AN EXAMPLE OF SIMULATION AS A COMPUTER SYSTEM DESIGN TOOL

by

Gary J. Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

November, 1976

This work was supported under National Science Foundation Grant Number MCS74-08328 A01.
The author's current affiliation is with Xerox Palo Alto Research Center, Palo Alto, California.

SIMULATION AS A
COMPUTER SYSTEM DESIGN TOOL

by

Gary J. Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-099-76 November, 1976

This work was supported under National Science Foundation Grant Number MCS74-08328 A01.

ABSTRACT

The pros and cons of using simulation during the design phase of machine development are discussed. An approach using two levels of simulation models to investigate proposed machine performance is illustrated by a case study of a parallel processor machine. It is argued that the approach alleviates some of the liabilities traditionally encountered in machine simulation, and that it is a cost-effective method of design analysis.

INTRODUCTION

Simulation techniques have been widely used in computer performance studies, including system selection, design of new systems, and software tuning of existing systems, [1-3]. Our experiences show that simulation can be particularly effective when used as a system design tool; in this paper, the use of simulation during the design of a parallel processor is described.

In the next section, some design evaluation tools are discussed along with assets and liabilities of each. Next, a brief overview of a parallel processor is provided; and finally, the use of two levels of simulation models in evaluating the design parameters of the parallel processor is described.

DESIGN EVALUATION TOOLS

Merikallio and Holland have listed a set of possible tools to be used to evaluate performance measures of alternative designs, [4]:

- Mathematical analysis using average values.
- Queuing theory models.
- Discrete simulation models.
- Experimentation with prototypes.

Each of these approaches has its strong and weak points. Models that predict performance based only on average values of the independent variables (i.e. input parameters) are relatively easy to analyze; however, they assume that the variance of such variables is small. If the variance is large, then an average value model will ignore sporadic service and request patterns, causing the predicted performance to be optimistic in terms of the resources required to handle peak loads.

Queuing models are more precise, since they allow service and request patterns to be described by probability distributions for the random values of the independent variables, i.e. they allow the model to incorporate more detail than average value models. Unfortunately, as queuing models increase in this level of detail, they become correspondingly more difficult to solve.

Discrete simulation models allow more detail to be incorporated into the model, since a program will be written to handle the added components of the model. Input parameters of a simulation model may

be derived from probability distributions (as in queuing models), or they may be provided by trace data. The difficulties encountered with simulation models, (discussed in more detail below) are that they may be hard to validate and expensive to exercise.

Prototype development is the ultimate design tool in the sense that the amount of detail is at the lowest level, i.e. it is a copy of the machine design. Unfortunately, prototypes are expensive to build and test, and they are also difficult to use to explore a number of alternative designs.

The design effort for a computer system should use all four tools. Average value models can be used to make gross performance predictions which will rule out certain design alternatives. Next, the remaining alternatives should be analyzed with queuing models, adding another level of detail. As the queuing models become more and more detailed, they will eventually become too complex to analyze. Queuing model studies might also eliminate some of the possible designs from further consideration. Simulation models should then be used to handle the increased detail in testing the design alternatives that have been shown to be acceptable by previous modeling techniques. The final step is to test the design alternatives, (that are still plausible), by building one or more prototypes of the designs.

Simulation: Pros and Cons

Any simulation model is a tradeoff between realism and the cost to simulate that realism. The amount of realism incorporated into the model is directly reflected by the level of detail incorporated into that model. At one extreme is a simulation of a queuing model, i.e. the simulation incorporates a component to sample a request distribution, one to model a queue, and a third component to model the service of the request (based on a service time distribution). At the opposite extreme is a simulation model containing components that are functionally equivalent to each component in a real system; the distinction between the model and the target system lies in the implementation of each component. The approach advocated here is that several simulation models should exist during the machine design phase, where some models tend toward the prototype and others tend toward queuing models.

In order to rely on the performance prediction of any model, one must have confidence in that model. Confidence can be gained in one of two ways: The simulation model can be verifed by comparing predicted performance of the model with an independent prediction, or the simulation program that implements the model can be validated to be a correct model of the system, and then the program can be verified to be "correct" with respect to the model definition. If the simulation model represents a machine that is still in the design phase, then true validation is impossible, since no real hardware exists that can be used to compare predicted performance and real performance. One popular technique used in such situations is to verify the simulation model and program against an independent queuing model, (e.g. see [5]). However, others have used simulation models to verify their queuing model, (e.g. see [6]). Either approach is not likely to be a rigorous validation, since one model is verified by another model.

