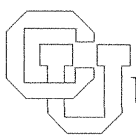
# Measuring User Programs for a SIMD Processor *

**Gary J. Nutt**

**CU-CS-089-76**



**University of Colorado at Boulder**
**DEPARTMENT OF COMPUTER SCIENCE**

* Supported by NSF NYI #CCR-9357740, ONR #N00014-96-1-0720, and a Packard Fellowship in Science and Engineering from the David and Lucile Packard Foundation.

# Measuring User Programs for a SIMD Processor*

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

Report #CU-CS-089-76 April, 1976

*This work was supported by the National Science Foundation under Grant number MCS74-08328 A01 (formerly number GJ-42251)

Index Terms: measurement and evaluation, parallel processors, SIMD processors, user program measurements.

ABSTRACT

Measurements to aid user programmers in writing efficient programs for a single-instruction-stream, multiple-data-stream (SIMD) computer such as ILLIAC IV, PEPE, or MAP are described. Applications of some of these measurements to determine when a program could effectively be implemented on a conventional sequential computer are also discussed. An archetype hardware monitor to obtain the desired measurements is also outlined.

## INTRODUCTION

A recurring problem encountered by computer programmers is the determination of the performance of their code on a sequential processor, [1]. A variety of performance monitors have been discussed in the literature which can aid the programmer in understanding how effectively his program is using the system resources available to him, [2]. The simplest form of performance feedback is a distribution of memory addresses referenced during the instruction fetch cycles of the execution of the program. The programmer can use this information to detect heavily-used portions of the code and either optimize the code at these locations, or reorganize the computation to obtain better overall performance.

Programs that are executed on a parallel processor system may require additional performance measures in order to gain an understanding of the use of multiple processing units. If the parallel processor is a general purpose multiprocessor, it is likely that the additional information is of little use to the programmer, since he has no direct control of the processor scheduling mechanism. If, however, the parallel processor is a single-instruction-stream, multiple-data-stream (SIMD) architecture, the user may be able to exert a great deal of influence as to the use of the multiple processing elements that perform computations on the data streams. It is this latter class of machines on which this measurement study concentrates.

In organizing a computation for a SIMD machine, it is sometimes the case that the number of data streams exceeds the number of processing elements. This problem has been frequently discussed, especially with respect to the 64 processing element array of the

ILLIAC IV, (e.g. see [3]). With the LSI technology improvements in the last few years, it seems plausible that future SIMD processors will be able to incorporate much larger numbers of processing elements, and that the severity of the restriction mentioned above will be reduced substantially. (Even so, there will always be some problems that require more processing elements than exist in any given machine.) Because of the previous work done on this problem and because of the technology trend, this study concentrates on other aspects of program measurements in a SIMD environment; in the analysis technique that follows it is assumed that there are enough processing elements to execute a computation at least once where each data stream employs its own processing element. After monitoring the computation in this configuration, data can be gathered to indicate if the computation might as effectively be run using fewer processing elements. "Fewer" might prove to be one, implying that the computation could be carried out on a sequential processor as effectively as on a SIMD processor.

Implementing algorithms on a SIMD processor in assembly language has, in our experience, proven to be a formidable task. The main difficulties have not been so much in recognizing where SIMD parallelism exists in an algorithm, but rather, in keeping track of the utilization of processing elements as they perform computations on the data streams. The immediate problem has often been a coding problem rather than a high level algorithm design problem. Much of this difficulty could be alleviated with the use of a good high level language for the machine. Even if a high level language were available, the need for performance monitoring of programs would not vanish. By the nature of data streams and

the data structure chosen to write a program, processing element utilization and instruction utilization are not as well understood by the programmer, as is the case in high level language programs written for sequential processors.

The objectives of the study described here are to derive a set of measures for SIMD programs that can be used to decide when more than one data stream can be handled by one processing element; and to investigate measures that indicate the utilization of machine resources so that the programmer can tune his SIMD program.

