel in partial fulfillment of the requirements for the Ph.D. in Electrical Engineering at the Illinois Institute of Technology. [†] Bell Laboratories.

In test pattern generation, fault simulation is used to determine the faults that are detected by the tests already generated so they can be removed from consideration. A yet undetected fault is then selected as a target for the next test. Additional faults detected by a newly generated test may be determined by simulation. Thus, fault simulation is used frequently as a part of the test generation process.

Fault simulation is also used to construct fault dictionaries for fault isolation. Other uses of fault simulation include the evaluation of test point effectiveness and the evaluation of self-checking circuitry. Effective test points are essential for factory testing. It is much cheaper and easier to locate and repair failures during manufacture than it is in the field. Also, good self-checking circuitry makes it easier to isolate faults in the field.

Currently, fault simulation is carried out on large general-purpose computers. This method has seen some use in large-scale integrated (LSI)* designs, but suffers from excessive run time at current levels of integration. Its applicability to very large-scale integration (VLSI) is doubtful, at least in the manner that it is currently used.[1] There is a need for more sophisticated and cost-effective fault simulators as very large simulation time and costs will result when dealing with circuits of VLSI complexity (more than 100,000 gates on a single chip).

II. PARALLEL FAULT SIMULATION

A number of different algorithms have been developed for performing fault simulation efficiently on general-purpose computers. Among these the best known and widely used are the parallel,[2] deductive,[3] and concurrent[4] methods. All these methods attempt to simulate the effects of a number of individual faults simultaneously. This paper will consider the use of the parallel fault simulation algorithm in the special-purpose simulation hardware architecture developed in Ref. 5.

In parallel fault simulation the fault-free circuit and several different faulty circuits are processed simultaneously. The number of faulty circuits simulated in parallel is normally constrained by the number of bits in the host computer word. One bit of the computer word represents the signal value on a line in the fault-free circuit, while the remaining bits represent values on the same line in the presence of different single faults. Word-oriented operations performed on the host computer imply that the fault-free and faulty circuits are handled simultaneously and in exactly the same manner.

The output fault word of a gate is computed by simple word-oriented logic operations on the input fault words. The logic operations performed on the fault words correspond to the logic operation performed

* Acronyms and abbreviations are defined in the Glossary at the back of this article.

by the gate. Faults are injected using predefined masks. A stuck at 1 fault on a lead is injected by ORing the fault word with a mask containing a 1 in the bit position for the faulty value and 0's elsewhere. Similarly, a stuck at 0 fault can be injected by ANDing the fault word with a mask containing a 0 in the bit position for the faulty value and 1's elsewhere.

Two logic values are not sufficient for accurate logic simulation. Since each bit position in the computer word can represent only a logical 0 or 1, more than one bit per signal is required for multiple-value simulation.[6] In this case more than one word is required. For three-value representation two bits are required to represent each signal and, therefore, two computer words are required for representing a set of fault-free and faulty values. Since a pair of words are used to represent a signal, some sort of coding method is required to implement parallel simulation. A commonly used method of coding denotes one of the words as the 0-word and the other word as a 1-word. Let i_0 represent the ith bit in the 0-word and i_1 represent the ith bit in the 1-word. Then the $i_0i_1 = 01$ combination represents a logical 1, the $i_0i_1 = 10$ combination represents a logical 0, the $i_0i_1 = 00$ combination represents an unknown, and the $i_0i_1 = 11$ combination is unused. Simple word-oriented operations are still sufficient for performing the parallel simulation. This coding is the same as that in Ref. 7. Faults are injected in the 0-word and the 1-word using a 0-mask and a 1-mask, respectively. This injection is also done using simple logic operations. The method for simulating three logic values can be extended to any number of logic values by coding them using a sufficient number of bits.[8] Only three-valued simulation is considered in this paper.

The most widely used method of parallel simulation is the *event-driven* method. Event-driven simulation means that an element is not simulated unless there is a change in one of its input fault words. The main operations performed in an event-driven simulation are processing of active elements and scheduling changes to occur in future. The scheduling is done on a timing wheel. A *timing wheel* is a list structure in which events are chained together in the order they are to occur. In parallel fault simulation an element is considered to be *active* if the fault word associated with its output changes. The fault word is considered changed even if only one of its bits changes. All the values (i.e., the whole fault word) are propagated even if only one of the values changes.

Since the number of faults simulated at one time is restricted by the length of the host computer word, multiple passes through the simulator are necessary to simulate a large number of faults. It is possible to reduce the number of passes by using extra computer words and simulating more faults during one pass. For example, 64 computer words can be used to simulate (two-value simulation) 1024 faults on a computer with a word size of 16 bits. The string of values in the set of computer words representing the fault-free and faulty signal values on a line will be called the *value vector.* The configuration of the value vector is shown in Fig. 1. The value vector consists of L_p word pairs for three-value simulation. Two bits at the same position in the two words of a pair represent the signal value of one faulty circuit. Simulating L_p word pairs at a time is better than making L_p passes through the simulator since the overhead involved in the fanout search associated with each pass is saved. (However, as we will see in Section 6.1, some of these savings are lost due to increased activity as the number of faults per pass increases.) The number of words used in the value vector is usually constrained by the space requirements of the computer.

During simulation, operations are performed on value vectors by considering word pairs. Thus, the time required to perform an operation on the value vector is proportional to the number of word pairs used in the value vector. For example, if there are several word pairs in a value vector, then faults are injected one word pair at a time.

The whole value vector is considered active if any of the values in the vector changes. Furthermore, all the word pairs are propagated even if only one word pair changes. An element is considered *active* if the value vector associated with its output is active.

III. CONCURRENCY IN FAULT SIMULATION

At least three types of concurrencies exist in fault simulation of logic circuits. The first type of concurrency occurs in the actual simulated hardware, the second type occurs in the simulation algo-

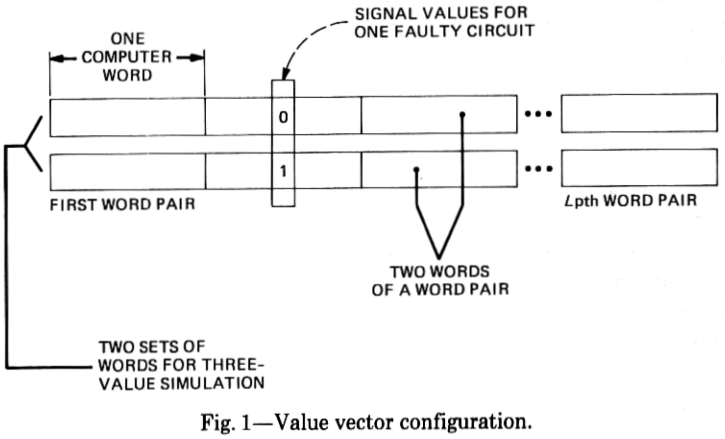

SIGNAL VALUES FOR ONE FAULTY CIRCUIT
ONE COMPUTER WORD
0
1
FIRST WORD PAIR
Lpth WORD PAIR
TWO WORDS OF A WORD PAIR
TWO SETS OF WORDS FOR THREE-VALUE SIMULATION

Fig. 1—Value vector configuration.

rithm, and the third type occurs in the form of fault activity. The first two types of concurrencies also occur in true value simulation of logic circuits and have been discussed in an earlier paper.[5]

The concurrency occurring in the actual simulated hardware can be called *logic circuit concurrency*. Utilizing this type of concurrency leads to distributed processing with identical processors performing identical tasks. The architecture for true-value simulation developed by Levendel et al.[5] and Denneau et al.[9–11] takes advantage of this type of concurrency.

The concurrency occurring in the simulation algorithm can be called *algorithm concurrency*. This concurrency is indirectly due to the concurrency occurring in the actual simulated hardware. Since several elements can be simultaneously active and a sequence of steps is to be performed for each active element, they can be processed in a pipeline fashion. Utilizing this type of concurrency leads to functional partitioning of tasks among several processors and a pipelined architecture. The architecture for true-value simulation developed by Barto and Szygenda[12] and Abramovici et al.[13] takes advantage of this type of concurrency.

For efficient fault simulation, a number of faults are simulated simultaneously in software-based simulators. This leads to *fault activity concurrency*, which can be utilized in special-purpose hardware for fault simulation.

This paper extends the architecture for true-value simulation described in Ref. 5 to fault simulation using the parallel method. The architecture takes advantage of the parallelism due to logic circuit concurrency and fault activity concurrency. The main difference between true-value simulation and fault simulation is in the algorithm executed by the individual processing units.

