THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 10, December 1982
Printed in U.S.A.

Special-Purpose Computer for Logic Simulation Using Distributed Processing

By Y. H. LEVENDEL, P. R. MENON, and S. H. PATEL*

(Manuscript received March 20, 1982)

This paper presents the architecture of a special-purpose computer for logic simulation using distributed processing. The architecture is based on the utilization of inexpensive microprocessors interconnected by a communication structure. The communication structure is cross-point based for simple evaluations and time-shared parallel bus based for functional evaluations. Analysis is carried out to show that the performance of the proposed simulator is better by over two orders of magnitude than traditional logic simulation carried out on a general-purpose computer. Also, the power of the simulator is proportional to the number of slave processors over a certain range.

I. INTRODUCTION

The logic circuit simulator is an important component of a CAD system. It is used to predict logic circuit operation and performance under normal and faulty conditions. The application of the logic circuit simulator can be divided into two major areas: verification of new logic hardware designs and fault analysis of these designs.

As an evaluation tool, it can be used to verify the logical correctness of new hardware designs. Other information that can be obtained using a logic simulator include timing and signal propagation characteristics, and race and oscillatory circuit conditions. If the results of simulation indicate an unsatisfactory design, i.e., the circuit does not perform as expected, then changes can be made to the design. The design can be reevaluated using the simulator. After a number of iterations, a satisfactory design ready for committing to hardware should result. A logic circuit simulator used as above is known as a true value simulator.

* This paper is based on 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.

Note that the results of true value simulation can be used for diagnosing faulty equipment using the guided probe technique.[1]

Most logic circuit simulators also have fault-simulation capability. This capability can be used to determine fault coverage by a proposed test sequence, production of a fault dictionary, evaluation of test-point effectiveness, and evaluation of self-checking circuitry. The behavior of a circuit under fault conditions can also be investigated. Fault analysis can be used to investigate initialization and fault-induced races, and to perform timing analysis under specific fault conditions. For fail-safe circuitry, selected faults can be inserted in the circuit and the effect of these observed on the outputs by simulation. If a forbidden output is obtained under some fault, then the circuit design must be changed.

Figure 1 shows the general environment of a logic circuit simulator. The circuit to be analyzed is modeled using a circuit-description language. This language describes the connectivity and behavior of the circuit. The modeling information typically includes element type (gate or functional), associated delays, and interconnection data. Once the data structure of the model is set up in the logic circuit simulator, simulated inputs are applied either dynamically (at prescribed times) or statically (after the circuit is stabilized). Fault simulation is performed using one of several methods: one fault at a time, parallel, concurrent, or deductive.[2-4] The simulated output is recorded either in a plot form (true value) or tabular form (true value and fault simulation).

Currently, most digital circuits are simulated on large general-purpose computers. This method of simulation is complex and expensive to operate and maintain.[5] There is a need for more sophisticated and cost-effective simulators as we get into the VLSI era. Very large simulation time and costs will result when dealing with circuits of VLSI complexity (more than 100,000 gates on a single chip).

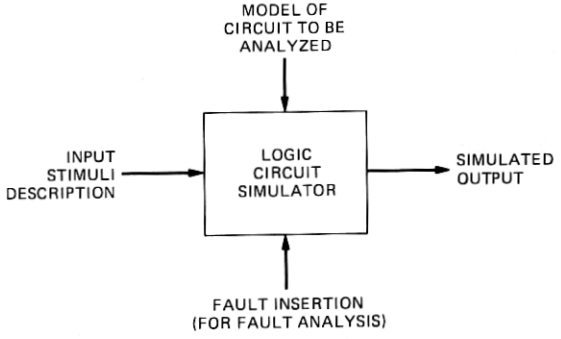


Fig. 1—Operating environment of a logic circuit simulator.

II. SPECIAL-PURPOSE SIMULATION HARDWARE

Existing logic circuit simulators are implemented in software that is executed on a general-purpose computer. To date, a large amount of work has been invested in optimizing this software. For further improvements in the performance of the logic circuit simulator, the hardware on which the simulation software executes must be optimized. With the advent of low-cost microcomputers, the development of special-purpose logic simulation hardware becomes attractive. Possible benefits are higher speeds, lower costs, and greater flexibility (for example, better integration in a test station).

Recently there has been some interest in developing special-purpose logic simulators. Barto and Szygenda have developed special-purpose simulation hardware based on distributed processing.[5] More recently, Abramovici et al. have presented a special-purpose architecture based on pipelining and concurrency.[6] Both approaches use dedicated processors for performing specific tasks. A special-purpose logic simulation machine using parallel processing (the Yorktown Simulation Engine) has been built by IBM.[7-9]

Concurrency allows simultaneous processing of several parallel events leading to a reduction in overall processing time. There are at least two types of concurrencies present in the simulation of logic circuits. One type of concurrency occurs in the simulation algorithm and the other in the actual simulated hardware.

The first type of concurrency can be called *algorithm concurrency*. In logic circuit simulation a number of operations have to be performed during a simulated time interval. Simulated time consists of discrete points in time (approximated to the nearest integer) at which changes in logic values on signal lines can occur. A simulated time interval is the time between two such consecutive discrete points. A simulated time interval is also sometimes referred to as a *time frame*. A *simulation cycle time* is the time required to carry out the processing during a simulated time interval. Typical operations carried out during a simulation cycle include determining current events, updating values at source, determining fanout, updating values at fanout, evaluating elements, and scheduling resulting events. An *event* is a change in logic value on a signal line. Scheduling an event is marking it to occur at some time in the future. Consider several elements being evaluated and several resulting events being scheduled during a simulation cycle. In traditional simulation, the elements are evaluated and the resulting events scheduled sequentially. No two operations are performed simultaneously. One can take advantage of the inherent concurrency by noting that after an element has been evaluated and while the resulting event is being scheduled, the evaluation of another element can be concurrently started. An average number of 80 such concurrent events

were observed per simulated time interval during the simulation of a 4000-gate circuit.[6] This concurrency appears in the simulation algorithm. The architecture proposed by Abramovici et al. takes advantage of this concurrency to some extent. The main ingredient of this solution is *functional partitioning* of the tasks to several microprocessors.

The concurrency can also be viewed from the point of view of the logic circuit. Concurrent events occur during a simulation cycle because of the way electrical signals propagate in the logic circuit. Several elements may be activated at the same time because signal propagation occurs simultaneously along several paths in the actual hardware. If the elements that become active at the same time are processed by different processors simultaneously, then the overall simulation time will be reduced. This type of concurrency can be called *logic circuit concurrency*. Its main ingredient is *distributed processing* among several processors, all of which are executing the same algorithm.

This paper describes the architecture of a special-purpose logic simulation machine designed to take advantage of the parallelism caused by concurrent activity of signals in a circuit. The system is essentially a processing network based on an interconnection of low-cost microcomputers. The circuit to be simulated is partitioned into subcircuits and each subcircuit is simulated in a separate microcomputer. Thus, several microcomputers can be simultaneously simulating several elements activated by parallel signals. This simulator is different from those proposed in the literature (see Refs. 5 and 6) in that the multiple processors do not perform dedicated tasks. Also, the modularity of the simulator proposed in this paper allows easy increase of computational power.

III. MULTIPROCESSOR OPERATION ENVIRONMENT

The operation environment of the multiprocessor digital logic simulator is shown in Fig. 2. The general-purpose computer acts as a preprocessor at the beginning of simulation and as a postprocessor at the end of simulation.

At the beginning of simulation, the circuit to be simulated is modeled on the general-purpose computer. The data structure is then loaded into the multiprocessor simulator. The loading problem is not discussed in this paper. After setting up the environment for the multiprocessor simulator, the general-purpose computer requests the simulator to start.

The simulation is carried out in the multiprocessor simulator. The simulator can be programmed to output intermediate results automatically to the general-purpose computer. The simulator can also be interrupted by the general-purpose computer for intermediate results. The user can ask for information about a simulation run while it is in

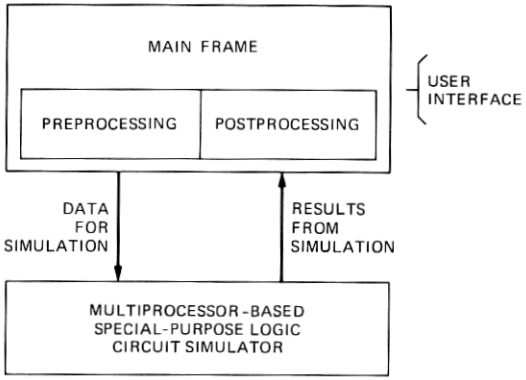


Fig. 2—Preprocessing and postprocessing for the multiprocessor-based special-purpose logic simulator.

progress (e.g., the status of a variable) and make certain run-time decisions such as continue simulation, apply extra input vectors, or stop. At the end of circuit simulation, the final simulation results and any other user-requested information is 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.

IV. ARCHITECTURE DESCRIPTION

The multiprocessor simulator consists of processors p_1 through p_n. The circuit to be simulated is partitioned into blocks a_1 through a_n. The signal connections between two blocks a_i and a_j are designated as b_{ij}. Each block a_x is then mapped into processor p_x as a subcircuit c_x. Figure 3 shows two blocks a_i and a_j mapped to processors p_i and p_j, respectively, as subcircuits c_i and c_j. The blocks are not necessarily clusters. That is, elements in a block can be from disjoint portions of the circuit. The signal connections b_{ij} between blocks a_i and a_j are mapped in a data path d_{ij} between processors p_i and p_j.

During simulation the subcircuits c_i and c_j are simulated independently. Different subcircuits become active as signal values proceed from the primary inputs to primary outputs. As simulation progresses, data will have to be carried between subcircuits c_i and c_j as the logic values on the signal connections between the two subcircuits change. This data is transported across the data path d_{ij}. Typical data sent across the data path consists of changed logic values.