The medium for this study is a hypothetical SIMD machine for which an interpreter is used to exercise programs. A brief discussion of this machine is given in the next section, and similarities between it and other array processors are pointed out. In subsequent sections, performance measures and their uses are described, followed by a description of a performance monitor that can take the desired measurements on a real machine. Finally, some experiments using the performance measures are described.

## THE SIMD PROCESSOR

The machine used for the monitoring studies is the Multi Associative Processor (MAP) system. The details of the architecture have been discussed elsewhere, [4,5], and only a brief description is provided here to provide context for the remainder of the paper.

MAP is composed of a Main Memory System to store the instruction stream(s) and global data used by all processing elements (PEs), (see Figure 1). The Main Memory also acts as a buffer for loading data streams from peripheral devices into PE memories (PEMs), where they will be used for SIMD computation. The system incorporates eight control units (CUs) each of which can decode an instruction stream loaded in the Main Memory. After decoding an instruction, a series of commands is broadcast over the Distribution Switch to a subset of the 1024 PEs that have been previously allocated to the broadcasting CU. These commands cause the PE subset to perform a computation on data loaded in the respective PE memories. PEs are treated as dynamically allocatable resources to the CUs, thus effective utilization of PEs by a CU is important to the overall performance of the system. Since this paper is concerned with monitoring individual programs to be executed on MAP, further discussion of the architecture describes only points concerned with one CU and a subset of from 1 to 1024 PEs. The details of CU coordination, Main Memory design, and the organization of the Distribution Switch are of no concern to this study, since they are invisible to (noncooperating) programs.

Although some instructions of a SIMD program are intended to apply to all data streams, others apply only to some of the data streams. A mechanism for handling this situation is an activity flag

associated with each PE; if the flag is set by an associative instruction, the corresponding PE participates in all computations (on its data stream) until an associative instruction resets the activity flag. Flag setting and resetting in an associative instruction can be determined by data dependencies, as in ILLIAC IV[6,7], or by a logical combination of past data dependencies saved in a SELECT register local to each PE. It is the presence of this SELECT register that gives rise to the term "associative" in the name of the machine. The result of allowing selective activity of each PE is that there is a sequence of times during which the PE is active, (and a complementary sequence of inactive times). The activity sequence determines the utilization of a particular PE.

In order to test MAP programs, an interpreter for an assembly language for MAP has been implemented on a Control Data 6400. Symbolic assembly language programs are prepared, assembled, and then executed by the interpreter. At the end of each instruction cycle, the interpreter invokes a simulated monitor so that measurements can be taken on the program being interpreted. All information kept by the interpreter, such as instruction counter contents, activation status of each PE, etc., is available to the monitor. The monitor used for this study is described later in the paper.

Although the MAP architecture differs in details from other current SIMD machines, the interpreter handles one CU and a subset of PEs in a manner similar to program execution on the ILLIAC IV [6, 7] and PEPE[8]. From the viewpoint of a programmer, the measurements and evaluation apply equally well to any other SIMD machine of this type as to MAP. Each machine uses the idea of an activity flag to determine the status of a

PE, and each CU operates roughly in the manner described above. One difference is that the motivation for effective PE utilization may not be as strong in machines that employ one CU as it is in MAP, since all PEs can be used only by one CU at a time in the former case.

## PERFORMANCE MEASUREMENTS AND THEIR USES

### Program Tuning

There are some measurements found to be particularly useful for tuning programs on sequential processors that are also useful on SIMD processors. The distribution of memory references by the instruction counter is one of those measures. By producing a histogram of Main Memory locations versus the number of instruction references to the respective locations, the programmer can determine localities in his code, and the relative use of each locality. As in the case of sequential processor programs, the programmer should use this information to carefully inspect these parts of the program to be sure that the code is as efficient as possible. The more frequently a given portion of code is executed, the more careful attention should be paid to optimization and PE utilization.

The instruction reference histogram can be obtained in a number of ways; all of these methods require inspection of the instruction counter through a sampling technique with a software monitor, or through continuous monitoring of the register with a hardware monitor, [9].