IV. SPECIAL-PURPOSE ARCHITECTURE

The simulator consists of one master and a number of slaves (processors) interconnected by a communication structure (Fig. 2). The communication structure is used as a medium for transferring data between the slaves and between the slaves and the master. The communication structure can be either a time-shared parallel bus or a cross-point matrix. The circuit to be simulated is partitioned into subcircuits and each subcircuit is simulated in a separate processor. Subcircuits in different processors become active as signal values proceed from the primary inputs to primary outputs. As simulation progresses, data are transferred between subcircuits as the logic values on the signal connections between two subcircuits change. These data are transported across the communication structure. Typical data sent across the data path consist of element information and changed value

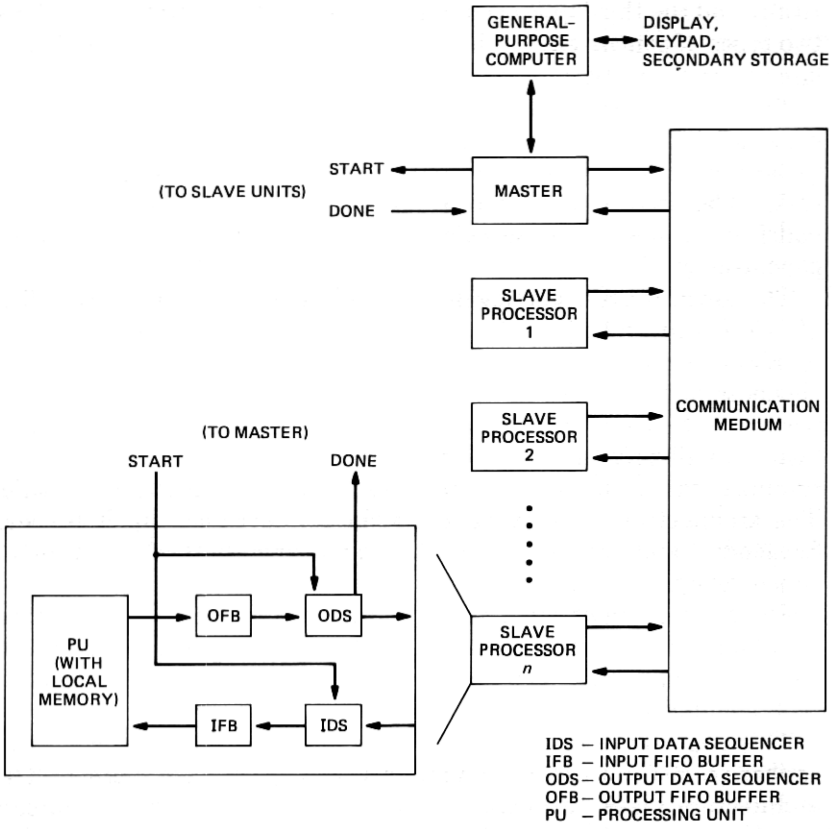


Fig. 2—Multiprocessor-based digital logic simulator.

vectors. The architecture of the simulator has been described in detail in the previous paper.[5] A summary of the architecture description is presented in Appendix A for completeness.

The overall architecture for true-value simulation is applicable to fault simulation since the algorithms for the two types of simulations are the same except that fault simulation requires:

1. Carrying of faulty signal values in addition to true-value signals (more than one word may be used to allow representation of a larger number of faulty signals)
2. Fault injection using masks
3. Multiple passes if the fault set is large.

The processing time per pass will be higher for fault simulation due to the extra processing required for injecting faults and manipulating multiple-word pairs in a value vector. The latter requirement applies only when the value vector contains more than one word pair. As far as communication between the processors is concerned, the data transferred between the processors are greater than those transferred for true-value simulation, since the signal values require one or more complete words in addition to a word for the element number.

V. ARCHITECTURE EVALUATION

The processing time per simulation cycle and the communication times using a time-shared parallel bus and a cross-point matrix are estimated first. These results are then used for selecting the communication structure. A multiple-bus communication structure is also discussed in this section.

5.1 Processing time t_p

The average number of active elements per processor during a simulation cycle for true-value simulation is given by kN/n, where N is the average number of active elements per simulation cycle during true-value simulation, n is the number of processors in the multiprocessor simulator, and k is the average unbalance factor representing the extra active elements per processor during a simulation cycle due to nonideal partitioning.[5] Ideal partitioning will cause an equal number of elements to be active in all the processors during all simulation cycles. However, because of some imbalance such as the fanout of all active elements during one simulation cycle not feeding equally into all the processors, some processors will have more active elements than the others during some simulation cycles. The average number of active elements per processor during a fault simulation cycle can be written as kN_f/n, where N_f is the average number of active elements per simulation cycle in one pass during fault simulation. The value of N_f is expected to be larger than N. Indeed, experimental runs on the Logic Analyzer for Maintenance Planning (LAMP) simulator[14] show that the overall activity (total number of active elements) during parallel fault simulation increases by a factor of about 2 for 16 faults per pass and by about 3.5 for 1024 faults per pass compared to true-value simulation. These results are averaged over several runs made using both combinational and sequential circuits with sizes ranging from 420 gates to 1912 gates. The number of faults simulated ranged from 1020 faults for the 420-gate circuit to 5046 faults for the 1912-gate circuit. The LAMP simulator is based on the deductive method. A mapping mechanism from deductive simulation activity to parallel simulation activity was implemented to predict the results for parallel simulation.

During one simulation cycle the following major operations occur in the given order:

1. Using the current list of events, list L_t of the timing wheel (Fig.

11 in Appendix B), update, and find fanout of the elements whose outputs changed during the current simulation cycle.

2. Using the next list L_{t+1} of the timing wheel, prepare external events to be transmitted to other processors for the next time interval.

3. From data in the Input FIFO Buffer (IFB) (sent by other processors), update and find fanout of the elements active during the current simulation cycle.

4. Evaluate the fanout of active elements (this includes fault injection).

5. Schedule on timing wheel elements whose output changes.

The detailed algorithm is given in Appendix C.

Let t_a be the time required to process one active element. The average processing time per processor during one simulation cycle is then given by:

$$t_p = \frac{kN_f}{n} t_a.$$

Assume a microprogrammable microprocessor (e.g., Am2900 series) for each slave unit processing unit (PU) and the following operation times (150 ns cycle time): memory-to-memory move = 1.2 μs, memory-to-register move = 0.6 μs, and memory-to-memory logical AND/OR operation = 1.5 μs. Using these major microprocessor operations, the execution time for each operation in the parallel simulation algorithm described in Appendix C can be estimated. For example, obtaining each fanout of an updated element (after the fanout list has been accessed) takes one memory-to-register instruction and moving the element number to the Output FIFO Buffer (OFB) takes one memory-to-memory instruction. The processing times per active element for the various steps of the algorithm are shown in Table I, where f_i is the average fan-in, f_o is the average fan-out, and L_p is the number of word pairs in the value vector.

The total processing time per element during one simulation cycle is the sum of all the expressions in Table I:

$$t_a = 9.6 + 8.7L_p + 3f_o + 3f_iL_p - \frac{(2.4 + 7.2L_p)}{f_o} \; \mu\text{s}.$$

Taking typical values of f_i and f_o to be 2 and an unbalance factor of $k = 1.1$, the processing time for a simulation cycle becomes:

$$t_p = (15.8 + 12.2L_p) \frac{N_f}{n} \; \mu\text{s}. \qquad (1)$$

5.2 Communication time t_c

The value of t_c will depend on the type of communication structure.

Table I—Simulation cycle timings for parallel simulation

Step	Expression (μs)
Update data from timing wheel	$\frac{(1.8 + 2.4f_o)}{f_o}$
Prepare external events for next time interval	$\frac{(f_o - 1)(6.6 + 3.6L_p)}{f_o}$
Update data from IFB	$\frac{(f_o - 1)(1.2 + 3f_o + 3.6L_p)}{f_o}$
Evaluate schedule	$2.4 + 1.5L_p + 3f_iL_p + \left(\frac{3.6}{f_o}\right)$

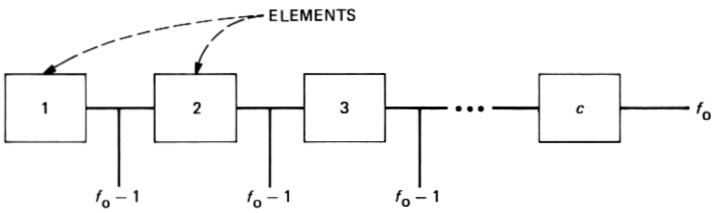


Fig. 3—An element string.