The alternative course of action is also difficult. One would like to ensure that the simulation program is equivalent to the design specifications of the system it models. If one attempts to prove that the model is equivalent to the specifications, then a formal argument must be put forth which relies on a formal definition of "equivalent". Current state of the art in proving programs does not seem to indicate that the approach is viable for large programs. Instead of a formal proof of equivalence, others have used the idea of machine-aided design directly from specifications to obtain certifiable programs, (e.g. see [7]). This appears to be a promising approach to validation.

Another difficulty with simulation (and queuing) models is that of adequately determining and characterizing the load for the models. Probability distributions are frequently used to represent individual parameters, and in some cases, joint probability distributions are used to interrelate parameters, (e.g., see [8]). When a distribution function is used to describe a known workload, then a model is being used to mask out detail of the real world situation. For example, an exponential distribution is frequently used to represent the interarrival rate of requests to a system, (this assumes that the arrival pattern is Poisson). Although convenient to use, and easy to argue for in the absence of real data, the random arrival assumption does

not hold if the requests are interrelated. Distribution functions should be based on observed patterns if at all possible. Determining the workload for a nonexistent system has the added problem that no current workload exists. In order to generate a workload, one must either hypothesize a workload or else explore the application areas and environments for which the (nonexistent) system is intended.

Although there are several problems to be overcome when using simulation, there are also advantages. For models that are too detailed to handle with queuing theory, simulation is far more cost-effective than testing unproven prototype designs. It is relatively easy to test a design, alter the design, and then retest the system when using simulation. One also has the freedom of judiously eliminating uninteresting detail for particular models by simplifying the model component that represents that detail. For example, the simulation model of a computer may be written to investigate memory conflicts; the details of operation of the processing unit(s) and I/O processor(s) are of interest only in terms of the pattern in which they make memory references. A simulation model can use distribution functions to describe the effect of these units without actually including their explicit actions into the model.

In the remainder of this paper, a case study of a parallel processor design effort using simulation is discussed. These techniques have been used to reduce the amount of difficulty in handling varying levels of detail, in exploring possible job loads and their representations for the machine, and in correctly handling the occurrence of parallelism.

MULTI ASSOCIATIVE PROCESSOR OVERVIEW

The Multi Associative Processor, hereafter MAP, is a parallel processor employing 8 control units and 1024 processing elements. (See Figure 1). Only a brief description of MAP will be given in this section, and the interested reader is invited to see the references, [9,10]. Each control unit, in combination with a subset of the processing elements, operates on a single instruction stream through the control unit and multiple data streams, one through each processing element (i.e. SIMD operation). The instruction execution cycle for one control

unit and one or more processing elements can be described as follows: The control unit fetches an instruction from the main memory and then decodes the instruction to obtain a series of low level operations that each processing element must execute in order to carry out the instruction. The control unit "broadcasts" this set of operations over the instruction bus shown in Figure 1, one at a time, to the set of processing elements that are currently assigned to that control unit. Each processing element executes the operations on data that resides in a memory that is local to the processing element. Since it is unlikely that all instructions will apply to each processing element that is assigned to the control unit at all times, a mechanism to temporarily activate or deactivate a given processing element is required; we refer to this mechanism as an "associative unit".

The processing element memories are loaded sequentially by the control unit from the main memory. This loading amounts to data loading rather than program loading, since the processing element memory is not used to store instructions. In order to provide this capability to transfer data from the main memory to the processing element memories, a data bus is required, as indicated in Figure 1. The philosophy of this machine will require high utilization of the instruction bus system and relatively low utilization of the data bus system provided that high I/O programs are not executed on the machine. For this reason, and since the data bus will generally be wider than the instruction bus, the architecture incorporates dedicated instruction buses and shared data buses. Although the general architecture of the machine is specified above, and in Figure 1, the parameters of the design are subject to the actual condition under which the machine might be used. For example, the main memory is shared among eight control units and must be designed to support the simultaneous access of all control units and the I/O system.