The architecture of the multiprocessor simulator proposed in this paper is shown in Fig. 4. The simulator consists of a communication

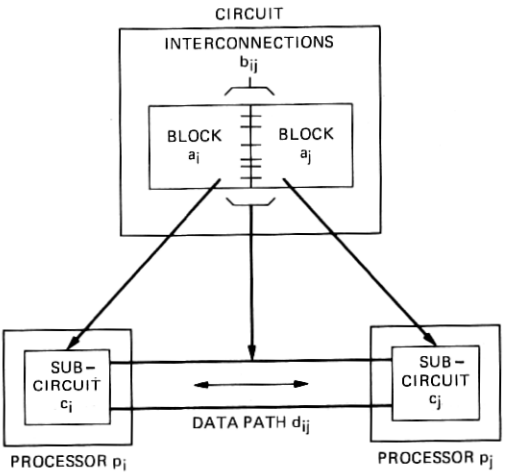


Fig. 3—Mapping of partitions a_i and a_j to processors p_i and p_j.

structure (communication medium and its associated control) connected to a master, several simple evaluators for simulating gate level blocks, and several functional evaluators for simulating functional blocks. A cross-point matrix is used to interconnect the master and the simple evaluators. The functional evaluators are connected to the cross-point matrix through a bus interface unit and a parallel bus. It is shown in Section VI that the speed of a cross-point matrix is required for transferring data between the simple evaluators. A parallel bus provides sufficient speed for functional evaluators. Note that if only

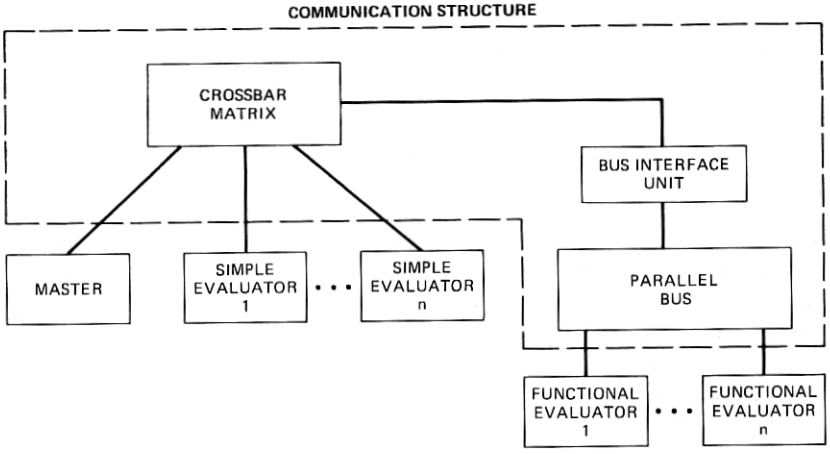


Fig. 4—Architecture of proposed multiprocessor simulator.

simple evaluations are to be carried out, then the bus interface unit, the parallel bus, and the functional evaluators can be removed. If only functional evaluations are to be performed, then the bus interface unit, the cross-point matrix, and the simple evaluators can be removed.

V. MULTIPROCESSOR IMPLEMENTATION

The configuration of the multiprocessor-based digital logic simulator is shown in Fig. 5. The simulator consists of one master and a multiplicity of slaves interconnected by a communication structure (communication medium and its associated control). The implementation of the simulator allows the use of either a shared or a dedicated communication structure. The master has local memory for its use. Each slave consists of a processing unit (PU) with its associated memory. The master and the slaves also each have two FIFO buffers and two data sequencers for interfacing to the communication structure (see Sections 5.1 and 5.2).

The processors p_i and p_j shown in Fig. 3 correspond to the slaves s_i and s_j in the multiprocessor simulator. The subcircuits c_i and c_j reside in the memories of the slaves s_i and s_j. The interconnections b_{ij} between blocks a_i and a_j are mapped into the data path d_{ij} that constitutes part of the communication structure.

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 is sent to the destination slave or the master using the communication structure. Data transferred between the slaves consists of scheduled events for the next time frame. A *scheduled event* is a change in logic value on a signal line scheduled to occur at some time in the future. Only data for the subsequent time interval is transferred between the slaves in order to reduce the amount of information sent over the communication structure, and thus the communication overhead. The scheduled time does not have to be sent. Data transferred from the slaves to the master consist of primary output values and user-requested information.

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.

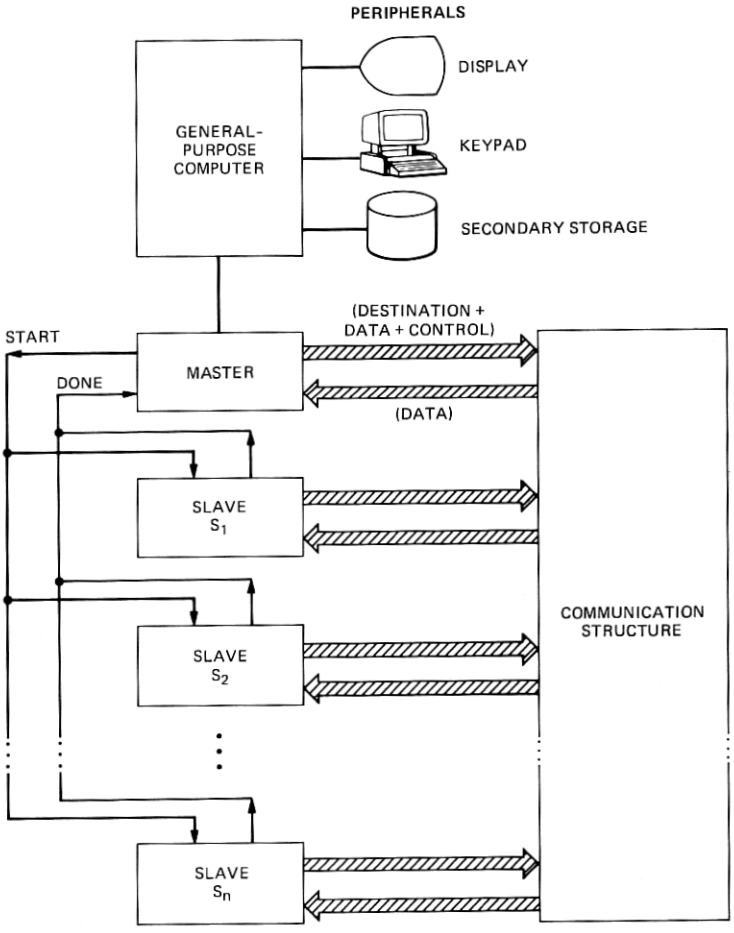


Fig. 5—Implementation of a multiprocessor-based digital logic simulator.

The different sections of the multiprocessor are discussed in greater detail below.

5.1 Slave unit

The slave unit configuration is shown in Fig. 6. The PU is a general-purpose 16-bit microprocessor. The input and output data sequencers can be either specially designed logic circuits or commercially available single-chip microcomputers. The FIFO buffers are commercially available devices.

The slave unit PUs perform the actual element/function evaluation and event scheduling. As noted previously, the partitioning of the logic circuit to be simulated is done on a general-purpose computer. Each

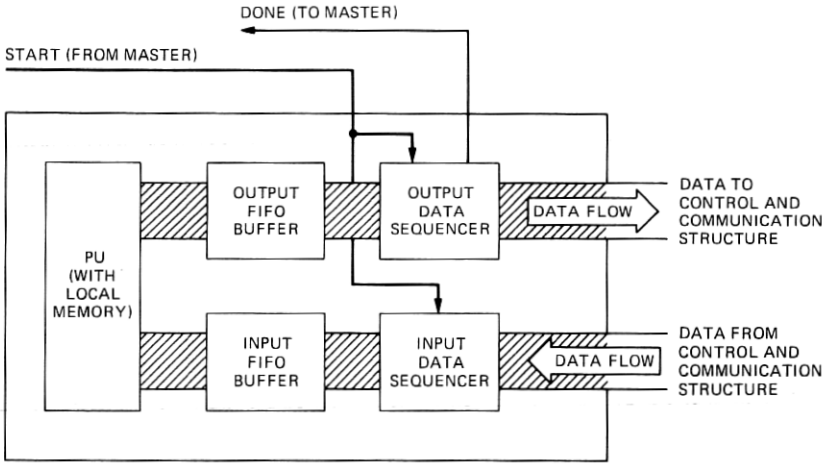


Fig. 6—Slave unit configuration.

PU contains a block of the logic circuit to be simulated. The simulation algorithm resides in the PU memory; all slaves contain identical algorithms.

Each slave uses an Output Data Sequencer (ODS) to transfer data out and an Input Data Sequencer (IDS) to receive data from the other slaves or the master via the communication structure. In a slave unit, the PU and the data sequencers are isolated from each other by means of FIFO buffers. Thus, the slaves can send and receive data independently of whether the PU is active or not.

The PU stores any data it has for other PUs or the master in the Output FIFO Buffer (OFB). The ODS makes a request for the communication structure if there is any data to transfer from the OFB. The ODS of the slave, if granted the use of the communication structure, takes data (scheduled values for slaves and primary output values and user-requested information for the master) from the OFB and sends it over the communication structure to other slaves or the master. The data is received by the IDS of the destination slave or the master (described in Section 5.2). Any data received by an IDS is put in its Input FIFO Buffer (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.

There are two signal lines between a 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 **DONE** line will become active when all the slaves have finished processing.

The **START** line from the master initiates the processing for the next PU processing cycle. This signal causes the IDS to load an EOD flag in the IFB of all slaves. At the beginning of each simulation cycle, the PU monitors the IFB for data from other slaves and the master for the current simulation cycle. The EOD flag marks the end of data from other slaves and the master for the current simulation cycle. When the PU reads this flag, it starts evaluation for the current simulation cycle. The **START** signal from the master also informs the slave ODS to start sending out any data to be used during the next simulated time cycle from the OFB.

At end of the simulation cycle the PU loads an EOD flag in the OFB and starts preprocessing for the next simulation cycle. When the ODS encounters the EOD flag in the OFB, it has finished transferring data for the current simulation cycle. The ODS informs the master using the **DONE** line.

5.1.1 PU operation

The following data tables are used by each PU for its operation (see also Fig. 7):

(*i*) *Circuit Description Table.* This table contains interconnection data for the subcircuit. For each element it contains the value, type, delay, input status word pointer and corresponding status fields, internal fanout list pointer and corresponding fanout lists, and external fanout list pointer and corresponding fanout lists. The input status word pointer and the corresponding status fields give the signal values on the fanin lines. The internal fanout list pointer and corresponding fanout lists give the fanout which remain in the subcircuit. The external fanout list pointer and corresponding fanout lists give fanout which go 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.

(*ii*) *Activity List.* This list is used to keep track of active elements during a simulation time interval. These elements are to be evaluated.