Another simple measure used in sequential processors is the frequency with which each operation code is used. As mentioned by Drummond, [10], and Lucas, [11], this operation code frequency is primarily used to analyze instruction mixes for machine selection. In the

SIMD context, operation code frequencies are useful to determine the number of arithmetic or logical operations compared to the expected number of such operations on a sequential processor. It also gives the programmer an indication of functional unit utilization, and the frequency of PE activity sequence changes through the use of associative instructions. This measure is not especially useful to the SIMD programmer, but it is easily obtained and can indicate some trends of utilization in the code. As in the case of instruction counter monitoring, operation code frequencies can be determined by sampling or continuous monitoring of the instruction word being decoded by the CU. In most cases, the instruction word is not program addressable, and so a hardware monitor is required for obtaining the measure.

The extended measures for SIMD processors considered here all have to do with PE utilization. The simplest figure of merit is the average number of PEs that were active throughout the computation. In order to obtain this measure, the following approach is taken: Assume that a computation consists of n tasks, $T_1, T_2, \ldots, T_n$ each corresponding to the execution of some portion of the instruction stream on one of n data streams. By a previous assumption, each task is to be carried out on a unique PE, thus the computation requires n PEs. During the computation, $PE_i$ is assumed to have been allocated to the CU at time $t_o$ and deallocated at time $t_{2m+1}$ (for $1 \leq i \leq n$). Each $PE_k$ executing $T_k$ will then be active at a sequence of time blocks, $\omega(T)$ denoted as

$$\omega(T) = [t_1, t_2], [t_3, t_4], \ldots, [t_{2m-1}, t_{2m}]$$

i. e. $PE_k$ executing $T_k$ is inactive during the times

$$[t_o, t_1), \ldots, (t_{2i}, t_{2i+1}), \ldots, (t_{2m}, t_{2m+1}].$$

The amount of time that $PE_k$ is actively computing $T_k$ is expressed as

$$|\omega(T_k)| = \sum_{i=1}^{m} (t_{2i}-t_{2i-1})$$

Thus, the expected number of active PEs is

$$\bar{\alpha} = \frac{\sum_{k=1}^{n} |\omega(T_k)|}{t_{2m+1}-t_o}$$

In order to determine $\bar{\alpha}$, a monitor must determine the allocation/deallocation times, ($t_o$ and $t_{2m+1}$), and the activity sequences ($\omega(T_k)$). These data can be gathered by an interrupt-intercept software monitor, [9],triggered by allocation/deallocation events and activation/deactivation events, or by a hardware monitor as discussed in the next section. In any case, a more descriptive representation of program utilization of the PEs is a two dimensional plot of the number of PEs active versus CU processing time, as used by Lloyd and Merwin, [12].

Although $\bar{\alpha}$ and the two dimensional plot indicate PE utilization, they do not suggest where to look into the code to improve utilization. To accomplish this, PE utilization should be correlated with memory locations containing instructions being executed. An extension to the instruction memory reference monitor provides the needed insight. At the time the instruction counter is inspected to build the instruction fetch histogram, a monitor should also determine the number of PEs active and enter this number into an array indexed by the instruction counter. This allows the user to determine the average number (and maximum number) of PEs active when a given block of instructions is being executed; thus the code can be tuned for PE utilization as well as operation code execution in heavily-used portions of the program.

As in obtaining data to determine $\bar{\alpha}$, these measures imply that a monitor is able to inspect the activity flags of each PE involved in the computation.

Comparative Performance

The measures described above are concerned with SIMD program tuning; one would also like to obtain measures that give an indication of when two or more tasks should be implemented on one PE. In considering such measures, it is useful to first make some basic observations about parallelism in computer programs.

The execution of a program may result in a number of tasks that can be done in parallel, e. g. at the expression level, two or more functional units could be performing addition and multiplication in parallel as is possible in the Control Data 6600 CPU [13]. In other programs, distinct statements or tasks could be executed in parallel on multiple processors where each task is concerned with a possibly different computation. Both of these cases are examples of independent parallelism, i.e. the parallel tasks are independent computations. For parallelism within programs on a SIMD processor, simultaneous computation for two or more tasks can take place only if the single instruction stream can be applied to multiple data streams; independent parallelism will not necessarily result in simultaneous operation on a SIMD machine, and in general will result in sequential operation. Thus, SIMD parallelism is a restriction of independent parallelism, and is the property that needs to be emphasized for machines like MAP, ILLIAC IV, and PEPE. Attention is now turned to the investigation of SIMD parallelism in programs.