The two types of communication structures discussed in Ref. 5, namely the time-shared parallel bus and the cross-point matrix, will be considered here also. The partitioning algorithm discussed in the previous paper[5] partitions a circuit along its depth rather than its breadth. Since the signals in a circuit propagate in parallel, this places concurrent activities in different blocks. The same partitioning algorithm will be assumed here, since during fault simulation the signals still propagate in the same manner. The logic circuit to be simulated is partitioned into elements strings (see Fig. 3). The average number of communication events generated by one active element during a simulation cycle that have to be sent over the communication structure is:

$$e = \frac{[f_o + (c - 1)(f_o - 1)]}{c},$$

where f_o is the average fanout and c is the average number of elements in one element string. The typical value of f_o can be taken as 2, and for large circuits c is expected to be greater than 10. For $f_o = 2$ and $c = 10$, e will be equal to 1.1.

5.2.1 Time-shared parallel bus

The total communication time during a simulation cycle for true-value simulation is given by[5]:

$$t_{c(\text{bus})} = (n + 200)Ne \text{ ns}$$

This expression assumes one word of data to be transferred per active element. For fault simulation with w words of data to be transferred per active element the expression becomes:

$$t_{c(\text{bus})} = (n + 200)N_f ew \text{ ns.}$$

The number of words to be transferred is given by:

$$w = 1 + 2L_p,$$

where L_p is the number of word pairs per value vector used in simulation. One word is required to carry the element number and the $2L_p$ words carry the value vector. Taking the value of $e = 1.1$:

$$t_{c(\text{bus})} = (1.1n + 220)(1 + 2L_p)N_f \text{ ns.} \tag{2}$$

5.2.2 *Cross-point matrix*

In a cross-point matrix-based communication structure several processors can be simultaneously sending data to other processors. The total communication time during a simulation cycle for true value simulation is given by[5]:

$$t_{c(\text{matrix})} = \frac{200Nke}{n} + 50j \text{ ns,}$$

where j is the number of events for which the destination processor is found busy, i.e., the destination processor is communicating with some other processor. Once again this expression assumes one word of data transferred per active element. For w words of data to be transferred, the expression for fault simulation becomes:

$$t_{c(\text{matrix})} = \frac{200(N_f)kew}{n} + 50jw \text{ ns.}$$

For $k = 1.1$, $e = 1.1$, $w = 1 + 2L_p$, and $j = 0.1N_f/n$ (the channel is found busy for 10 percent of the transfer requests), the above expression can be rewritten as:

$$t_{c(\text{matrix})} = (247 + 494L_p)\frac{N_f}{n} \text{ ns.} \tag{3}$$

5.3 *Choice of communication structure*

The expressions for the processing time per simulation cycle per active element and the communication times per simulation cycle per active element for the bus and matrix structures are plotted in Fig. 4. The expressions are plotted for value vector length of one word (16 bits/word), i.e., $L_p = 1$.

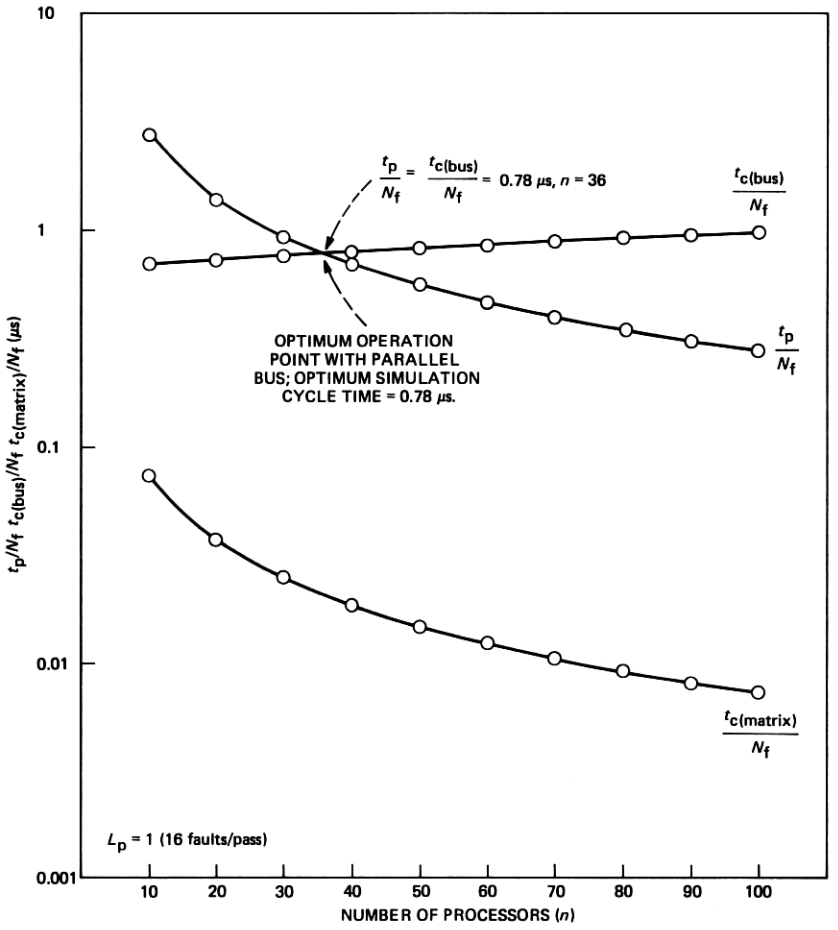


Fig. 4—Variation in processing and communication time.

The curves for the processing time and bus communication time for the parallel bus intersect at $n = 36$. The processing time is greater than the bus communication time for $n < 36$. Thus the processing time is the bottleneck. The processing time decreases as the value of n increases. For $n > 36$ the bus communication time becomes larger than the processing time and the bus communication time becomes the bottleneck. Therefore, using more processors than $n = 36$ will not speed up the simulation. For optimum performance $n = 36$, and the length of the simulation cycle per active element becomes $t_m = 0.78$ μs. Based on the parallel simulation algorithm given in Appendix C, the actual simulation cycle length per active element for a single processor can be estimated as $t_1 = 24$ μs. The multiprocessor fault simulator with a bus-based communication structure provides a speedup of 31 over the traditional single-processor logic fault simulator.

For further speedup a faster communication structure must be used. Figure 4 also shows the curve for the matrix communication time. This curve does not intersect with the curve for the processing time, and the communication time is always less than the processing time. The communication time will therefore never be a bottleneck. More processors can be added to speed up the simulator. For example, for $n = 100$ the speedup compared to the traditional single-processor logic fault simulator is 86 and for $n = 256$ the speedup is 220. The speedup of simulation is expected to be greater than two orders of magnitude for $n > 120$.

5.4 Multiple-bus communication structure

The results of the previous section show that for a given number of faults per pass, the time-shared parallel bus is useful only for up to a fixed value of n. For further speedup the cross-point matrix has to be used. However, the cross-point is not used up to its maximum capability. For example, at 16 faults per pass and $n = 100$, the simulation cycle length is $280N_f$ ns while the communication cycle length is $7.4N_f$ ns, i.e., the communication structure is used less than 3 percent of the time. A communication structure that is slower and cheaper than the cross-point matrix might prove more cost-effective. This will be true especially since the control for the cross-point is very complex and thus expensive.