(*iii*) *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.[10,11]

The 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). During one simulation cycle the following operations occur in the given order:

1. Update line values from current list L_t of timing wheel.

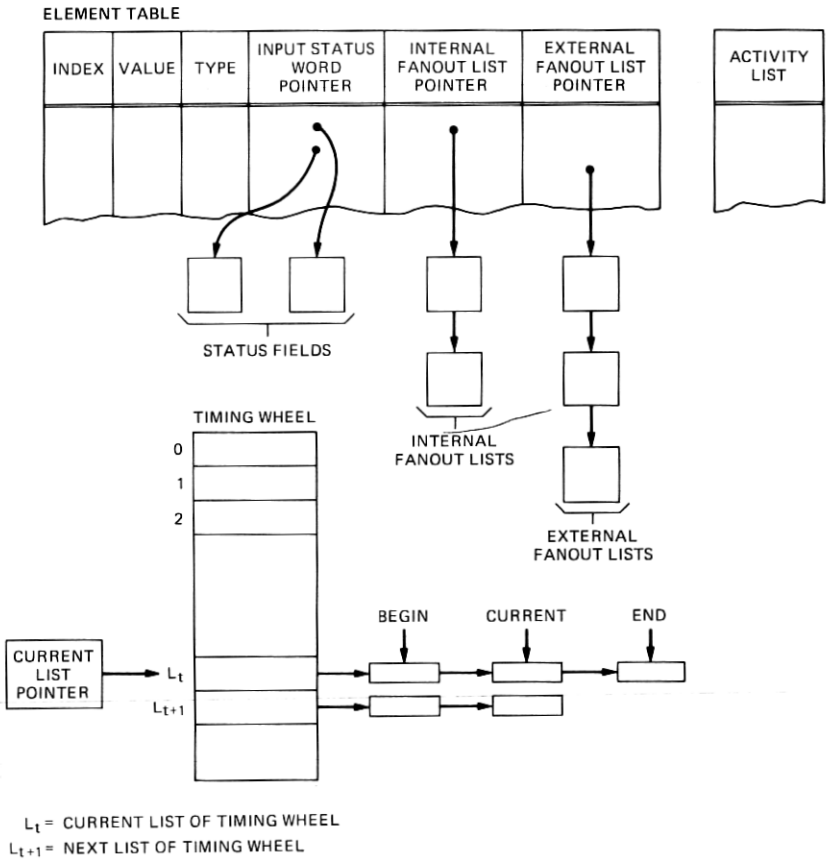


Fig. 7—Data tables for PU operation.

2. Find fanout and put active elements in activity list.

3. From next list L_{t+1} of timing wheel, for each entry that is an external fanout node, store scheduled event in OFB. Remove entry from timing wheel if it does not have any internal fanout also.

4. Update line values from IFB until EOD flag is received. 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.)

Note: Any user requests received are stored for later processing.

5. Find fanout and put active elements in activity list.

6. Evaluate elements in activity list.

7. For active elements whose output changes, if delay of element is one and it is an external output node, schedule change in OFB.

8. If test in step 7 fails, schedule change on timing wheel.

9. Gather any user-requested information and store it in OFB.
10. Store EOD flag in OFB.
11. Increment current list pointer of timing wheel to the list of next time.

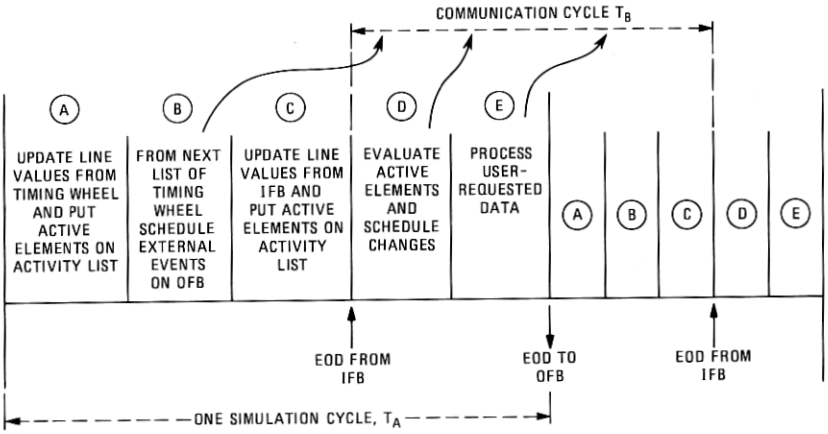
From steps 7 and 8 it can be seen that events on an external fanout are scheduled on the timing wheel of the processor in which the event occurred except in the case where the external fanout is going to be active in the next time interval. In this way the scheduled time does not have to be transmitted with the scheduled event, thus saving communication overhead.

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. 8. Note that the end of a communication cycle is an appropriate point for the multiprocessor simulator to stop for processing any interrupt requests from the general-purpose computer or sending out intermediate results. The master can issue the start for the next simulation cycle by sending a **START** command to the slave data sequencers after it satisfies all requests.

All the slave unit PUs contain the same software. Note that this algorithm is similar to the one used in traditional logic simulators except that the operations are sequenced differently.

5.1.2 Operation of slave data sequencers

The function of the data sequencers (IDS and ODS) is to transfer data from the OFB to the communication structure and from the communication structure to the IFB.



NOTE: DATA GENERATED DURING SLOTS (B), (D) AND (E) OF T_A IS TRANSMITTED DURING T_B.

Fig. 8—Relationship between a simulation cycle and the communications cycle.

The IDS and the ODS have some local memory. The IDS uses this memory to store any data it receives from the outside in case the IFB overflows. The ODS requires this memory only in certain circumstances. A case for its use is given Section 5.3.2.

The communication protocol between the data sequencers and the communication structure depends on the type of communication structure. This is discussed in detail in Section 5.3.

5.2 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. It also stores any user-requested monitored point values sent to it by the slaves.

The configuration of the master is similar to that of a slave unit and is shown in Fig. 9. It consists of a central processing unit (CPU) with some local memory, an input FIFO buffer (IFB), an output FIFO buffer (OFB), an input data sequencer (IDS), and an output data sequencer (ODS). The master is connected to the slave units through the communication structure. The master initiates processing for the next simulation cycle by issuing a **START** command on its signal line. When the slaves finish processing for the current simulated time interval, they inform the master through the **DONE** signal.

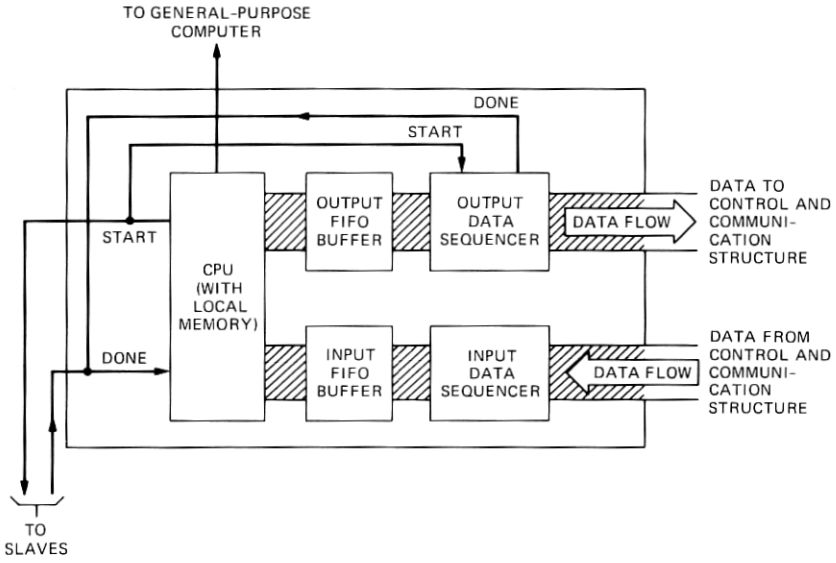


Fig. 9—Master configuration.

The master sends and receives data to and from the slaves using the communication structure. The interface between the master and the communication structure is similar to that between the slaves and the communication structure. The master stores any data (primary input values or user requests for monitored point values) it has for the slaves for the next simulation cycle in its OFB. The **START** signal which goes to the slaves also informs the ODS of the master to start transferring out data. The ODS encounters an EOD flag in the OFB when it has transferred all the data from the OFB. The ODS informs the master that it has finished sending out data by setting the **DONE** line. The IDS receives data from the slaves and puts it in the IFB.

Figure 10 shows the master, slave unit PU, slave unit ODS, and slave unit IDS operations during a simulation cycle.

5.3 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 simulator. Two types of communication structures will be considered here, namely the time-shared parallel bus and the cross-point matrix. Each case is treated separately below. The criteria for selecting the type of communication structure are given in Sections VI and VII.

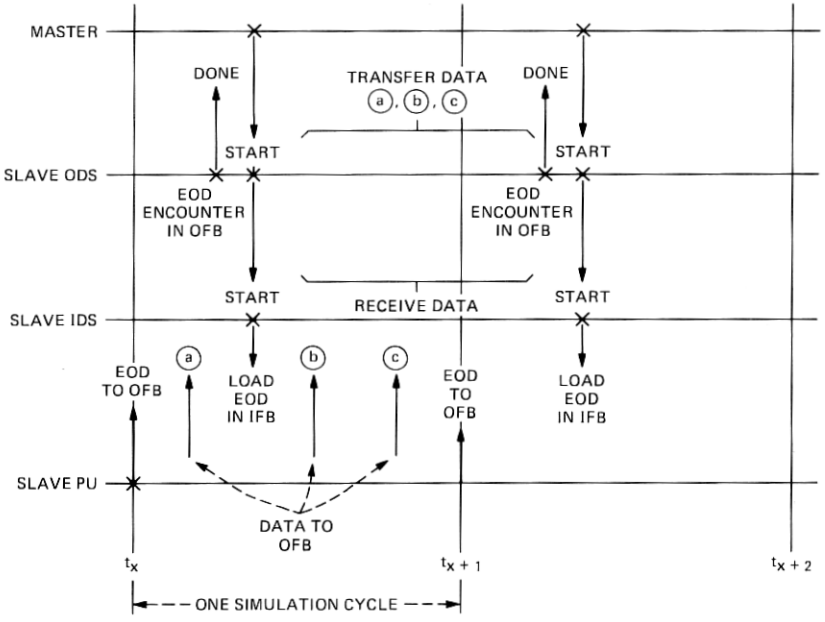


Fig. 10—Master and slave unit operations during a simulated time interval.

5.3.1 Time-shared parallel bus

The interface between the data sequencers and a time-shared parallel bus based communication structure is shown Fig. 11. When the ODS has some data to send, it sets the Request To Send (RTS) line high. The bus control grants it the use of the communication medium by sending a pulse on the Bus Grant line. The ODS sends out all the data present in its OFB. The data received by the IDS of the destination unit is put in its IFB. The ODS then sets the RTS line low. This releases the bus, which is then granted by the bus control to another requesting slave or the master. All units have equal priority. The ODS will set the RTS 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 consists of a scheduled event for the next simulation cycle. The data sent to the master consists of the address of the sending slave, element number (primary output or monitored point) and element value. 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 element value is stored in the master IFB by the master IDS.