Let $\omega(T) = [t_{2i-1}, t_{2i}][t_{2i+1}, t_{2i+2}]\ldots[t_{2k-1}, t_{2k}]$

be a partial activity sequence of the task T on some PE. If there exists T' such that $\omega(T)$ and $\omega(T')$ are partial activity sequences where $\omega(T) = \omega(T')$, then T and T' are SIMD-equivalent for the time period $(t_a, t_b)$ where $t_a = \max(t_{2i-1}, t_{2i'-1})$ and $t_b = \min(t_{2k+1}, t_{2k'+1})$. Denote the set of tasks that are SIMD-equivalent for $(t_a, t_b)$ as

$$\{T\}_a^b = \{T' \mid T \text{ and } T' \text{ are SIMD-equivalent for } (t_a, t_b)\}$$

Now, let $\{T_i\}_a^b$ be the partitions generated on $T_1, \ldots, T_n$ by SIMD-equivalence. If $|\{T_i\}_o^{2m+1}| = n$, then the program is full SIMD parallel, i.e. all PEs are active at exactly the same times, and inactive at the same times.

Only a hygenic example is likely to illustrate full SIMD parallelism; a measure of distance between two activity sequence can be used to investigate the amount of SIMD parallelism that exists in a program. Let $\omega(T)$ and $\omega(T')$ be full activity sequences; then the sets $\omega(T) \cup \omega(T')$ and $\omega(T) \cap \omega(T')$ are well-defined. The activity distance, of T to T', denoted d(T,T') is

$$d(T,T') = |\omega(T) \cup \omega(T') - \omega(T) \cap \omega(T')|$$
$$= |\omega(T) \cup \omega(T')| - |\omega(T) \cap \omega(T')|.$$

The activity distance is the length of the symmetric difference of $\omega(T)$ and $\omega(T')$, i.e. it is an accumulation of the times at which T is active (inactive) and T' is inactive (active). The activity distance has the following properties:

- $d(T,T') = d(T',T)$
- $d(T,T') = 0 \iff \omega(T) = \omega(T')$
  $\iff$ T and T' are full SIMD-equivalent

- $d(T,T')=|\omega(T)\cup\omega(T')|=|\omega(T)|+|\omega(T')| \Leftrightarrow \omega(T)\cap\omega(T)=\emptyset$
- $o \leq d(T,T') \leq |\omega(T)\cup\omega(T')|$
- as $d(T,T')$ tends to 0, the amount of SIMD parallelism between T and T' increases.
- as $d(T,T')$ tends to $|\omega(T)\cup\omega(T')|$, the amount of sequential operation of T and T' increases.

Before using the activity distance as a measure of efficiency note that

$$|\omega(T)\cup\omega(T')| = |\omega(T)| + |\omega(T')| - |\omega(T)\cap\omega(T')|$$
$$= |\omega(T)| + |\omega(T')| - |\omega(T)\cup\omega(T')| + d(T,T')$$
$$= \frac{|\omega(T)| + |\omega(T')| + d(T,T')}{2}$$

and

$$|\omega(T)\cap\omega(T')| = |\omega(T)\cup\omega(T')| - d(T,T')$$
$$= \frac{|\omega(T)| + |\omega(T')| + d(T,T') - d(T,T')}{2}$$
$$= \frac{|\omega(T)| + |\omega(T')| - d(T,T')}{2}$$