Design studies of the MAP system have primarily concentrated on the following problem areas:

1. Main Memory conflict among the control units [11].
2. The number of shared data buses required in a MAP system [11].
3. Allocation algorithms to reduce data bus conflicts, [12].
4. Operating System Strategies for MAP [13].
5. Measuring the performance of individual MAP programs [14].

In the first two studies, simulation programs were used to investigate hardware designs; the latter three studies investigated software performance using simulation. In all cases, a realistic job load for the machine was generated using other simulations.

TWO-LEVEL SIMULATION MODELS

An immediate difficulty encountered in this project was that no machine with characteristics similar to MAP was available. Hence, the uses to be made of such a machine were based on pure conjecture; no understanding of the job load that the system might experience was known. Clearly, the performance of the machine on a purely hypothetical job load is of little interest to anyone; some appropriate application areas had to be investigated, and individual program requirements had to be obtained. This led to the development of the first in a set of hierarchical simulation programs, namely an interpreter for the machine, (simulating one control unit and an arbitrary number of processing elements). The interpreter, described in more detail below, allowed real MAP programs to be written and tested one at a time. This, in turn, provided a mechanism for building realistic workloads based on actual, running MAP programs. However, it did not accurately model MAP in a number of other respects, e.g. multiple control units were not simulated, memory conflicts were ignored, shared data bus conflicts were ignored, etc. The interpreter provides an opportunity for one MAP program to execute in a noncompetitive environment.

The overall system performance is dictated by the handling of competitive situations. The second level of simulation models handles this aspect of system operation. As programs were interpreted, a simulated monitor was used to obtain program performance statistics; these measures were then available as input data to the second level models.

The First Level Model

The MAP program interpreter, called MAPSIM, is a complex simulation program used to interpret a single MAP program stored in a pseudo main memory in absolute binary format. Thus, it is used to model a single control unit and an arbitrary number of processing elements, allowing actual programs to be written and executed in a noncompetitive

environment (i.e. resources are always immediately available as required). The input data to MAPSIM includes a designation of the number of processing elements to be used, the amount of processing element memory required, and a MAP program in absolute binary form. MAPSIM is written in the assembly language of the Control Data 6400 (called COMPASS) since it is desirable that the program be as efficient as possible during execution. This is necessary due to the large degree of parallel processing element activity that must be simulated on the sequential Control Data machine. Even with MAPSIM coded in assembly language, the simulated time/real time ratio is much greater than one. (This ratio is dependent on the number of processing elements being simulated and on the activity of the set of processing elements with respect to the set of operations being broadcast by the control unit.)

Consider the action taken for each MAP instruction execution. First, MAPSIM fetches an instruction from the pseudo main memory; this instruction is then decoded into a set of actions that the active processing elements must execute. The operations required to simulate the effect of a machine instruction do not directly correspond to the theoretical set of operations broadcast over the instruction bus as described previously, and thus the utilization of the instruction bus is not modeled. The set of processing elements currently allocated to a processing element is separated into two lists; the first list chains together all currently active processing element descriptors, and the second list includes the remaining processing elements. Therefore, to apply the instruction to the set of active processing elements, the list of active processing elements must be traversed.

All MAP program input and output (to and from the main memory) is accomplished by including a FORTRAN subroutine, within MAPSIM. The MAP program then makes "supervisory calls" which are passed on to the FORTRAN input/output routine. Again, a portion of the target system is modeled at substantially less detail than other portions.

MAPSIM is used to execute a MAP program, producing user-defined output. This has proven to be a worthwhile approach, since it has provided a medium for writing and testing a diversity of programs that illustrate several application areas for MAP. Additionally, MAPSIM allows one to write monitoring routines to simulate software and hard-

ware monitors. MAPSIM is designed to call a subroutine named MONITOR at the completion of each instruction cycle.* If the user does not wish to monitor his program, the default MONITOR returns control to MAPSIM. Once the monitor has been called, it is free to inspect any of the tables maintained by MAPSIM, e.g. the active processing element list, the simulated time, whether or not the instruction made a data reference to main memory, whether or not the instruction required the data bus, etc. The output provided by the monitoring routine may be defined by the user. He may generate a set of full trace data, a partial trace, or merely collect statistics to define distributions reflecting the character of the MAP program.