5.3.2 Cross-point matrix

The interface between the data sequencers and a communication structure based on a cross-point matrix is shown in Fig. 12. When the ODS has some data to send, it puts the address of the destination unit on the address bus and makes a request to transfer data by sending a pulse on the RTS line. If the destination is not busy, the control for

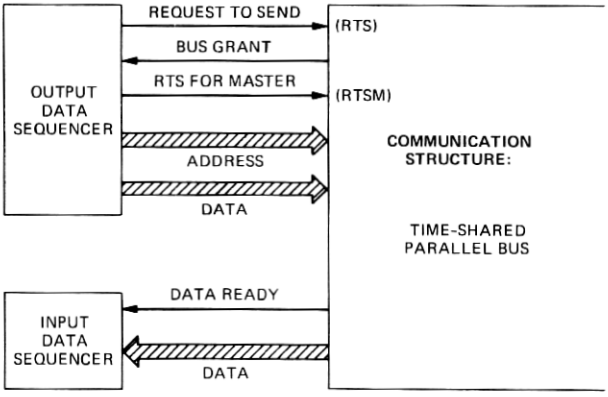


Fig. 11—Interface between the data sequencers and a time-shared parallel bus.

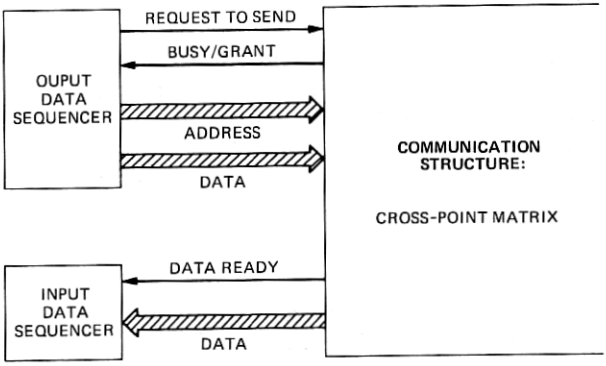


Fig. 12—Interface between the data sequencers and a cross-point matrix.

the matrix grants the transfer request. The ODS sends out data serially over the data line. The data received by the IDS of the destination unit is put in its IFB. The Data Ready line connected to the IDS is used to show the presence of data. It might happen that the destination is busy when an ODS makes an RTS to the cross-point matrix. In this case, the ODS gets a busy signal from the control of the cross-point matrix. In response to the busy signal, the ODS stores away the data that was to be transferred in its local memory and makes an RTS for the next set of data to be transferred from the IFB. A retry to send the blocked data is made later.

As indicated in Section 5.3.1, the master requires the address of the sending slave unit. A slave unit ODS will recognize a request to transfer data to the master and it will transmit its slave unit address together with element number and value.

The interface discussed above will apply for a nonblocking switching network also. However, this network will not be discussed here.

VI. ARCHITECTURE EVALUATION

In this section, various performance functions are derived for the multiprocessor architecture and compared with those for the traditional logic simulator implemented on a general-purpose computer. The requirements for a circuit-partitioning algorithm are considered first. Based on these requirements, expressions and values for processing and communication times are derived next. Comparisons in terms of evaluations per second are then made between the multiprocessor simulator and the traditional logic simulator implemented on a general-purpose computer.

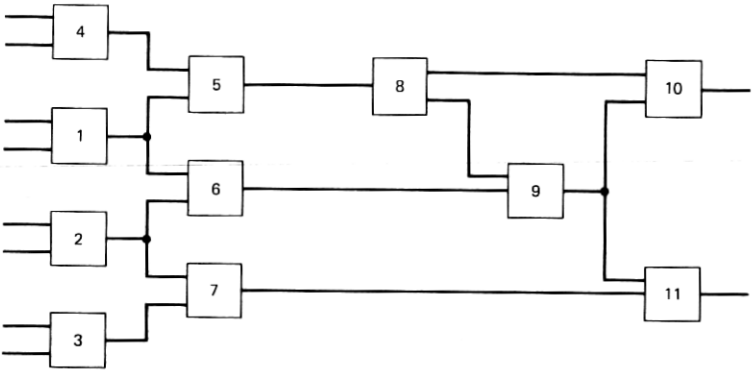


Fig. 13—A logic circuit.

6.1 Logic circuit partitioning

The logic circuit to be simulated is partitioned into a number of subcircuits. Each subcircuit can essentially be regarded as an independent circuit. During simulation, data is passed between different subcircuits as signal values propagate from the primary inputs to the circuit outputs. The subcircuits will be stored in the appropriate memories of the slave unit PUs. (The element numbers in the original circuit are translated to a slave address and an index number.) Partitioning is a key to the operation and performance of the multiprocessor digital logic simulator. Partitioning must maximize multiprocessing while limiting communication.

A partitioning algorithm is required to partition a logic circuit into subcircuits. The partitioning algorithm must produce subcircuits such that during logic circuit simulation, 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. Obviously, minimization of interprocessor communication and the proper choice of communication structure are necessary to avoid this bottleneck (see Sections VII and VIII). The fact that signals may propagate in parallel indicates that partitioning should be done along the depth of the circuit rather than the breadth of the circuit since this will tend to place concurrent activities in different blocks. One approach is to start with a primary input and trace a path towards a primary output forming an element string. An element string is a single fanout path. For example, elements (1, 6, 9, 11) and (5, 8, 10) in Fig. 13 constitute two element strings. Since two elements in a string will not be normally active simultaneously, the whole string can be put in one subcircuit. Each subcircuit corresponds to a block described in Section IV. Note

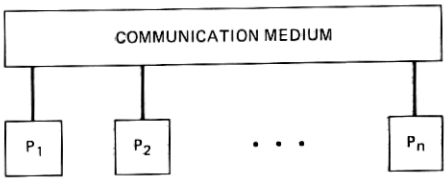


Fig. 14—Multiprocessor simulator architecture.

that if there is a fanout from any element to a zero delay element, then the two elements must be in the same block. For a multiprocessor system with n processors, a logic circuit must be partitioned into at most n blocks. A partitioning algorithm based on this approach is given in Appendix A.

6.2 Processing and communication

Consider the architecture of a multiprocessor simulator shown in Fig. 14. Processors p_1 through p_n are connected by a communication structure. The logic circuit to be simulated is partitioned into n blocks using the partitioning algorithm described in Appendix A. Each block is then assigned to a processor. During simulation, the communication structure will be used to transfer data between the processors as the interconnections between different blocks become active.

During a simulation cycle each of the processors will be evaluating active elements and scheduling events. Concurrently, data will be flowing across the communication structure as events in one processor activate elements in another processor. We define t_p as the average processing time per processor during a simulation cycle. Time t_p represents the processing of all active elements and scheduling of resulting events in one processor during a simulation cycle. Define t_c as the total communication time during one simulation cycle. This value represents the amount of time the communication structure is busy (i.e., the time taken to service all requests to transfer data from all processors) during a simulation cycle. Since the operations during t_p and t_c occur concurrently, the length of the simulation cycle and, hence, the number of evaluations per second will be determined by the greater of t_p and t_c.

Expressions and estimates for t_p and t_c for an optimum architecture will be derived in the next two sections. The value of t_c will be derived for two cases, namely, a communication structure based on a time-shared parallel bus and a cross-point matrix.

6.3 Processing time t_p

Let N be the average number of active elements per simulation cycle

and n be the number of processors in the multiprocessor simulator. Then N/n will be the average number of active elements per processor during a simulation cycle for the ideal case. There will be some extra active elements in a processor during a simulation cycle due to nonideal partitioning. If k is the average unbalance factor, then the number of active elements is kN/n. Let a be the time required to process one active element, then:

$$t_p = (N/n)\ ka,\ 1 < n < N;\ k = 1 \text{ for } n = 1.$$

This expression gives the average processing time per processor during one simulation cycle.

The value of the processing time a is estimated next. Traditional logic simulators can perform about 30,000 simple evaluations per second (e.g., IBM 370/168). This represents 33 μs per element evaluation. The simulation algorithms for the multiprocessor simulator and the traditional logic simulator are somewhat similar. The element evaluation time for the multiprocessor simulator can be written as $a = 33u_l$ μs. The factor u_l represents a slowdown factor due to the difference in speed between a microprocessor and a general-purpose computer. For an Intel* 8086 16-bit microprocessor the slowdown is about 5.5 over an IBM 370/168.[12] The element evaluation time for an Intel 8086 becomes 181.5 μs. The operation of the microprocessor can be speeded up by using a microprogrammable processor and microprogramming the simulation algorithm. For an Am29116 16-bit microprogrammable microprocessor, a speed-up factor of 5 to 10 can be obtained over the Intel 8086. Assuming an Am29116 and taking a speed-up value of 7, the average time required to process a simple element will be $a = 26$ μs. The value of a for functional elements will be 30 to 50 times larger.

Until now it has been assumed that all the processors have the same number of active elements. Assume that the unbalance due to nonideal partitioning introduces 10 percent more active elements in a processor ($k = 1.1$) and a is 40 times larger for functional evaluations than for simple evaluations [$a_{(se)} = 26$ μs for simple evaluations and $a_{(fe)} = 1040$ μs for functional evaluations]. The processing time for simple evaluations becomes $t_{p(se)} = 28.6(N/n)$ μs. The processing time for functional evaluations becomes $t_{p(fe)} = 1144(N/n)$ μs.

The processing time per active element (t_p/N) as a function of the number of processors (n) in the multiprocessor simulator is plotted for simple and functional evaluations in Fig. 15. If there is only one processor, then the processing time per active element for one simu-

* Trademark of Intel Corporation.

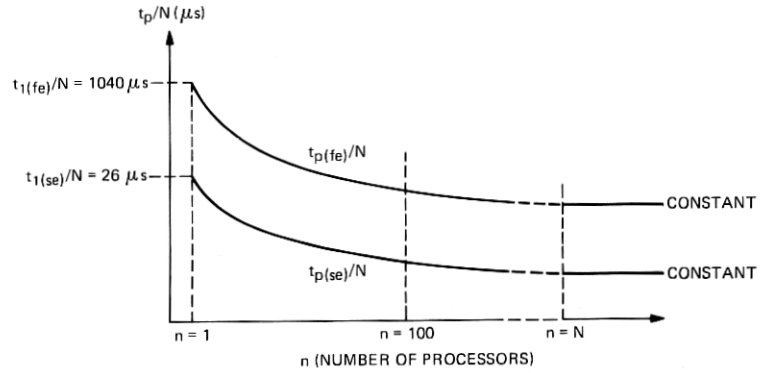


Fig. 15—Processing time as a function of number of processors for functional and simple evaluations.

lation cycle is $t_{1(se)}/N = 26$ μs for simple evaluations and $t_{1(fe)}/N = 1040$ μs for functional evaluations. Note that the unbalance factor k does not apply when there is only one processor.

The above analysis is done for $n \ll N$. For $n < N$ and $n \geq N$ there will be one active element per processor in the best case and the processing time per active element will remain constant at $t_p = (a/N)$ for all values of n greater than N (see Fig. 15). Also if $n \geq N$ and there is unbalance, then the value of k will be much larger.