If $d(T,T')$ is tending toward $|\omega(T)\cup\omega(T')|$, then the quantitative effect of executing tasks T and T' on one PE can be computed. The fractional PE utilization of executing each task on its own PE is the fraction of the time when one or both of the PEs are active, and is expressed as

$$\frac{|\omega(T)\cup\omega(T')|}{t_{2m+1} - t_o}$$

assuming that both PEs are allocated and deallocated at the same time. The fractional utilization of executing both tasks on one PE, U(T,T'), is expressed as

$$U(T,T') = \frac{\max(|\omega(T)|,|\omega(T')|)+|\omega(T)\cap\omega(T')|}{t_{2m+1}-t_o}$$

$$= \frac{\max(|\omega(T)|,|\omega(T')|)+\frac{|\omega(T)|+|\omega(T')|-d(T,T')}{2}}{t_{2m+1}-t_o}$$

and if, for convenience, it is assumed that $|\omega(T)|\geq|\omega(T')|$,

$$U(T,T') = \frac{3|\omega(T)|+|\omega(T')|-d(T,T')}{2(t_{2m+1}-t_o)}$$

Then, the ratio of utilizations, S(T,T'), can be expressed as

$$S(T,T') = \frac{2|\omega(T)\cup\omega(T')|}{3|\omega(T)|+|\omega(T')|-d(T,T')}$$

$$= \frac{|\omega(T)|+|\omega(T')|+d(T,T')}{3|\omega(T)|+|\omega(T')|-d(T,T')}$$

where $|\omega(T)|\geq|\omega(T')|$. When $S(T,T')>1$, fractional utilization is better using only one PE, than with using two PEs. This expression ignores the additional overhead involved with implementing both tasks on one PE; to include overhead involved with h "context changes" between the two task, each requiring g additional time units,

$$U'(T,T') = \frac{3|\omega(T)|+|\omega(T')|-d(T,T')+2hg}{2(t_{2m+1}-t_o)}$$

and

$$S'(T,T') = \frac{|\omega(T)|+|\omega(T')|+d(T,T')}{3|\omega(T)|+|\omega(T')|-d(T,T')+2hg}$$

Hence, two tasks, T and T', can be merged with no loss in fractional utilization if

$$S'(T,T')>1$$

$$\text{i.e.,} \quad \frac{|\omega(T)|+|\omega(T')|+d(T,T')}{3|\omega(T)|+|\omega(T')|-d(T,T')+2hg} > 1$$

$$\text{and} \quad |\omega(T)|+|\omega(T')|+d(T,T')>3|\omega(T)|+|\omega(T')|-d(T,T')+2hg$$

and thus

$$d(T,T')>|\omega(T)|+hg$$

implies that T and T' should be executed on the same PE, on the basis of activation sequences.

The above analysis ignores a crucial aspect of programming on the computer, viz. data dependencies determine the resource utilization. What the analysis shows is that for the execution of a program on a given set of data, conditions can be found under which two tasks can be executed on the same PE.

Given that one accepts the applicability of the above analysis despite data dependencies, the effect of combining many (all) tasks onto one PE can be computed. Define $\omega(T+T')$ to be the resulting activity sequence of executing T and T' on one PE, and thus, let $\omega(\sum_{i=1}^{n} T_i)$ be the activity sequence for executing n tasks on one PE. Then, $d(\sum_{i=1}^{n-1} T_i, T_n)$ can be computed and compared to

$$\left|\sum_{i=1}^{n-1} \omega(T_i)\right|+(n-1)hg$$

to test the condition for merging all tasks on to one PE. if

$$d\left(\sum_{i-1}^{n-1} T_i, T_n\right)>\left|\sum_{i-1}^{n-1} (T_i)\right|+(n-1)hg$$

then the computation can be done as effectively on one PE as on n PEs, and should be executed on a sequential processor rather than an SIMD machine.

To carry out the evaluation, it is necessary to measure programs for their activity sequences. This may result in a volumous amount of data, and the processing of these activity sequences to obtain distances is an algorithm that requires on the order of $n^2$ computations for an n data stream program. Thus, the cost of analysis can only be justified if the program is to be used repeatedly after it has been executed at least once with one PE per data stream.

## A POSSIBLE USER PROGRAM MONITOR

There are basically three classes of monitors that can be used for obtaining performance data: software monitors, hardware monitors and hybrid monitors, [9]. Because system software and the machine instruction set depend heavily on the machine, software monitors will not be discussed here; knowledge of these components is also necessary to design a hybrid monitor. However, a hardware monitor to take the measurements described in the previous section can be designed based on the architectural description given in an earlier section.