A communication structure that provides a capacity in between that of the time-shared parallel bus and the cross-point matrix is the multiple-bus structure. It consists of a bus arbitrator and several parallel buses. The configuration of the multiple-bus structure and its interface to the Output Data Sequencers (ODSs) and Input Data Sequencers (IDSs) of the slave units are given in Fig. 5. When the ODS needs to send data, it sets the Request To Send (RTS) line high and puts the destination address on the address lines. The requesting ODS keeps the RTS line high until granted a bus. The bus arbitrator grants it the use of the communication medium when it finds an unused bus and determines that the requested destination is not busy. The bus arbitrator grants the bus by setting the Bus Grant line high. The data are switched through the bus selector switch to the available bus. At the other end of the bus there is a line selector from which the data are sent to the destination processor. The ODS sends out all the data present in its Output FIFO Buffer (OFB). The data received by the IDS of the destination unit are put in its Input FIFO Buffer (IFB). The ODS then sets the RTS line low. This releases the bus, which is

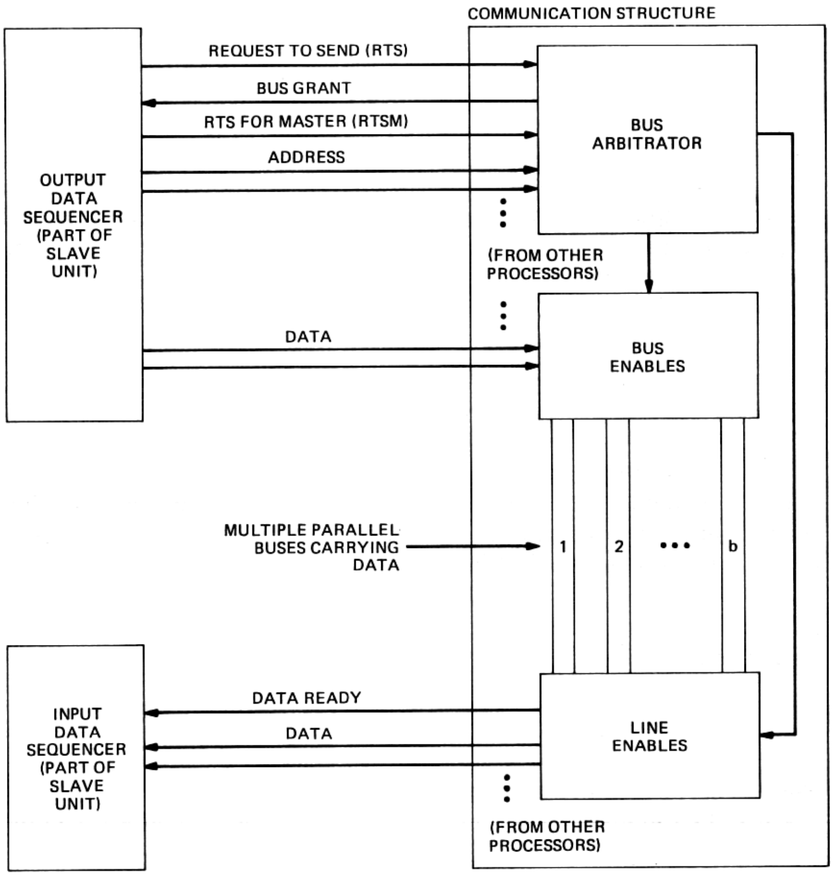


Fig. 5—Configuration of multiple-bus structure.

then granted by the bus control to another requesting slave or the master. All units have equal priority. The ODS will set the Request To Send line high again if it gets more data to transfer in the OFB.

The data sent out to a slave unit from another slave unit or the master consist of element information and changed value vectors. The data sent to a master consist of the address of the sending slave, element number (primary output or monitored point), and value vector. A separate line Request to Send to Master (RTSM) is used to address the master. When the destination is the master, the address lines from the ODS contain the sending slave unit address. This address together with the element number and value vector is stored in the master IFB by the master IDS.

To obtain an expression for the communication time for the multiple-bus structure, consider first of all the bus structure with only a single bus. For this case, the communication time will consist of the same components as that for time-shared parallel bus[5]:

$$t_{c(\text{mbus})} = (t_{\text{brg}} + t_{\text{ds}} + t_{\text{da}} + t_{\text{br}})(1 + 2L_p)(N_f)e,$$

where t_{brg} is the bus request and grant time, t_{ds} is the address and data setup time, t_{da} is the data acknowledge, and t_{br} is the bus release time. For the multiple-bus structure the bus request and grant time, t_{brg}, will be greater than in the parallel bus since extra checking has to be done before a bus is granted. The bus arbitrator will have to determine if a bus is available and if available then it has to further determine if the requested destination is busy. Assuming these extra actions double the time required for the bus request and grant time and the other times remain the same: $t_{\text{brg}} = 200$ ns, $t_{\text{ds}} = n$ ns, $t_{\text{da}} = 50$ ns, $t_{\text{br}} = 50$ ns, and $e = 1.1$. Each transaction across the multiple-bus has to wait for a bus to be granted. As more buses are added, the transactions can occur in parallel. Assuming the number of parallel buses is much smaller than the number of processors, the probability of the destination processor being busy will be small. The decrease in the total communication time will then be proportional to the number of buses. The expression for the total communication time for the multiple-bus communication structure becomes:

$$t_{c(\text{mbus})} = \frac{(1.1n + 330)(1 + 2L_p)N_f}{b}, \tag{4}$$

where b is the number of parallel buses.

Let n_o be the number of processors at the optimum operation point, where $t_p = t_{c(\text{mbus})}$. If the simulator operates with the number of processors not equal to n_o, then the speed of simulation will be lower. An expression for n_o can be derived by equating eqs. (1) and (4):

$$n_o = -150 + 150\left(1 + b\left(\frac{0.49L_p + 0.64}{2L_p + 1}\right)\right)^{0.5}.$$

This expression is plotted for various values of b with $L_p = 1$ in Fig. 6. It can be seen that higher performance is available with the multiple-bus structure compared to the single time-shared parallel bus for $b \geq 2$.

The implementation of the multiple-bus structure is expected to be less expensive than that of the cross-point matrix. The complexity of the communication structures and thus their cost is proportional to the number of switch points. For the cross-point matrix, the number of cross-points is proportional to the square of the inputs, i.e., n^2. For the multiple-bus structure two points have to be connected to establish a connection and thus the number of switch points is $2bn$. The cost of the multiple-bus structure will be less than that of the cross-point

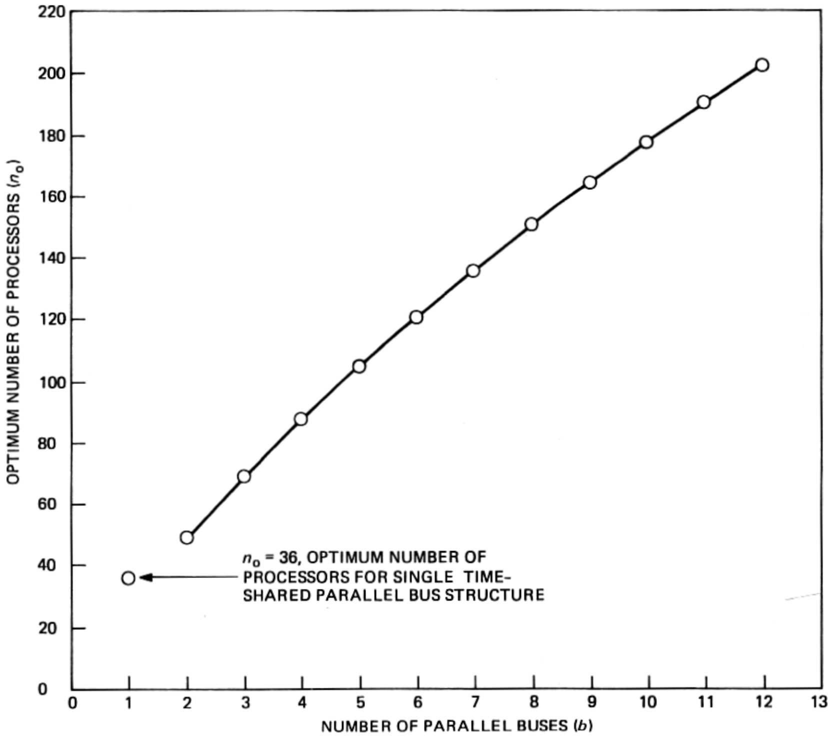


Fig. 6—Variation in the optimum number of processors with the number of parallel buses.

matrix as long as $2b < n$. As we saw in the analysis done earlier this will always be the case. For $b = 5$ and $n_o = 105$, the cost of the multiple-bus structure will be an order of magnitude lower than that of the cross-point matrix. Furthermore, it must also be noted that physical switching is not necessary in the case of the multiple-bus structure; it is cheaper to use logic enables for connecting an ODS and an IDS to a bus. This will result in even lower cost when compared with the cross-point matrix.

VI. EFFECT OF NUMBER OF FAULTS PER PASS

When simulating a large number of faults per pass in parallel simulation, several pairs of words carrying the true and faulty values (L_p) must be manipulated per active element. All the faulty values in the words can be considered together as a vector. All the faulty values are evaluated even if only one faulty value is active.

The behavior of the multiprocessor fault simulator, using the parallel fault simulation algorithm, will be investigated for variations in the number of faults simulated per pass. Comparisons will be made between 16, 32, 64, 256, and 1024 faults per pass, assuming a processor word length of 16 in all the cases.