6.4 *Communication time t_c*

The value of t_c will depend on the type of communication structure. Two types of communication structures will be considered here, a time-shared parallel bus and a cross-point matrix.

To estimate t_c for each case, first consider an element string yielded by the partitioning algorithm discussed previously (see Fig. 16). If f is the average fanout, one fanout line remains in the processor and $f - 1$ fanout lines go out to elements in other processors except the end of the string, where all fanout lines go to elements in other processors. Let c be the average number of elements in one string. Normally, one element per string is active during a simulation cycle. Define two adjacent element strings as two strings with common interconnections. Assume that all adjacent strings are in separate processors. This will give the situation that requires most communication and therefore the fastest communication structure. 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:

$$e = [f + (c - 1)(f - 1)]/c.$$

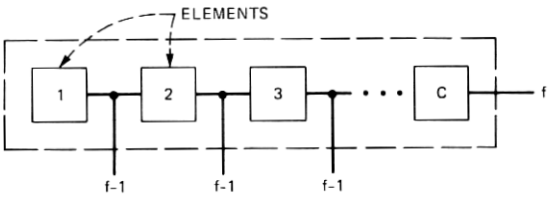


Fig. 16—An element string.

The typical value of f can be taken as 2 and for large circuits c is expected to be greater than 10. For f = 2 and c = 10, e will be equal to 1.1.

6.4.1 Time-shared parallel bus

Since the parallel bus is time-shared, the total communication time will be given by the time required to transmit one event multiplied by the number of communication events in a simulation cycle, i.e., Ne. The time required to transmit one event will be the sum of bus request and grant time (t_{brg}), address and data setup time (t_{ds}), data acknowledge time (t_{da}), and bus release time (t_{br}). An expression for the total communication time is:

$$t_{c(bus)} = (t_{brg} + t_{ds} + t_{da} + t_{br})(N)(e).$$

The address and data-setup time (t_{ds}) is composed of propagation delay without capacitance (t_{pd}) and capacitance delay (t_{cd}). These two parameters are functions of the bus length, which in turn will depend on the number of processors. If d is the distance between two processors then the average distance an event has to be sent is (nd)/2. Typical signal delay without capacitance is 1 ns/foot and if d is assumed to be 0.5 foot, then $t_{pd} = 0.25n$ ns. Capacitance will cause an extra delay of about 3 ns/foot giving $t_{cd} = 0.75n$ ns. The expression for $t_{c(bus)}$ becomes:

$$t_{c(bus)} = [n + (t_{brg} + t_{da} + t_{br})](N)(e) \text{ ns}.$$

Taking some typical values $t_{brg} = 100$ ns, $t_{da} = 50$ ns, $t_{br} = 50$ ns, and e = 1.1. The communication time per active element becomes:

$$t_{c(bus)}/N = (1.1n + 220) \text{ ns}.$$

This expression as a function of the number of processors in the multiprocessor simulator is plotted in Fig. 17a together with the expression for evaluation time per active element for simple evaluations and functional evaluations. Let t_1 be the length of the simulation cycle for a single-processor simulator and t_m be the length of the

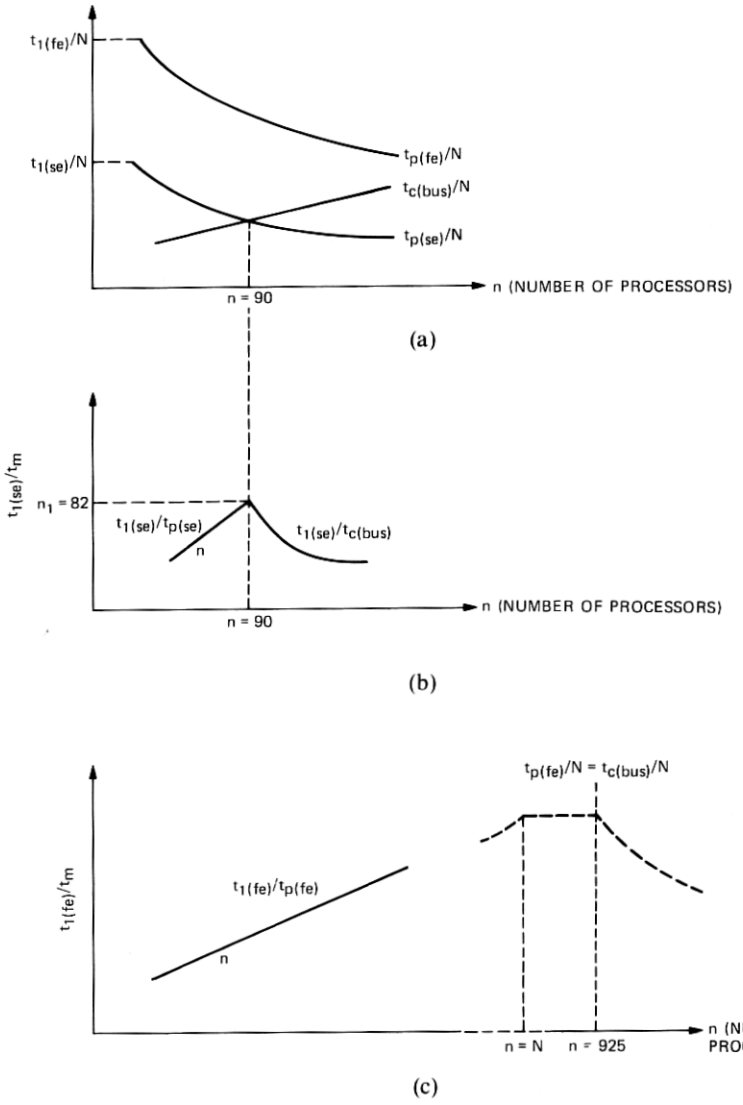


Fig. 17—Processing time and bus communication time considerations.

simulation cycle for a multiprocessor simulator. The performance of the multiprocessor simulator is defined as the ratio of single-processor time to multiprocessor time: t_1/t_m. The ratio of t_1 to t_m is plotted in Fig. 17b for simple evaluations and in Fig. 17c for functional evaluations.

From Fig. 17b it can be seen that for simple evaluations the ratio t_1

to t_m is maximum for $t_{p(se)} = t_{c(bus)}$. For this condition, the multiprocessor simulator gives best speed performance over the single-processor simulator. The value for n is about 90 for this condition. Adding more processors will not speed up the simulator because the communication time will be a bottleneck.

Note also from Fig. 17b that for $t_c < t_p$ the processing time is a bottleneck and processors can be added (as long as $n \ll N$) to speed up the simulation time. For $t_c > t_p$, the communication time is a bottleneck, and a faster communication structure has to be added to speed up the simulation time. For best case $t_c = t_p$ and for further speed improvements, both faster processors (or more processors if $n \ll N$) and a faster communication structure must be added.

It is worthwhile noticing that the communication activity will be slightly lower than the predicted activity for smaller values of n than for larger values, since the probability that two adjacent element strings will be in the same processor is higher for smaller values of n. An element string is as defined in the section on circuit partitioning (Section 6.1), and two adjacent element strings are two strings with common interconnections. This is not expected to have any significant impact on the above analysis.

Figure 17c shows that for functional evaluations the processing time will be a bottleneck and the time-shared parallel bus provides the required communication speed as long as $n \ll N$.

A problem that may be encountered in the parallel bus structure is data skewing. The greater the number of lines on the communication bus, the greater the effect of data skew. Line conditioning in the form of bus extenders might be required for proper operation. Also, for large n, a hierarchical bus structure will be required for suitable operation.

6.4.2 *Cross-point matrix*

In a cross-point based communication structure, several processors can simultaneously send data to other processors. The total communication time will be governed by the processor having the maximum data to be sent over the communication structure, i.e. (N/n)(k)(e) communication events. The time required to transmit one event will be the sum of the channel request and grant time (t_{crg}), delay incurred in transmission of message through matrix (t_{dm}), and channel release time (t_{cr}). Also when the processor asks to use the matrix, the destination processor might be busy. This requires selecting another event for transmission and trying to resend the blocked event at a later time. Let j be the number of events for which the channel is found busy and t_{rb} be the time wasted in processing a blocked request. An expression for the total communication time is:

$$t_{c(matrix)} = [t_{crg} + t_{dm} + t_{cr}](N/n)(k)(e) + (t_{rb})(j).$$

Taking some typical values t_{crg} = 50 ns, t_{dm} = 100 ns, t_{cr} = 50 ns, k = 1.1, t_{rb} = 50 ns, e = 1.1 and j = 0.1(N/n) (the channel is found busy for 10 percent of the transfer requests). The communication time per active element becomes:

$$t_{c(matrix)}/N = 247/n \text{ ns.}$$

Comparing this value with $t_{p(se)}/N = 28.6/n$ μs and $t_{p(fe)}/n = 1144/n$ μs, it can be seen that the matrix will never cause a bottleneck in the multiprocessor simulator.

It is interesting to note that since the matrix provides a high-speed communication structure, some inefficiency in the partitioning algorithm can be tolerated. For c = 1 (smallest possible average chain) and a worst-case average fanout of 5, e will be equal to 5 and $t_{c(matrix)}/N = 1.105/n$ μs. This value is still much less than the processing time for simple evaluations.

Figure 18 shows the various processing and communication times. Once again, the above analysis has been done for $n \ll N$. For $n < N$ and $n \geq N$ the matrix communication time will remain constant.

6.5 Comparisons

Estimated values for t_c and t_p will now be derived. For realistic situations $1 \ll n \ll N$. For a circuit with 100,000 elements, assuming 2-percent activity per time frame, N will be 2,000.

For simple evaluations and a parallel bus, the maximum value of n is 90. For n greater than 90, the performance goes down since the bus becomes a bottleneck. For n = 90 the length of the simulation cycle is $t_{p(se)}/N = 28.6/n = 0.32$ μs. The number of evaluations per second becomes 3,125,000. This represents an increase by a factor of 100 over a traditional logic simulator (30,000 evaluations per second). The growth of the multiprocessor with parallel bus is restricted in the sense that this is the optimum performance and increasing the processors will not yield any further improvements. For modularity and greater performance, a matrix based communication structure is required for simple evaluations. With a cross-point matrix, the performance can be increased by increasing n as long as n remains much less than the activity N. With n = 256 and a cross-point matrix, the number of simple evaluations per second becomes 9,000,000. This represents an increase by a factor of 300 over the traditional logic simulator. For n = 512 and a cross-point matrix, the number of simple evaluations per second becomes 18,000,000. This represents an increase by a factor of 600 over a traditional logic simulator.