The goals of this monitor are to measure and record the following variables:

- CU instruction counter contents
- CU instruction word operation codes
- PE activity flags

These measurements allow one to generate:

- CU instruction counter histogram
- Operation code use
- Average number of PEs active, $\bar{\alpha}$
- PE activity versus instruction address
- Number of PEs active versus processing time
- PE utilization
- Activity sequences
- Activity distances.

The hardware monitor that is sketched out in this section will be considered to be a case of "overkill" by some readers, but it is justified for the following reasons:

- It is extendable to a more complete monitor for system monitoring of MAP (as opposed to user program monitoring).
- It is applicable to SIMD machines other than MAP.
- It illustrates that the measurements indicated in the previous section can be made by some monitor.

Because of the cost trend in semiconductor memories, free use has been made of small memory modules. An unspecified processor (i.e. a minicomputer or a microcomputer)is also assumed to implement many of the monitoring tasks, although the detailed algorithms for such a processor are not discussed.

Figure 2 is a diagram of the hardware monitor; it is composed of six modules to accomplish:

- Instruction counter monitoring for all instructions.
- Instruction counter monitoring for associative instructions.
- Operation code monitoring.

- PE activity flag monitoring for status
- PE activity counting
- Monitor supervision and mass storage handling.

A brief discussion of each module follows.

Module A: Instuction Counter Monitor

In monitoring the instruction counter values, the following assumptions are made about the Main Memory System: An 18-bit address is used to reference a maximum of 128K Main Memory words; each instruction counter content is an address relative to a base register, (such as the RA register in the Control Data 6000 series computers, [13]), where the base register is internal to the CU; the content of the base register can be made known to the hardware monitor. These assumptions imply that the instruction register contents are relative to address 0, making it unnecessary to perform a transformation on the address based on where a CU's program is loaded in Main Memory. If the length of the program is also known, the position of the 12 most significant (nonzero) bits of an instruction address can be determined. The cycle time of the CU for MAP is assumed to be approximately 100 ns, and the minimum number of such cycles required to execute a MAP instruction is two cycles, with the average number of cycles being about 6. Module A uses the specified 12 bits of the instruction counter to address a 4Kx16 bit random access memory (RAM) which has a memory cycle time less than or equal to 200 ns(2 cycles). After the address is determined, the content of the corresponding memory location is incremented by one. This is a rapid operation and, in all cases must be performed before the instruction counter is changed to reflect the next Main Memory address used by the CU. In the case of overflow, one of two strategies

can be employed, (not specified here); either the CU can be blocked while the monitor processor, Module F, dumps the RAM contents to mass storage, or Module A can be disabled, losing monitor data as the CU continues operation, while Module F unloads the RAM. The RAM contents specify the instruction counter histogram.

Module B: Instruction Counter for Associative Instructions

Module B operates in exactly the same manner as Module A, except that counts are made only when the current operation code is an associative instruction as determined by Module C. This is used to specify the number of instruction counter samples that also cause the number of active PEs to be monitored by Module D.

Module C: Operation Code Monitor

PE status need only be monitored when the status changes due to an associative instruction. Assuming an 8-bit operation code as used in MAP, an 8x256 decoding network is used to detect associative operation codes. The result of detecting an associative instruction is used by Module B as mentioned above, by Module D to sample PE activity flags, Module E to monitor the number of PEs active at any given time, and by Module F to save a record of information indicating the activity sequence of each PE on a mass storage device. Since Module C must employ this decoder, a table of operation code frequencies can easily be generated by including a 256x16 bit RAM and increment unit similar to that used in Module A. Overflow of a word in the RAM is again assumed to be handled by Module F as described above.