Only the time-shared parallel bus and the cross-point matrix are discussed. The communication structure with multiple buses is not considered since the results for the time-shared parallel bus will apply, except for a scaling factor.

6.1 Simulator with time-shared parallel bus

The expressions for processing time per simulation cycle, t_p, and the parallel bus communication time per simulation cycle, $t_{c(bus)}$, derived earlier in eqs. (1) and (2) are used to analyze the effects of variations in the number of faults per pass. It was seen earlier that the optimum operation point for the simulator occurs when the processing time and communication time are equal, i.e., $t_p = t_{c(bus)}$. An expression for the number of processors required to meet this condition for a given length of value vector (L_p) can be derived by equating the expressions for t_p and $t_{c(bus)}$. Let n_o be the number of processors at the optimum operation point and t_o be the length of the processing and communication cycles at the optimum operation point. Values of n smaller than the optimum n_o cause the processing to be a bottleneck, while larger values of n cause communication to be a bottleneck. Equating eqs. (1) and (2) yields the following expression for n_o, the number of processors required for optimum operation:

$$n_o = -100 + 100\left(\frac{3.11L_p + 2.44}{2L_p + 1}\right)^{0.5}.$$

Let α be the number of faults per pass. Then $\alpha = 16L_p$ assuming 16 bits per word. The above expression for n_o can be rewritten as a function of the number of the faults α, as:

$$n_o(\alpha) = -100 + 100\left(\frac{0.19\alpha + 2.44}{0.125\alpha + 1}\right)^{0.5}. \tag{5}$$

For $n = n_o(\alpha)$, the processing time per simulation cycle and the bus communication time per simulation cycle both reduce to:

$$t_o(\alpha) = (15.8 + 0.76\alpha)\frac{N_f(\alpha)}{n_o(\alpha)}. \tag{6}$$

For a simulator with optimum number of processors, $n_o(\alpha)$, the total time required to simulate a fixed number of faults is obtained by multiplying the time required for processing one active element ($t_o(\alpha)/N_f(\alpha)$) by the total number of active elements during the simulation ($N_T(\alpha)$):

$$T(\alpha) = \frac{t_o(\alpha)}{N_f(\alpha)} N_T(\alpha).$$

The total number of active elements during simulation is given by:

$$N_T(\alpha) = N_f(\alpha) \times \text{(number of simulation cycles per pass)} \times \text{(number of passes)}.$$

Substituting the expression for $t_o(\alpha)$ given in eq. (6):

$$T(\alpha) = (15.8 + 0.76\alpha)\frac{N_T(\alpha)}{n_o(\alpha)}. \qquad (7)$$

Define the *simulation time ratio* as the ratio of the total time required to simulate a set of faults with α faults per pass to the total time required to simulate the given set of faults with 16 faults per pass, i.e., $T(\alpha)/T(16)$. Using a value of $n_o(16) = 36$ (eq. 5), the expression for the simulation time ratio is given by:

$$\frac{T(\alpha)}{T(16)} = \frac{(20.3 + 0.98\alpha)}{n_o(\alpha)} \frac{N_T(\alpha)}{N_T(16)}. \qquad (8)$$

The values of the simulation ratio are first calculated theoretically and then compared with the experimental results.

6.1.1 Theoretical simulation time ratio

Let *simulation activity*, $N_T(\alpha)$, refer to the number of active elements during all passes of a simulation. The simulation activity can be expected to be inversely proportional to the number of faults per pass:

$$\frac{N_T(\alpha)}{N_T(16)} = \frac{16}{\alpha}.$$

For example, the expected simulation activity at 32 faults per pass will be half the simulation activity of 16 faults per pass since the number of passes needed will be halved. The theoretical expression for the simulation time ratio becomes:

$$\frac{T(\alpha)}{T(16)} = \frac{(20.3 + 0.98\alpha)}{n_o(\alpha)} \frac{16}{\alpha}. \qquad (9)$$

Note that the theoretical simulation time ratio is independent of the simulation activity. Table II gives the variation of the optimum number of processors, n_o, and the variation of the simulation time ratio as a function of the number of faults per pass, α. The theoretical results show that the simulation time ratio, and thus the total simulation time, decreases as the number of faults per pass increases.

Also, it is interesting to note that for 16 faults per pass the operation

Table II—Theoretical simulation time ratio

Faults per Pass, α	Optimum Number of Processors, n_o	Simulation Time Ratio, $\frac{T(\alpha)}{T(16)}$
16	36	1.0
32	32	0.81
64	29	0.72
256	26	0.65
1024	25	0.64

Table III—Experimental simulation time ratio

Faults per Pass, α	Simulation Activity, $N_T(\alpha)$	Optimum Number of Processors, n_o	Simulation Time Ratio, $\frac{T(\alpha)}{T(16)}$
16	17586	36	1.0
32	9374	32	0.86
64	5204	29	0.84
256	1787	26	1.06
1024	689	25	1.6

peaks at 36 processors, while for 1024 faults per pass the operation peaks at 25 processors. As the number faults per pass increases, the point of optimum operation occurs for a slightly smaller number of processors. This is because the time required to transfer the increased data is more than the time required to process the increased data.

6.1.2 Experimental simulation time ratio

The values of $N_T(\alpha)$ averaged over several experimental runs made on the LAMP simulator[13] (with a mapping algorithm from deductive simulation to parallel simulation as discussed in Section 5.1) are given in Table III. Also shown in Table III are values for the simulation time ratio derived from eq. (8).

The experimental simulation time ratio, $T(\alpha)/T(16)$, is plotted together with the theoretical simulation time ratio in Fig. 7.

It is interesting to note that as the number of faults per pass increases, the experimental simulation time ratio falls below 1.0 initially, i.e., the simulation speeds up. After a certain point, the simulation time ratio then starts increasing and goes above 1.0. The fastest simulator is a 64 faults per pass simulator with 29 processors. On the other hand, the theoretical simulation time ratio always stays below 1.0 and keeps decreasing as the number of faults per pass increases. The difference in the experimental and theoretical results can be

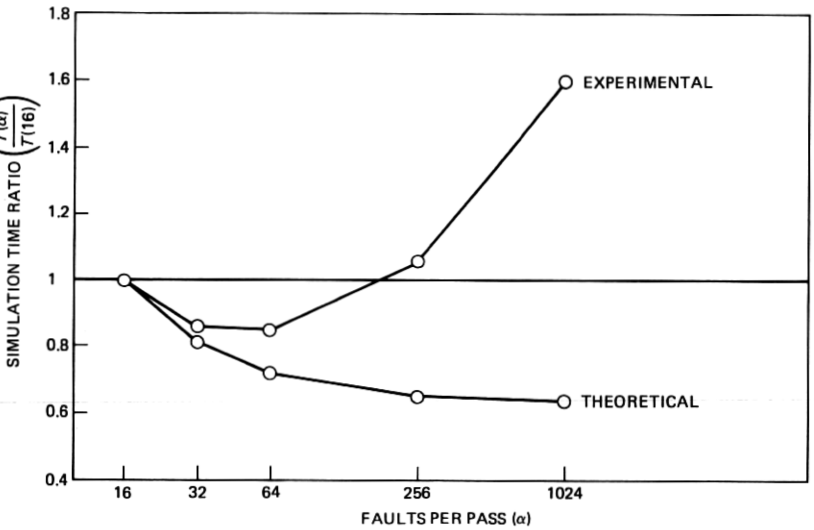


Fig. 7—Experimental and theoretical simulation time ratio for variation in number of faults per pass—a time-shared parallel bus.

explained by examining the variation in simulation activity as the number of faults per pass changes.

The curve for the theoretical simulation time ratio in Fig. 7 shows that the simulation speed increases as the number of faults per pass increases. This is as expected, since increasing the number of faults per pass decreases the number of passes and thus the fanout search and related processing time. In practice, however, there is extra simulation activity due to longer value vectors and this tends to increase the processing time. As more faults are simulated per vector, i.e., as the value of α gets larger, the simulation activity during the simulation will be higher than the theoretical. For example, as shown earlier, the theoretical activity at 32 faults per pass will be half the simulation activity at 16 faults per pass. However, the experimental simulation activity at 32 faults per pass will be more than the expected half. This is because the active faults in two value vectors at 16 faults per pass will not always directly map into one value vector at 32 faults per pass. Any active fault in the value vector will cause simulation activity even if the good value does not change. The effect of this is to cause extra schedulings. This increase in schedulings can be represented by an effective increase in the length of the simulation cycle and the number of simulation cycles. The runs made on the LAMP simulator show that most of the increase is in the length of the simulation cycle (i.e., more computation during the simulation cycle). The expected simulation activity and the actual simulation activity obtained from runs made using the LAMP simulator are given in Table IV.