The speedup for functional evaluations with a time-shared parallel bus is of the same order. The maximum value of n in this case is 925. For n = 90, the speedup factor is 100 and for n = 256 the speedup

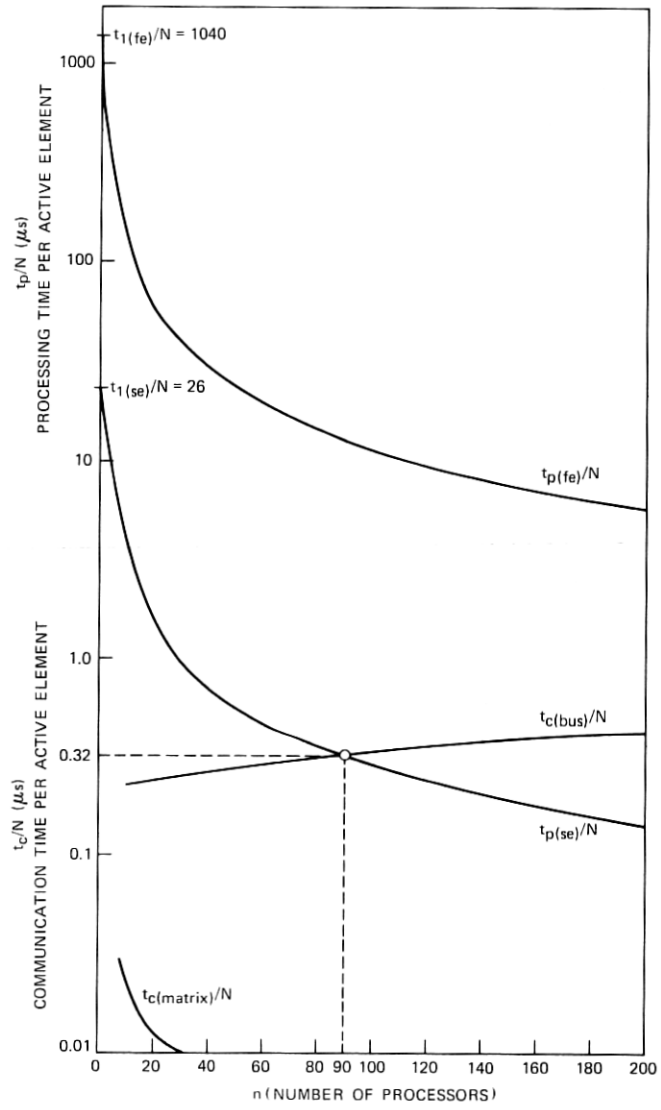


Fig. 18—Comparison of various communication times and processing times.

factor is 300. However, smaller values of n will be used in practice since the activity will be much lower for functional evaluations.

If slightly lower performance can be tolerated, then a nonmicroprogrammable microprocessor can be used. For the Intel 8086, the element-evaluation time becomes a = 181.5 μs. For a parallel bus, the value of n for optimum performance is 320. The number of simple evaluations per second becomes 1,600,000, an increase in performance

of 53 over a traditional logic simulator. Once again, however, this is the best performance that can be obtained using the parallel bus. Using a cross-point matrix with n = 512, the speed up over the traditional logic simulator is 90.

VII. ARCHITECTURE CHOICE

It is seen from the previous section that a cross-point matrix is preferable for simulating a circuit containing simple elements. A time-shared parallel bus can be used for simple evaluations, but a speed penalty will be incurred and modularity will be lost. A time-shared parallel bus is sufficient for functional evaluations. A combination of the cross-point matrix and parallel bus can be used to simulate circuits containing both simple and functional elements.

For both simple evaluations and functional evaluations, a communication structure consisting of both a cross-point matrix and a time-shared parallel bus will prove cost effective. For transferring data between the cross-point matrix and the parallel bus a Bus Interface Unit (BIU) is required. The configuration of the BIU is shown in Fig. 19.

Data sequencer 1 transfers data from functional evaluators connected to the parallel bus to the simple evaluators and the master connected to the cross-point matrix. The data sequencer receives signal information and data from the parallel bus and transforms it for

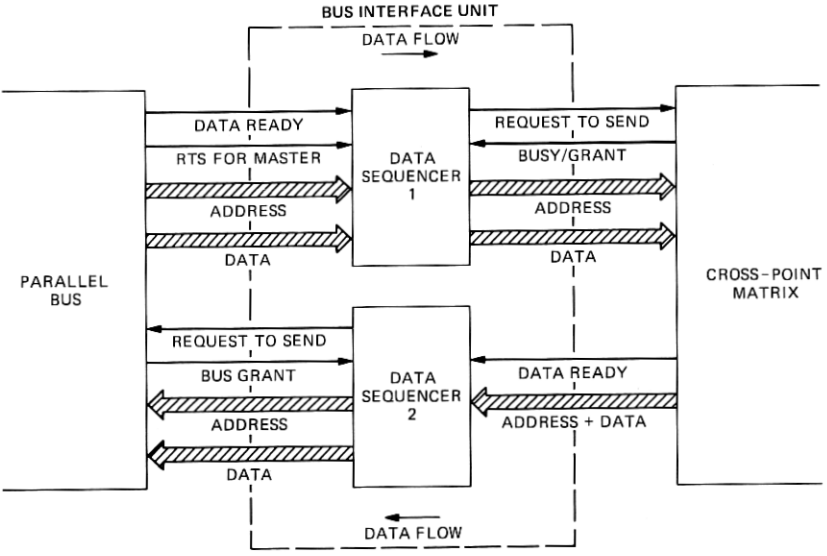


Fig. 19—Interface between parallel bus and cross-point matrix.

suitable transmission over the cross-point matrix. The parallel data received from the bus is transferred serially over the matrix.

Data sequencer 2 sends data from the cross-point matrix to the parallel bus. The serial data received from the cross-point matrix is transferred to parallel data for suitable transmission over the parallel bus.

VIII. PERFORMANCE ANALYSIS

The effect of varying various parameters on the performance of the multiprocessor is considered in this section. The analysis so far has been done using average values. The performance of the multiprocessor will be affected to some extent by variations in various parameters. The processing times $t_{p(se)}$ and $t_{p(fe)}$ are functions of N, n, k and a. The communication time for bus $t_{c(bus)}$ is a function of N, n, f, c. The communication time for matrix $t_{c(matrix)}$ is a function of N, n, k, f, c. The effect of variations in some of these parameters on performance of the logic simulator will be investigated.

8.1 Effect of changes in circuit activity N

8.1.1 Bus architecture

8.1.1.1 Simple evaluations. As seen from Fig. 20a, if activity N increases then $t_{p(se)}$ and $t_{c(bus)}$ increase proportionately. The operating point shifts from (1) to (2) for increasing values of N. This means that the length of the simulation cycle for the proposed simulator will increase by the same percentage that N increases. In the case of a single processor simulator, the simulation cycle will also increase by the same percentage. The performance index of the multiprocessor simulator (in terms of single processor time to multiprocessor time ratio) is not affected by the circuit activity, N (Fig. 20b).

8.1.1.2 Functional evaluations. The processing time of the multiprocessor simulator $[t_{p(fe)}]$ is proportional to the processing time for simple evaluations $[t_{p(se)}]$ by a slow-down factor of between 30 and 50. The analysis for simple evaluations done above, therefore, also applies for functional evaluations (i.e., the performance index of the multiprocessor simulator is not affected by circuit activity).

8.1.2 Matrix architecture (simple evaluations)

Since the processing time $[t_{p(se)}]$ and matrix communication time $[t_{c(matrix)}]$ are both proportional to N, they will increase by equal proportions. The length of the simulation cycle is governed by $t_{p(se)}$ and the performance index (single processor to multiprocessor time) is given by $t_1/t_m = (Na)/[(N/n)ka] = n/k$. Once again this value is independent of N and the performance index of a multiprocessor simulator with matrix architecture is not affected by circuit activity.

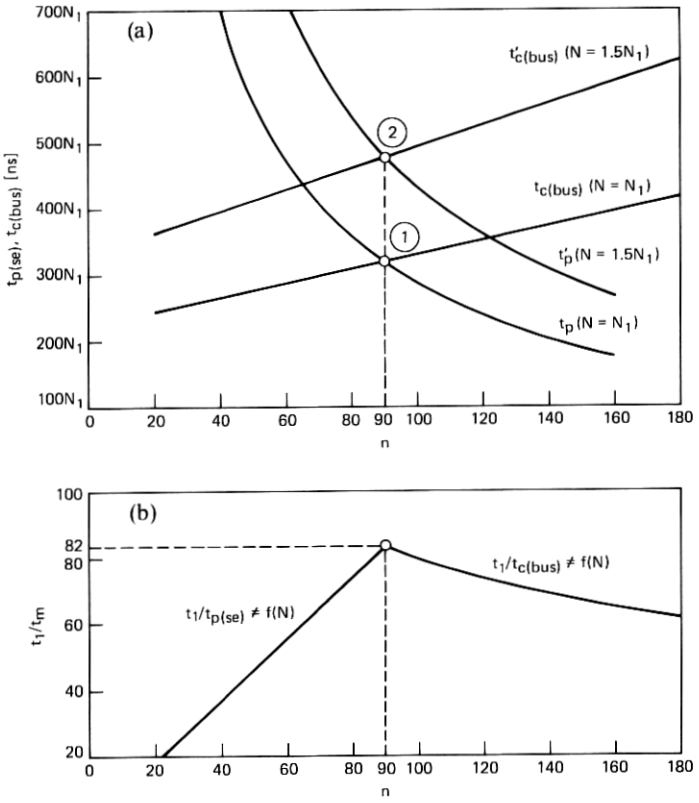


Fig. 20—Effect of changes in circuit activity. (a) Processing time (simple evaluations) and bus communication time for different values of N. (b) Performance variation for (a).

8.2 *Effect of partitioning: variations in k*

8.2.1 *Bus architecture*

8.2.1.1 Simple evaluations. The factor k represents the increase in activity in a processor, over the average N/n, due to non-ideal partitioning. An increase or decrease in k changes $t_{p(se)}$ in proportion but does not affect $t_{c(bus)}$. As seen from Fig. 21a for k = 1.1, the operating point is at (1) with n = 90. If k increases then the operating point moves to (2). The length of the simulation cycle increases in proportion to the increase in k. Since the length of the simulation cycle for the single-processor simulator is not affected by k, the multiprocessor performance index goes down. For n = 90, the performance index of the multiprocessor goes down from 82 to 64 as k increases from 1.1 to 1.4.