Although MAPSIM has been carefully written in assembly language to minimize the required execution time, the monitoring routines will usually be written in some combination of COMPASS and FORTRAN. Although these routines may be called very frequently, they will perform only a minimal amount of computation. They must also be able to do flexible I/O. Because of these properties, and since monitoring programs will frequently change, high level languages should be used to implement the simulated monitors.

As an example of the use of the monitoring routines, consider the problem of determining the load on the interface between the control unit and the main memory as generated by a given program. One can distinguish between instruction fetches and global data references to the main memory. In order to determine the memory load, then, the monitor must examine the activity caused by each instruction executed on the control unit. After each instruction is processed, its execution time is looked up in an operator table. This time is used as the time since the last instruction fetch, and the data can be written to an auxiliary file to generate the trace of instruction fetches or it can be entered into a data structure to generate a distribution of the frequency of instruction fetches.

In order to determine the global data reference distribution, one must determine if the current instruction referenced memory or not.

---

* It is also possible to have MAPSIM call MONITOR only after certain instructions have been executed. This option is invoked by reassembling MAPSIM with appropriate assembly-time options set.

This information can also be saved in a table. If the instruction does not reference data, the instruction time is added to an accumulator that keeps track of the time since the last data reference; otherwise, a portion of the instruction execution time for this instruction is added to the time since the last reference and then the sum is entered into a distribution. Again, this data could be used to generate a full trace of data references.

MAPSIM has been a useful tool for analyzing single program execution. It has allowed us to explore such potential application areas as numerical mathematics, operating systems, and operations research. Furthermore, we have been able to expose several weaknesses in the original instruction set for the machine; this has resulted in the implementation of a second version of the assembler and interpreter.

The first level model does not incorporate resources that are shared among two or more (programs executing on) control units, and thus, does not model this resource competition. There are two possible approaches to study this competition, given that MAPSIM is available. The first is to increase the level of detail of MAPSIM so that it is cognizant of shared resources, incorporating other control units into the interpreter. The second approach is to use a second level of models, (driven by measurement data from MAPSIM), which are less detailed in terms of program function, but more detailed in that the utilization of shared resources is taken into account. The first approach has the advantages of added realism and the ability to model inter control unit communication mechanisms (e.g. testing variants of Dijkstra's semaphore operations by independent processes). The disadvantage to the first approach is that the interpreter becomes more and more complex, and hence more difficult to build. It is possible to make a convincing argument that MAPSIM accurately simulates the action of a single control unit and a set of processing elements; if the program is to use quasi-parallel techniques to model multiple control units, the memory system, and the shared data bus, it is significantly more difficult to convince oneself that the code does what is intended. Despite the disadvantage of complexity, we are currently implementing a multi control unit interpreter. It will be used primarily to study synchronization and communication aspects of MAP and

not for resource utilization studies. Only if a continuing argument can be made that the multiple control unit interpreter is "correct" will it be used for resource utilization studies.

Second Level Models

Second level models correspond to simulation models that are commonly used to predict the performance of the extension of an existing system. Such studies typically model an existing system that has been reconfigured to employ a different CPU, more memory, different peripherals, etc. The second level models for the MAP study are analogously based on the first level model (MAPSIM), rather than existing hardware. Second level models incorporate less detail in terms of control unit and processing element operation, i.e. they do not simulate instruction execution. They incorporate more detail in the sense that shared resources are modeled in order to investigate the system-wide effects of resource competition. For example, a second level memory conflict model might include components to represent the memory system, but not the contents of memory locations; multiple control unit memory access patterns, but not instruction execution; I/O subsystem memory access patterns, but not data items; and a component to represent data paths between the other components, which is excluded from the first level model.

This class of simulation models is preferred to a multiple control unit interpreter because it is less expensive to exercise, and it is easier to argue that it performs the desired functions. On the other hand, validity may be more questionable since the level of detail of a second level model ignores certain facets of operation.

These higher level models are intended for studying individual aspects of the system. One should be able to write several second level models to investigate the effect of shared data bus competition, resource allocation algorithms, operating system strategies, etc. While it was crucial that MAPSIM be as efficient as possible, the emphasis on higher level models is for ease in preparing and testing. Therefore, the second level MAP programs have all been written in higher level programming languages, including FORTRAN and special purpose simulation languages.