For 32 faults per pass, the simulation activity is only 1.07 times that theoretically expected. Thus the increase in processing time due to this extra activity is not substantial and the overall speed of simulation is higher due to the greater savings in the fanout search processing. For 1024 faults per pass, the simulation activity is 2.51 times that theoretically expected. In this case, the increase in processing time due to the extra activity is substantial compared to the savings obtained in the fanout search processing. This results in lowering the overall speed of simulation when compared with 16 faults per pass.

In summary, for the multiprocessor fault simulator with a parallel-bus-based communication structure, the fastest simulation speed occurs for 64 faults per pass and 29 processors. Note, however, that the number of processors required for 64 faults per pass is greater than the number of processors for 1024 faults per pass. Decreasing the number of processors for 64 faults per pass to 25 yields the total simulation time of 13,413 μs. This still favors the 64 fault per pass simulator over the 1024 fault per pass simulator.

6.2 Simulator with cross-point matrix

The expressions for the processing time per simulation cycle, t_p, and the cross-point matrix communication time per simulation cycle, $t_{c(matrix)}$, derived earlier in eqs. (1) and (3) are applicable to variations in the number of faults per pass. The increase in simulation activity caused by simulating more faults per pass will increase both the processing time and the communication time for the cross-point matrix. However, adding one word pair to the value vector will cause an increase in the communication time that is only 4 percent of the increase in the processing time. Thus, the communication time will always be less than the processing time. The cross-point matrix provides sufficient communication capacity for the parallel fault sim-

Table IV—Effect of multiple passes on simulation activity

Faults per Pass, α	Experimental Simulation Activity, $N_T(\alpha)$	Theoretical Simulation Activity, $N_T(\alpha)$
16	17,586	17,586
32	9374	8793
64	5204	4396
256	1787	1099
1024	689	275

ulator and does not cause a communication bottleneck. For faster fault simulation, more processors can be added.

The variation of the simulation time ratio as a function of the number of faults per pass will be investigated to obtain the optimum number of faults per pass. Since the processing time dominates, the total time required to simulate a set of faults with α faults will be equal to the total processing time. Using eq. (1):

$$T(\alpha) = (15.8 + 0.76\alpha)\frac{N_T(\alpha)}{n}.$$

The simulation time ratio is given by:

$$\frac{T(\alpha)}{T(16)} = (0.56 + 0.027\alpha)\,\frac{N_T(\alpha)}{N_T(16)}. \tag{10}$$

The experimental and theoretical simulation time ratios are plotted in Fig. 8. As was the case for the time-shared parallel bus, the experimental simulation time ratio increases after 64 faults per pass. This is because the time required to process the increased simulation activity is greater than the time saved in the fanout search overhead associated with each pass. For the multiprocessor fault simulator with a cross-point-matrix-based communication structure, the optimum number of faults per pass is 64. Using a different value for the number of faults per pass will decrease the speed of simulation. Note that the

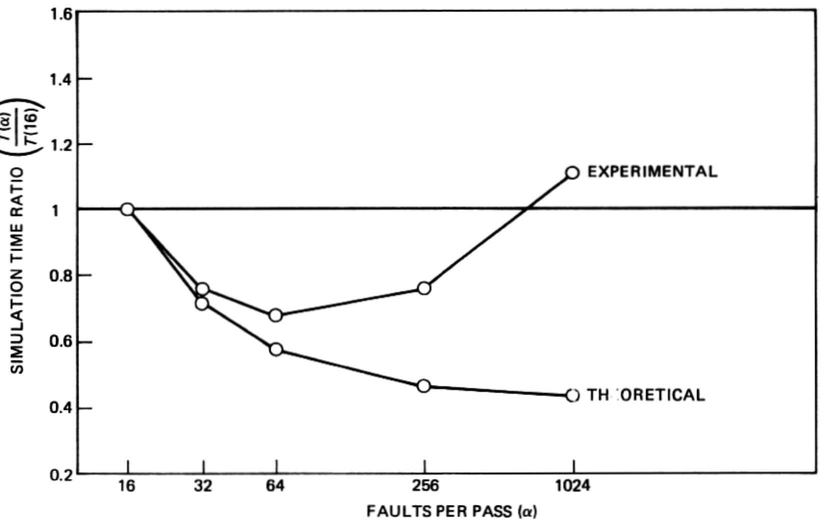


Fig. 8—Experimental and theoretical simulation time ratio for variation in number of faults per pass—cross-point matrix.

number of processors, n, does not affect the simulation time ratio. More processors can be added to obtain greater speed.

When compared with the parallel bus the cross-point matrix provides greater speed for any value of n greater than the optimum n for the parallel bus. For example, for 64 faults per pass, the cross-point matrix can be made faster than the parallel bus by selecting $n > 29$.

VII. SUMMARY

In this paper we presented a special-purpose logic fault simulator based on the parallel simulation method. The simulator is expected to be two orders of magnitude faster than traditional logic fault simulators implemented on general-purpose computers. For both the time-shared parallel bus and the cross-point matrix, the simulator performs the best at 64 faults per pass. Decreasing the number of faults per pass slows down the simulation due to fanout search and other overhead required for every pass. Increasing the number of faults per pass also slows down the simulation due to the increase of simulation activity.

When the parallel bus is used, the power of the simulator can be increased over a certain range by increasing the number of slaves. The power of the simulator can be further increased by using the cross-point matrix.

The application of the special-purpose simulator to other fault simulation methods is being investigated currently.

REFERENCES

1. D. F. Barbe (ed.), "Very Large Scale Integration (VLSI), Fundamentals and Applications," Berlin, Germany: Springer-Verlag, 1980.
2. E. W. Thompson and S. A. Szygenda, "Digital Logic Simulation in a Time-Based, Table-Driven Environment: Part 2. Parallel Fault Simulation," Computer, *8-3* (March 1975), pp. 38–49.
3. D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Trans. Computers, *C-21*, No. 5 (May 1972), pp. 464–71.
4. E. G. Ulrich and T. G. Baker, "Concurrent Simulation of Nearly Identical Networks," Computer, *7-4* (April 1974), pp. 39–44.
5. Y. H. Levendel, P. R. Menon, and S. H. Patel, "Special-Purpose Logic Simulator Using Distributed Processing," B.S.T.J., *61*, No. 10, Part 1 (December 1982), pp. 2873–2909.
6. M. A. Breuer, "A Note on Three Valued Logic Simulation," IEEE Trans. Computers, *C-21*, No. 4 (April 1972), pp. 399–402.
7. S. G. Chappell, "Automatic Test Generation for Asynchronous Digital Circuits," B.S.T.J., *53*, No. 8 (October 1974), pp. 1477–1503.
8. Y. H. Levendel and P. R. Menon, "Fault Simulation Methods—Extensions and Comparison," B.S.T.J., *60*, No. 9 (November 1981), pp. 2235–58.
9. M. M. Denneau, "The Yorktown Simulation Engine," 19th Design Automation Conference, Las Vegas, Nevada, June 14–16, 1982, pp. 55–9.
10. E. Kronstadt and G. Pfister, "Software Support for the Yorktown Simulation Engine," 19th Design Automation Conference, Las Vegas, Nevada, June 14–16, 1982, pp. 60–4.
11. G. Pfister, "The Yorktown Simulation Engine: Introduction," 19th Design Automation Conference, Las Vegas, Nevada, June 14–16, 1982, pp. 51–4.

12. R. Barto and S. A. Szygenda, "A Computer Architecture for Digital Logic Simulation," Electronic Engineering, September 1980, pp. 35–66.
13. M. Abramovici, Y. H. Levendel, and P. R. Menon, "A Logic Simulation Machine," 19th Design Automation Conference, Las Vegas, Nevada, June 14–16, 1982, pp. 65–73.
14. S. G Chappell, C. H. Elemendorf, and L. D. Schmidt, "LAMP: Logic Circuit Simulators" B.S.T.J., *53*, No. 8 (October 1974), pp. 1451–76.
15. N. D. Phillips and J. G. Tellier, "Efficient Event Manipulation, A Key to Large Scale Simulation," IEEE 1978 Semiconductor Test Conference, Cherry Hill, New Jersey, October 31–November 2, 1978, pp. 266–73.
16. J. G. Vaucher and P. Duval, "A Comparison of Simulation Event List Algorithms," Comm. ACM, *18-4* (April 1975), pp. 233–30.