If the multiprocessor simulator is designed for a higher value of k, then the number of processors required for maximizing the perform-

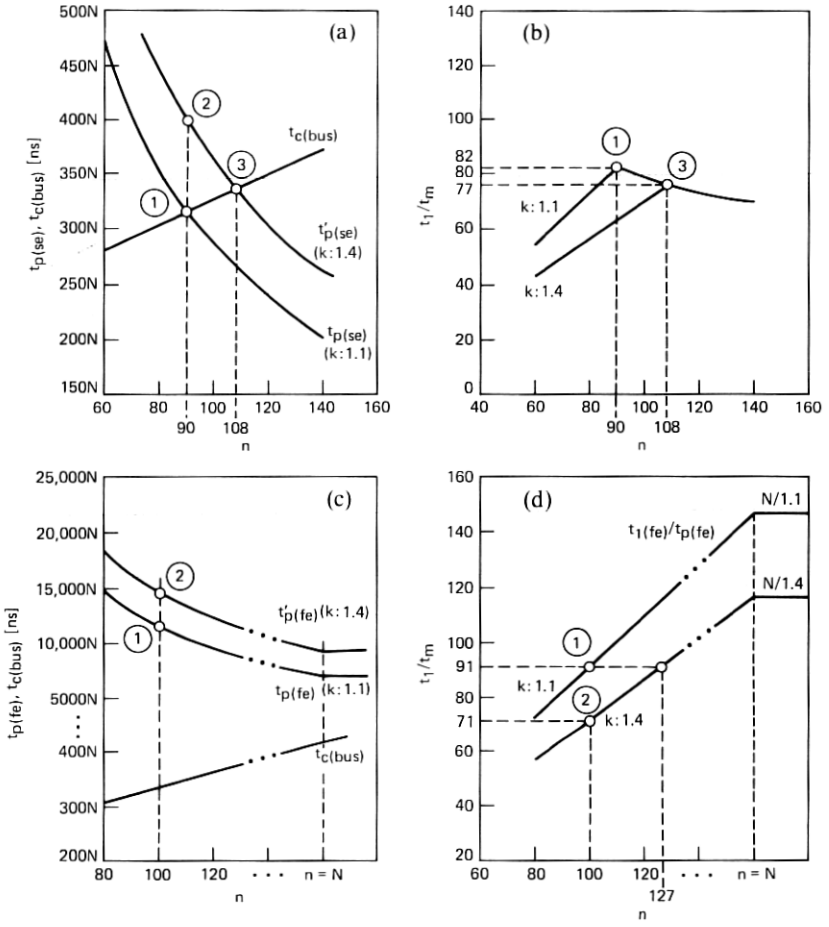


Fig. 21—Effect of partitioning: variations in k. (a) Processing time (simple evaluations and bus-communication time for various values of k. (b) Performance variation as affected by changes in k for (a). (c) Processing time (functional evaluations) and bus-communication time for various values of k. (d) Performance variation as affected by changes in k for (c).

ance index is greater than 90 and the maximum performance index decreases as k increases. This is shown by points (1) and (3) in Fig. 21b. For a multiprocessor simulator designed for k = 1.1, the maximum performance index is 82 for n = 90. If the multiprocessor is designed for k = 1.4, the maximum performance index is 77.

8.2.1.2 Functional evaluations. For functional evaluations, the curves for $t_{p(fe)}$ and $t_{c(bus)}$ meet for a large value of n (= 925). The length of the simulation cycle will be governed by the processing time. Figure 21c shows that for n = 100, the operating point will move from (1) towards

(2) as k increases from 1.1 to 1.4. Thus, the length of the simulation cycle will increase. The increase will not affect the single-processor simulator since its simulation cycle is not affected by k. Thus, the overall performance index will decrease as the activity increases. The performance index decreases from 91 to 71 as k increases from 1.1 to 1.4 for n = 100 (see Fig. 21d).

If the multiprocessor is designed for a higher value of k, then the number of processors required to maintain the performance index constant goes up. For example, for k = 1.1 and n = 100 the performance index is 91. If the simulator is to be designed for k = 1.4, then 127 processors are required to maintain the performance index at 91.

8.2.2 Matrix architecture (simple evaluations)

Once again, the processing time [$t_{p(se)}$] and matrix communication time [$t_{c(matrix)}$] are proportional to k. The processing time will determine the length of the simulation cycle regardless of the value of k. This case is similar to that in the previous section for functional evaluations and a parallel bus (Section 8.2.1). The overall performance index will go down as the activity increases. Also if the multiprocessor is designed for a higher value of k then the number of processors required to maintain the performance index constant goes up.

8.3 Effect of partitioning: variations in c

8.3.1 Bus architecture

8.3.1.1 Simple evaluations. Variations in c, the average number of elements per element string, will affect the bus communication time $t_{c(bus)}$. A decrease in c will increase the bus communication time. Figure 22a shows that the operating point moves from (1) to (2) as the value of c decreases from 10 to 1. For a constant number of processors and a decreasing c, the communication time may become a bottleneck. The length of the simulation cycle will increase and the performance index will go down.

If the simulator is designed for smaller values of c, the number of processors required will be smaller but the maximum performance goes down [operating point (3) in Fig. 22b].

8.3.1.2 Functional evaluations. For functional evaluations, the processing time is a bottleneck. The worst-case value of c is one (i.e., chains of length one). Even for this case the curves for $t_{p(fe)}$ and $t_{c(bus)}$ meet for a large value of n (= 663). Thus, the simulator for functional evaluations is not affected by c (typical value of n is 100).

8.3.2 Matrix architecture (simple evaluations)

As noted at end of Section 6.4.2, the processing time $t_{p(se)}$ will still be

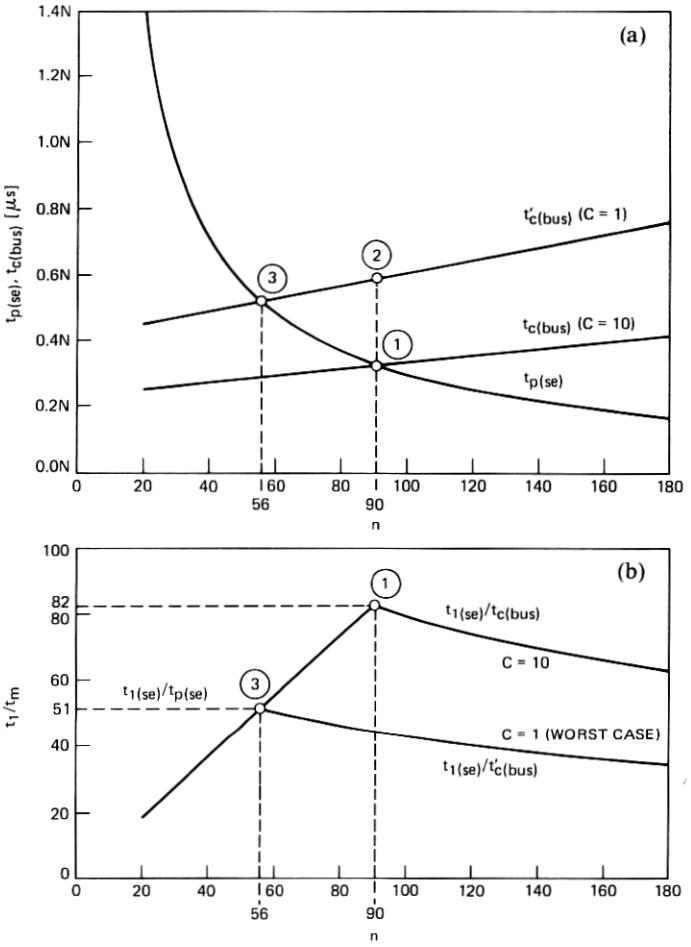


Fig. 22—Effect of partitioning: variations in *c*. (a) Processing time (simple evaluations) and bus-communication time for different values of c. (b) Performance variation as affected by changes in c for (a).

a bottleneck for worst-case value of c = 1. Thus, the performance of the simulator with a matrix is not affected by variations in c.

8.4 Effect of variations in fanout, f

8.4.1 Bus architecture

8.4.1.1 Simple evaluations. An increase in f will increase the bus communication time $t_{c(bus)}$. The processing time will not be affected by changes in f. Figure 23a shows that the operating point moves from (1) to (2) as fanout increases. Running circuits with larger fanout on a

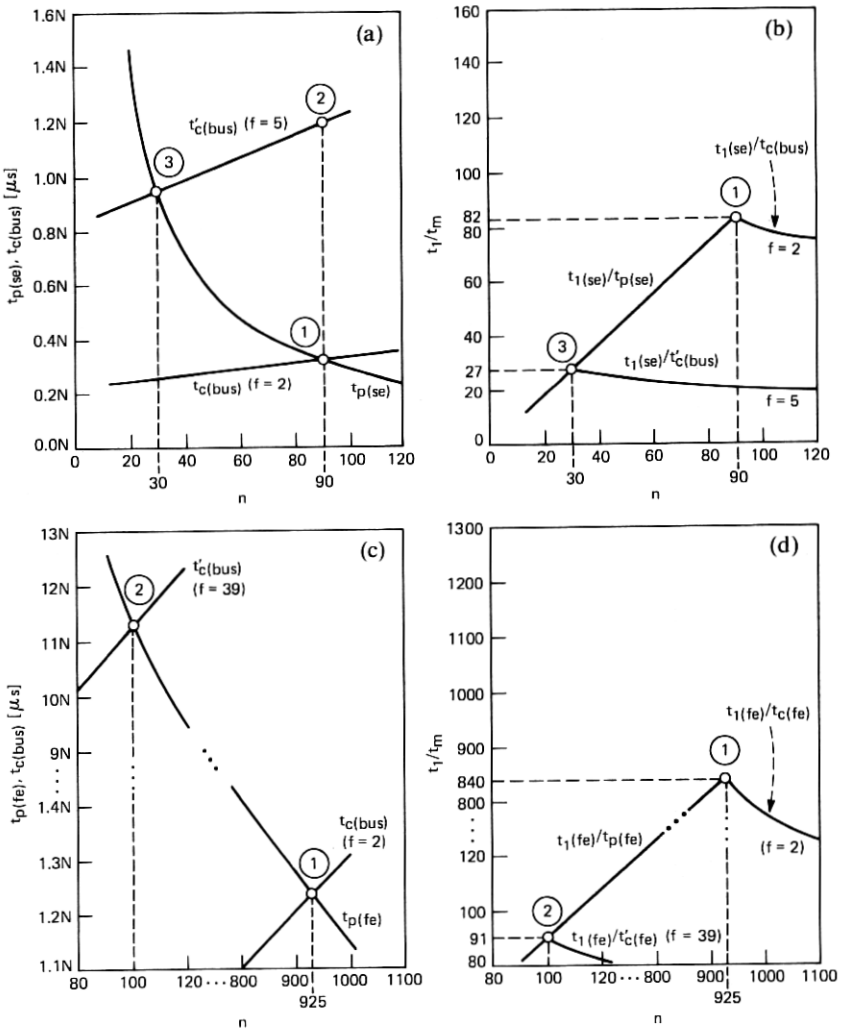


Fig. 23—Effect of variations in fanout. (a) Processing time (simple evaluations) and bus-communication time for different values of f. (b) Performance variation as affected by changes in f for (a). (c) Processing time (functional evaluations) and bus-communication time for different values of f. (d) Performance variation as affected by changes in f for (c).

simulator designed to operate with an average fanout of 2 will cause the communication time to become a bottleneck. Since the processing time does not change, the performance index of the simulator will decrease. Similarly, if a simulator is designed for larger fanout its maximum performance index would be less than that of a simulator designed for a smaller number of fanout. (See Fig. 23b.)