The two level approach has had a decided advantage over other techniques in the area of representing the job load. Individual programs are executed one-at-a-time on MAPSIM in conjunction with an appropriate monitor. In most cases, the monitors have gathered precise trace information describing when the program requires access to shared facilities, (in the other cases probability distributions were built to represent program activity). After a sufficient number of programs have been interpreted in the noncompetitive MAPSIM, the traces for the programs are used to drive the second level model. Gathering trace data is extremely simple compared to the work involved with monitoring a prototype system; the monitors are quickly written, and they record the exact events required by the higher level model. If monitor data needs to be filtered before being used, the monitor can do all filtering on-line; no artifact will be introduced, since the first level simulation will allow all monitoring to be totally invisible to the monitored program.

Introducing parallelism and resource competition at the second level is easily handled with monitor traces from MAPSIM. Trace data for each program is written to a separate file; each record on the file contains an event, a MAPSIM time of occurrence, and other miscellaneous information. As each record is read from all trace files, appropriate action is taken and a simulated clock for each program is updated. Whenever a shared resource conflict occurs, the model resolves the conflict, allowing one program to continue, and the other to be blocked until the first completes its use of the resource. The first program can then be resumed with the total blocked time added to the recorded time of occurrence for all succeeding events. A detailed example of the use of this technique is given in reference [11].

CONCLUSION

The two level simulation approach to the design of the Multi Associative Processor has provided a cost-effective means of analyzing several aspects of the performance. The first level has been used to write and test a wide variety of MAP computer programs, and has allowed us to monitor these programs in very general ways without introducing artifact at the lowest level. This approach has the advantages of a

software monitor in generality and of a hardware monitor in non-interference. The total computing budget for implementing MAPSIM, writing MAP programs, monitoring MAP programs, and implementing second level models to study memory conflict, shared data bus conflict, resource allocation algorithms, and operating system comparisons was less than $6,000.

The work has allowed us to test a variety of machine designs, including two distinct instruction sets for MAP all within two years at an effort of approximately 12-15 man-months distributed over an average of 1.5 graduate research assistants and an investigator working 10% of the time during the academic year and for two months each during two summers.

Finally, the simulation studies have now come to the point where a prototype MAP system can be built with good ideas about how it will perform.

ACKNOWLEDGEMENT

The author wishes to thank the National Science Foundation for its support of this work under grant number MCS74-08328 A01. The following graduate students have also contributed much in their respective efforts: Roger Arnold, John Burke, Karl Williamson, Heinrich Siegmann, and Bob Palasek.

REFERENCES

[1] Lucas, H. C. Jr., "Performance Evaluation and Monitoring", ACM Computing Surveys, Vol. 3, No. 3, pp 79-91, September, 1971.

[2] Agajanian, A. H., "A Bibliography on System Performance Evaluation", IEEE Computer, Volume 8, No. 11, pp 63-74, November, 1975.

[3] --, Proceedings of the Symposium on the Simulation of Computer Systems, sponsored by ACM SIGSIM and National Bureau of Standards, 1973-1976.

[4] Merikallio, R. A. and F. C. Holland, "Simulation Design of a Multiprocessing System", Proceedings of the Fall Joint Computer Conference, Volume 33, pp 1399-1410, 1968.

[5] Teorey, T. J., "Validation Criteria for Computer System Simulation", Proceedings of the Symposium on the Simulation of Computer Systems, pp. 161-173, 1975.

[6] DeCegama, A., "A Methodology for Computer Model Building", Proceedings of the Fall Joint Computer Conference, Vol. 41, pp. 299-310, 1972.

[7] Rose, C. W., "LOGOS and the Software Engineer", Proceedings of the Fall Joint Computer Conference, Vol. 41, pp 311-323, 1972.

[8] Sreenivasan, K. and A. J. Kleinman, "On the Construction of a Representative Synthetic Workload", Communications of the ACM, Vol. 17, No. 3, pp 127-133, March, 1974.

[9] Nutt, G. J., "A Parallel Processor for Evaluation Studies", Proceedings of the National Computer Conference, Vol. 45, pp 769-775, 1976.

[10] Arnold, R. D. and G. J. Nutt, "The Architecture of a Multi Associative Processor", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-070-75, 45 pages, June, 1975, (revised October, 1976).

[11] Nutt, G. J., "Memory and Bus Analysis of an Array Processor", to appear in IEEE Transactions on Computers.