APPENDIX A

Architecture Description

A.1 Introduction

The simulator consists of one master and a multiplicity of slaves (processors) interconnected by a communication structure (see Fig. 2). The communication structure can be either shared or dedicated. The circuit to be simulated is partitioned into subcircuits and each subcircuit is simulated in a separate processor. The partitioning is such that the number of simultaneously active subcircuits (processors) is maximum and the number of simultaneously active elements in each subcircuit (processor) is minimum while keeping the communication from being a bottleneck.

The circuit to be simulated is initially modeled in the general-purpose computer, partitioned and loaded into the slave memories. The general-purpose computer performs all functions of input, output, and user interactions. The simulation is carried out in the multiprocessor simulator.

The simulator can be programmed to output intermediate results to the general-purpose computer. It also can be interrupted by the general-purpose computer for intermediate results. The user can ask for information about a simulation run while it is in progress (e.g., the status of a variable) and make certain run-time decisions like continuing simulation, applying extra input vectors, or stopping. After simulating each vector input, the simulation results and any other user requested information are sent to the general-purpose computer. User-requested information typically includes output values of elements (monitored points) at specific simulated times or under some other specified conditions. The general-purpose computer formats this information for suitable presentation to the user.

A.2 Multiprocessor operation

At the beginning of each simulation cycle the master sends primary input values (if any) to the appropriate slaves using the communication structure. The master then issues a start signal to the slaves. This signal informs the slaves to start processing for the next simulation cycle. During the processing of a simulation cycle a slave unit may generate data for the other slaves or the master. The data are sent to the destination slave or the master using the communication structure. Only data for the subsequent time interval are transferred between the slaves to reduce the amount of information sent over the communication structure, thus minimizing the communication overhead. Therefore, the scheduled time is not sent.

Each slave informs the master when it has finished processing and transferring data for the current simulation cycle. When all slaves have informed the master about their completion of processing for the current simulated time interval, and also the master has finished transferring any primary input values scheduled for the next simulated time interval to the slaves, the master issues a start signal to the slaves for the next simulation cycle.

There are two signal lines between each slave unit and the master. The master signals the slaves using a START signal and the slaves signal the master using the DONE line. The START line from the master initiates processing for the next simulation cycle. The DONE line will become one when all the slaves have finished processing for the current simulation cycle.

A.3 Slave unit

Each slave unit consists of a processing unit (PU), an input FIFO buffer (IFB), an output FIFO buffer (OFB), an input data sequencer (IDS), and an output data sequencer (ODS).

The slave unit PUs perform the actual fault simulation. The PU stores any data it has for other PUs or the master in the OFB. The ODS makes a request for the communication structure if there are any data to be transferred from the OFB. The ODS of the slave, if granted the use of the communication structure, takes data from the OFB and sends it over the communication structure to other slaves or the master. The data are received by the IDS of the destination slave or the master. Any data received by an IDS are put in its IFB. End of Data (EOD) flags are used to separate data streams since a PU can be writing new data to the OFB before its ODS has finished transferring current data, and similarly, an IDS can be receiving new data in the IFB before its PU has finished reading current data.

The slave unit PU operation can be described in terms of two essentially concurrent processes, namely the simulation cycle (execution of simulation algorithm) and the communication cycle (communication of events). Data generated during a simulation cycle are transferred as they are generated in a corresponding communication cycle (see Fig. 5). The algorithm executed by the Slave Unit PU is similar to the traditional parallel simulation algorithm used on general-purpose computers except that it is modified to operate on a data structure distributed over several processors (see Appendix C).

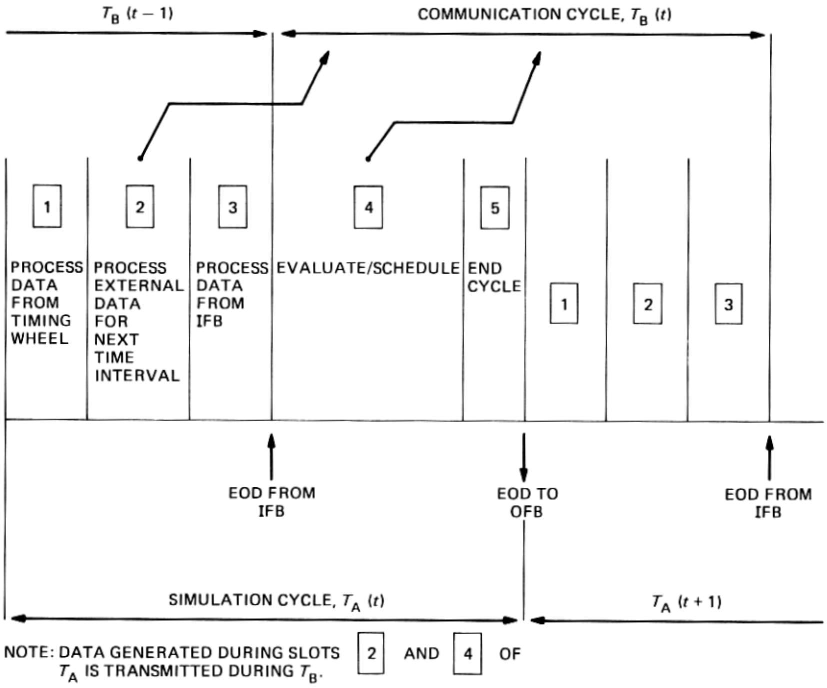
A communication cycle is the period in between two START commands issued from the master. This cycle is phased with respect to the simulation cycle as shown in Fig. 9.

A.4 Master processor

The master processor is the interface between the general-purpose computer and the simulator. Its main functions are to keep track of simulated time, keep the slaves in synchronism, supply the slaves with primary input values, and gather the primary output values from the slaves. The configuration of the master is similar to that of a slave unit and is shown in Fig. 10. It consists of a central processing unit (CPU) with some local memory, an IFB, an OFB, an IDS, and an ODS.

A.5 Communication structure

The communication structure is used as a medium for transferring data between the slaves and between the slaves and the master. Either a shared or a dedicated structure can be used for the multiprocessor



T_B $(t-1)$
COMMUNICATION CYCLE, T_B (t)
1
2
3
4
5
PROCESS DATA FROM TIMING WHEEL
PROCESS EXTERNAL DATA FOR NEXT TIME INTERVAL
PROCESS DATA FROM IFB
EVALUATE/SCHEDULE
END CYCLE
1
2
3
EOD FROM IFB
EOD TO OFB
EOD FROM IFB
SIMULATION CYCLE, T_A (t)
T_A $(t+1)$
NOTE: DATA GENERATED DURING SLOTS 2 AND 4 OF T_A IS TRANSMITTED DURING T_B.

Fig. 9—Relationship between a simulation cycle and communication cycle.

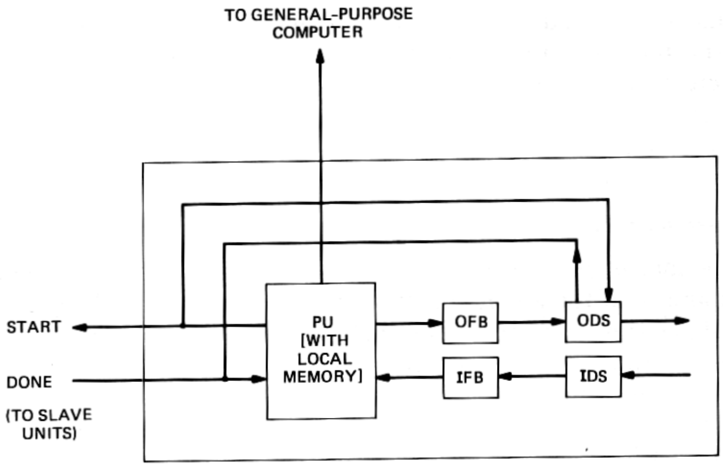


Fig. 10—Master configuration.

simulator. The details of a communication structure based on time-shared parallel bus and the cross-point matrix are discussed in detail in Ref. 5.