8.4.1.2 Functional evaluations. The simulator for functional evaluations is sensitive to changes in f. As seen from Fig. 23c, for n = 100 and f > 39 the communication time becomes a bottleneck and the performance index of the simulator goes down. The maximum performance index of the simulator is also lower for higher fanout. For large fanout, the communication time becomes a bottleneck for functional evaluations and a matrix may have to be used.

8.4.2 Matrix architecture

8.4.2.1 Simple evaluations. For an average fanout of 2, the processing time is a bottleneck. As the fanout increases, the curves for $t_{p(se)}$ and $t_{c(matrix)}$ approach each other. The communication time becomes a bottleneck for f > 128. This is an unrealistically large value and will not occur in practice.

8.4.2.2 Functional evaluations. The effect on functional evaluations is the same as discussed above, but f > 5,100.

IX. SUMMARY

The architecture of a multiprocessor simulator has been presented. The speed/performance ratio of the simulator is expected to be greater than two orders of magnitude compared to traditional simulation methods implemented on general-purpose computers. The power of the simulator can be increased over a certain range by increasing the number of slaves. Also the cost of the CPU time should be much lower than that obtained from general-purpose computers.

The architecture presented in this paper is expected to be faster than those of Barto and Szygenda, and Abramovici et al. The Yorktown Simulation Engine (YSE) built by IBM is reported to be faster than the architecture presented here, but the cost of the machine would be substantially higher since it uses special-purpose hardware. The architecture presented here and the YSE both try to take advantage of logic circuit concurrency to improve simulation performance. Unlike the YSE, our architecture implements event-driven simulation and is applicable to simulation with arbitrary delays at both gate and functional levels. Further work to be done includes detailed comparison of the various architectures in terms of performance and cost.

The application of the multiprocessor simulator to fault simulation is being investigated at the present time.

X. ACKNOWLEDGMENTS

The authors wish to thank J. Grason and D. B. Armstrong for reviewing this paper and providing helpful suggestions.

APPENDIX A

Partitioning Algorithm

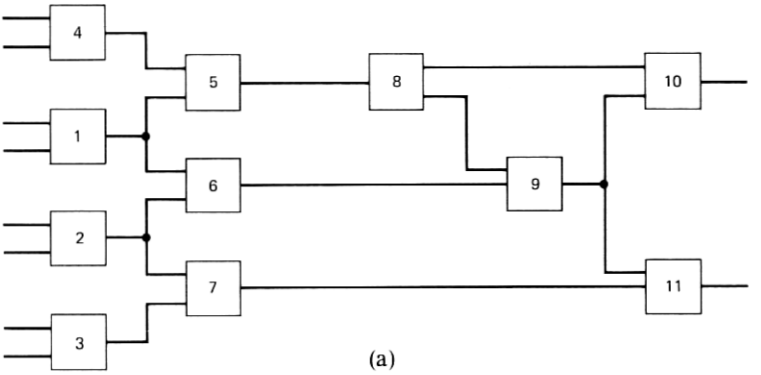
A way to partition the logic circuit is to first generate and assign element strings to blocks, where an element string is as defined in Section 6.1. The next step is to balance the load (number of elements) in the blocks such that all blocks have approximately the same number of elements. Note that the elements have to be ordered according to levels prior to partitioning of the logic circuit. The partitioning algorithm is detailed below:

A.1 Generate and assign strings

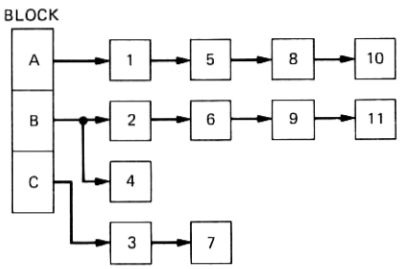
1. $i = 0$
   [Note: a_i is the block to which assignments are currently being made.]
2. Select an unmarked element whose fanins have all been assigned previously to blocks other than a_i. Call the selected element e_s. If there is no such element and there are still some unmarked elements, go to step 6. If all elements are marked, go to algorithm A.2 (Load Balancing).
   [Note: Primary inputs can be treated as elements whose fanin has been previously assigned to blocks other than a_i.]
3. Assign the selected element e_s to block a_i and mark e_s.
4. If any fanout e_k of e_s is in a_i, go to 6.
   [Note: If a fanout element has been previously assigned to the current block a_i, then assigning another fanout element to the current block will require sequential processing of two elements.]
5. If there is an unmarked fanout e_k of e_s such that no fanin (e_k) (except e_s) is in a_i,
   then: (a) $s = k$
   (b) Go to step 3.
   [Note: If all of the fanout elements have been assigned to some other blocks, then the string cannot be extended. Note that primary outputs can be treated as being assigned to a null block.]
6. (i) $i = (i + 1)$(modulo n)
   (ii) Go to step 2.

A.2 Balance load

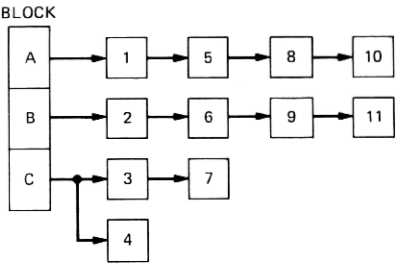
1. Total the number of elements in each block.
2. a_{max} = block with most elements
   a_{min} = block with fewest elements



(a)



(b)



(c)

Fig. 24—Partitioning example. (a) Logic circuit to be partitioned. (b) String assignments before load balancing. (c) String assignments after load balancing.

e_{max} = number of elements in a_{max}

e_{min} = number of elements in a_{min}

3. If $(e_{max} - e_{min}) <$ [(total number of elements)/n]0.1, stop.
   [Note: If the maximum unbalance is less than 10%, stop.]
4. Select string s_i in a_{max} such that length $(s_i) < e_{max} - e_{min}$.
5. Move string s_i to block a_{min}.
6. If $(e_{max} - e_{min}) <$ [(total number of elements)/n]0.1, go to step 4.
   Else: Go to step 2.

As an example, consider the circuit in Fig. 24 to be partitioned into three blocks. The blocks and strings are then assigned as shown while looping through steps 2 through 6 in Section A.1. Note that element 4 is not assigned to block A since the fanout element 5 is already in block A (step 4 in Section A.1.). At end of load balancing element 4 will be assigned to block C.

Note that zero delay elements are not considered in the algorithm. Any zero delay element and all of its fanouts must be in the same block of the partition.

APPENDIX B

Glossary

BIU—Bus Interface Unit.
CPU—Central Processing Unit (part of the master unit).
EOD—End of Data (flag).
IDS—Input Data Sequencer.
IFB—Input FIFO Buffer.
L_t—Current list of timing wheel.
N—Average number of active elements per simulation cycle.
ODS—Output Data Sequencer.
OFB—Output FIFO Buffer.
RTS—Request to Send line.
RTSM—Request to Send to Master.
PU—Processing Unit. (Part of a slave unit)
a—Time required to process one active element.
a_i—A block of a partitioned circuit.
b_{ij}—Signal connections between blocks a_i and a_j.
c—Average number of elements in an element string.
c_i—Subcircuit located in processor p_i.
d_{ij}—Data path between processors p_i and p_j.
f—Average fanout of an element.
k—Imbalance factor owing to non-ideal partitioning.
n—Number of processors in the multiprocessor simulator.
p_i—Processor in multiprocessor simulator.
t_1—Length of simulation cycle for a single processor simulator.
$t_{1(fe)}$—Processing time during one simulation cycle with single processor (functional evaluations).
$t_{1(se)}$—Processing time during one simulation cycle with single processor (simple evaluations).
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 parallel bus structure.
$t_{c(matrix)}$—Total communication time during one simulation cycle for a matrix structure.
t_{cd}—Capacitance delay portion of address and data setup time, t_{ds}.
t_{cr}—Channel release time for a matrix structure.
t_{crg}—Channel request and grant time for a matrix structure.
t_{da}—Data acknowledge time for a parallel bus structure.
t_{dm}—Delay incurred in transmission of message through matrix.
t_{ds}—Address and data setup time for a parallel bus structure.
t_m—Length of simulation cycle for a multiprocessor simulator.

t_p—Average processing time per processor during a simulation cycle, where average processing time consists of the time required to process all active elements and schedule resulting events.

$t_{p(fe)}$—Average processing time per processor during one simulation cycle for functional evaluations.

$t_{p(se)}$—Average processing time per processor during one simulation cycle for simple evaluations.

t_{pd}—Propagation delay portion (without capacitance) of t_{ds}.

REFERENCES

1. M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Rockville, Maryland: Computer Science Press, 1976, p. 148.
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. E. G. Ulrich and T. G. Baker, "Concurrent Simulation of Nearly Identical Networks," Computer, *7-4* (April 1974), pp. 39–44.
4. D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," IEEE Trans. Comp., *C-21*, No. 5 (May 1972), pp. 464–71.
5. R. Barto and S. A. Szygenda, "A Computer Architecture for Digital Logic Simulation," Elec. Eng., *55*, No. 642 (September 1980), pp. 35–66.
6. M. Abramovici, Y. H. Levendel, and P. R. Menon, "A Logic Simulation Machine," Proc. 19th Design Automation Conf., Las Vegas, Nevada, June 14–16, 1982, pp. 65–73.
7. G. Pfister, "The Yorktown Simulation Engine: Introduction," Proc. 19th Design Automation Conf., Las Vegas, Nevada, June 14–16, 1982, pp. 51–54.
8. M. M. Denneau, "The Yorktown Simulation Engine," Proc. 19th Design Automation Conf., Las Vegas, Nevada, June 14–16, 1982, pp. 55–59.
9. E. Kronstadt and G. Pfister, "Software Support for the Yorktown Simulation Engine," Proc. 19th Design Automation Conf., Las Vegas, Nevada, June 14–16, 1982, pp. 60–64.
10. N. D. Phillips and J. G. Tellier, "Efficient Event Manipulation, A Key to Large Scale Simulation," Proc. IEEE 1978 Semiconductor Test Conf., Cherry Hill, New Jersey, October 31–November 2, 1978, pp. 266–73.
11. J. G. Vaucher and P. Duval, "A Comparison of Simulation Event List Algorithms," Comm. ACM, *18-4* (April 1975), pp. 223–30.
12. J. Feder, unpublished work.