[12] Nutt, G. J., "Some Resource Allocation Policies in a Multi Associative Processor", Acta Informatica, Vol. 6, pp 211-225, 1976.

[13] Nutt, G. J., "A Parallel Processor Operating System Comparison", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-085-75, 25 pages, December, 1975.

[14] Nutt, G. J., "Measuring User Programs for a SIMD Processor", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-089-76, 30 pages, April, 1976.

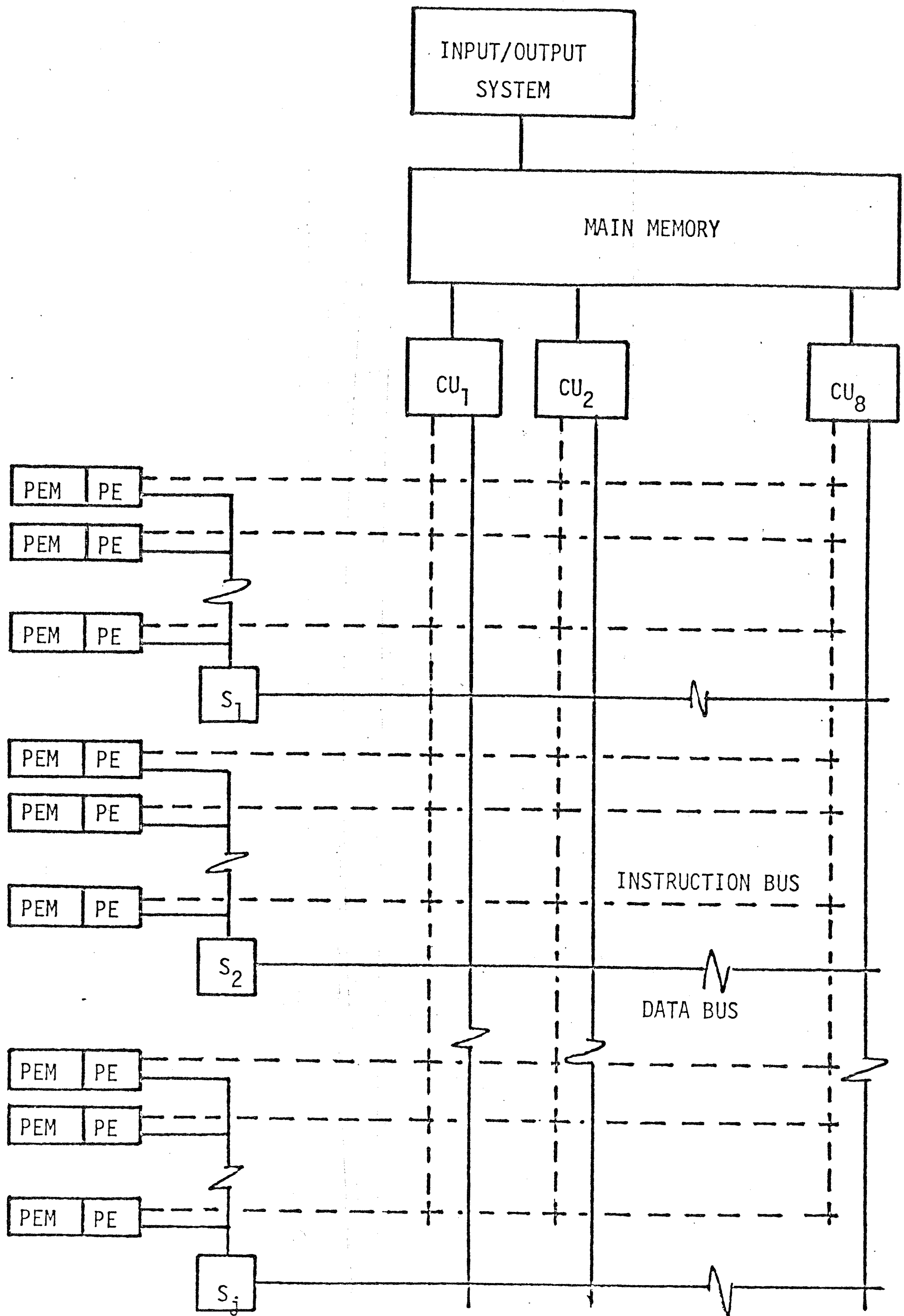

FIGURE 1