APPENDIX B

Data Structure for Parallel Method

The following data tables are used by each PU for its operation (see also Fig. 11):

1. Element table—This table contains the interconnection data and signal values for the circuit. For each element it contains the value vector, type, delay, fan-in list pointer and corresponding fan-in list, internal fanout list pointer and fanout list, external fanout list pointer and external fanout list. The value vector consists of a word pair. The first bit of each word of the word pair represents the good machine, while the remaining bits represent faulty machines. Three values can be represented: 0, 1, and X. If multiword parallel simulation is required, then additional word pairs are used to represent more faulty machines. The internal fanout list pointer and corresponding fanout lists give the fanout that remains in the subcircuit. The external fanout list pointer and corresponding fanout lists give fanout that goes to subcircuits located in other slaves. An element may have only internal fanout, only external fanout, or both internal and external fanout. Note that storing the external fanout takes up more space than storing an internal fanout, since both the destination processor address and element index have to be stored for the external fanout.

2. Activity list—This list is used to keep track of active elements during a simulation time interval. These elements are to be evaluated.

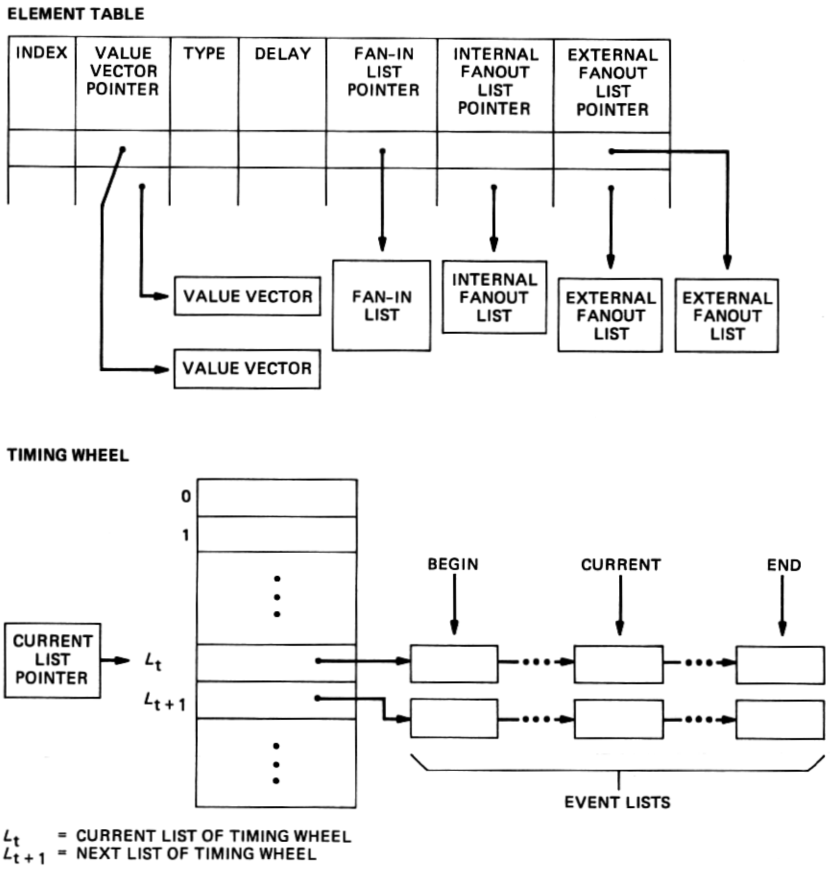


Fig. 11—Data tables for PU operation (parallel simulation).

3. Timing wheel—This data area contains the events that are scheduled in the future. A large amount of work has been done in this area.[15,16]

APPENDIX C

Parallel Simulation Algorithm

```
1. Update data from timing wheel
    for each event in list $L_t$ of timing wheel
      begin
        update value vector pointer in Element Table;
        for each fanout $f$ of updated element
          if activity flag of $f$ not set {
            set activity flag;
            put f on activity list;
            }
      end
   deallocate space on Timing Wheel;
2. Prepare external events for next time interval

   for each event in list $L_{t+1}$ of Timing Wheel
   begin
        if element has external fanout {
           for each external fanout {
              move destination processor address to OFB;
              move element number to OFB;
              for each word of value vector associated with event
                 move word to OFB;
           }
           if element does not have internal fanout also
              deallocate space on Timing Wheel;
        }
      end
3. Update data from IFB

   enable interrupts from IFB;
   for each interrupt received from IFB
      begin
        if event is not EOD flag {
           for each word of value vector associated with event
              move word to Element Table;
           for each fanout f of updated element
              if activity flag of f not set {
                 set activity flag;
                 put f on activity list;
              }
        }
        else {
           disable interrupts from IFB;
           reset activity flags in Element Table;
           go to step 4 (Evaluate/Schedule);
        }
      end
   (The EOD flag signifies end of data present in the IFB for the
   current simulation cycle. The EOD flag is loaded by the IDS
   when it receives a START signal from the master.)
4. Evaluate/schedule

   for each entry in activity list
      begin
```

```
        inject faults, if any, on inputs;
        for each set of words associated with input value vectors
        of element
          begin
            calculate corresponding output word;
            if output word changed from previous value
              set change flag;
          end
        inject faults, if any, on outputs;
        if change flag set or faults injected {
          schedule event on Timing wheel;
          if (element delay is 1 && element has external fanout)
            for each external fanout {
                move destination processor address to OFB;
              move element number to OFB;
              for each word of value vector associated with event
                move word to OFB;
          }
      }
    end
5. End cycle
   store EOD flag in OFB;
   increment t;
   go to STEP 1.
```

APPENDIX D

Glossary

BIU	bus interface unit
CPU	central processing unit (part of a master unit)
EOD	end of data (flag)
IDS	input data sequencer
IFB	input first in first out (FIFO) buffer
L_p	number of word pairs used in parallel simulation
LAMP	logic analyzer for maintenance planning
LSI	large-scale integration
L_t	current list of timing wheel
N	average number of active elements per simulation cycle in true value simulation
N_T	total number of active elements during all passes of a simulation
N_f	average number of active elements per simulation cycle in fault simulation with 16 faults per pass
ODS	output data sequencer

OFB—output FIFO buffer
PU—processing unit (part of a slave unit)
T—time required to simulate a given set of faults
c—average number of elements in an element string
f_i—average fan-in of an element
f_o—average fanout of an element
j—number of events for which the channel is found busy in cross-point matrix
k—imbalance factor due to nonideal partitioning
n—number of processors in the multiprocessor simulator
n_o—number of processors required for optimum operation of multiprocessor simulator
t_l—length of simulation cycle for a single processor simulator
t_a—time required to process one active element
t_{br}—bus release time for a parallel bus structure
t_{brg}—bus request and grant time for a parallel bus structure
t_c—total communication time during one simulation cycle
$t_{c(bus)}$—total communication time during one simulation cycle for a single parallel bus structure
$t_{c(matrix)}$—total communication time during one simulation cycle for a matrix structure
$t_{c(mbus)}$—total communication time during one simulation cycle for a multiple parallel bus structure
t_{da}—data acknowledge time for a parallel bus structure
t_{ds}—address and data setup time for a parallel bus structure
t_m—length of simulation cycle for a multiprocessor simulator
t_o—average processing time, t_p, for number of processors $n = n_o$
t_p—average processing time per processor during one simulation cycle, where average processing time consists of the time required to process all active elements and schedule resulting events
VLSI—very large-scale integration
w—number of words to be transferred across the communication structure for one active event

AUTHORS

Ytzhak H. Levendel, B.S.E.E., 1971, Technion-Israel; M.S.C.S., 1974, The Weitzman Institute of Science; Ph.D., 1976, University of Southern California; Bell Laboratories, 1976—. Mr. Levendel has done research in fault diagnosis and until lately was involved in the development of a logic and test design aid system. He is now a Supervisor in 5*ESS*.™ Member, IEEE, Eta Kappa Nu.

Premachandran R. Menon, B.S. (Electrical Engineering), 1954, Banaras Hindu University; Ph.D. (Electrical Engineering), 1962, University of Washington; Bell Laboratories, 1963—. Mr. Menon has done research in switching theory and fault diagnosis and is currently involved in the development of a logic simulation system. Recipient of the Distinguished Technical Staff Award; member, IEEE.

Suresh H. Patel, B.E. (Electrical), 1975, University of Zambia; M.S.E.E., 1976, Illinois Institute of Technology; Association of American Railroads, 1977–1981; Bell Laboratories, 1981—. At the Association of American Railroads Mr. Patel was involved in hardware/software design, development and testing of a multi-microcomputer-based instrumentation system for freight trains. He was also involved in development of analytical and computer models on track circuit. At Bell Laboratories he is a member of the Processor Design Department. Mr. Patel was a part-time instructor at Illinois Institute of Technology during 1977–1978.