Module D: PE Activity Monitor

Because the set of 1024 PEs in MAP can be dynamically allocated to any of eight CUs, a mechanism must be included in the monitor to

save only the status of those PEs currently allocated to the CU executing the target program. MAP must also resolve such problems, and the approach is to include an 8-bit ID register in each CU and an 8-bit OWNER register in each PE. If $PE_j$ is currently allocated to $CU_i$, then the OWNER register of $PE_j$ matches the ID register of $CU_i$. In Figure 2, Module D incorporates 8K exclusive-NOR gates and 3K AND gates to compare ID-OWNER registers and to multiplex the activity flags of allocated PEs into a 1024 bit activity register. Since associative operations occur relatively infrequently, the gate count could be reduced considerably by sharing ID-OWNER recognition hardware among j PEs. Then, only 8i exclusive-NOR gates and 3i AND gates are required to set statuses in an i-bit activity register, where ixj=1024. But, the activity register would have to be loaded, (and processed by Module F as indicated in Figure 2), j times for each associative instruction.

Module E: PE Activity Counter

Since Module F must ultimately store the trace data from the activity register settings determined by Module D, it is a simple matter to have Module F count the number of PEs active each time Module D is invoked, and write the integer value to a PE count register within Module E. Using the instruction address determined by Module A, another 4Kx16 bit RAM is addressed so that the content of the PE count register can be added to the corresponding location to keep a cumulative sum of the number of PEs active when the program is executing an associative instruction from a given area of Main Memory. Used in conjunction with data stored in the RAM in Module B, the average number of PEs active, indexed by the instruction counter, can

be computed by a postprocessor. Overflow must again be handled by Module F, as described in the Module A subsection.

Module F: Monitor Processor

Module F is composed of a general purpose, programmable minicomputer (possibly a microcomputer) that performs overflow processing for Modules A, B, C, and E as the need occurs. It is conceived as being invoked for overflow processing by an interrupt from one of the other modules, where processing consists of writing the contents of the given RAM onto a mass storage device, e.g. a 10 mbyte disk. The other primary function of the processor is to store activity register information, as determined by Module D, on the mass storage device. Each block of data written to the mass storage device would be prefaced by a header describing the semantics of the block and a real time clock reading. The only nonstandard interface required to the processor is the activity register. Note that Module F could easily be used for postprocessing whenever it is not being used to supervise monitoring.

An Alternative Hardware Monitor

It is clear that most of the functions of the modules of the monitor described above could be implemented on a minicomputer, provided that the measurement probes have a suitable I/O interface. The increment unit used in Modules A, B, and C is already included in most modern minicomputers. The most restrictive constraint is that of timing, i.e. the minicomputer would have to be able to retrieve information from an I/O bus, and add to memory in a time period that is shorter than the cycle time for most minicomputer memories. The existence of microprogramming to tailor these operations would almost be required.

## EXPERIMENTS WITH THE MEASUREMENTS

Although no hardware monitor for MAP has been constructed, (MAP hardware itself has not even been built), the functions described in the previous section have been implemented as a simulated hardware monitor invoked by the MAP program interpreter. The simulated monitor produces the following data about each program executed by the interpreter:

- Histogram of Main Memory References
- Operation code frequency count
- Histogram of average and maximum PE activity versus instruction counter address
- Trace data describing PE activity status for each PE.

From this trace data, postprocessors produce:

- Plot of PE activity for each PE versus processing time.
- $\omega(T)$ for each task T allocated to a PE.
- $d(T,T')$ for each task T, T' allocated to its own PE.

Some examples of user program monitoring on two MAP programs are now given. The first program implements the Gauss-Jordon algorithm, with full pivoting,for solving a linear system of equations. For a system of n equations and n unknowns, the program employs n+1 PEs, each storing a column of the coefficient matrix or else the column vector representing the right hand sides of the system. The program works for any system where n is less than or equal to 1023, and the results discussed here are for a problem with n=10,i.e.11 PEs are used.

The program occupies 68 Main Memory locations (multiple instructions are stored in each location) and requires 24,290 CU cycles to

input data into the Main Memory, load the 11 PEs, obtain a solution, and print the solution. Figure 3 shows the instruction counter histogram for the execution of the program. Memory locations 46-55 contain code to eliminate coefficients, and locations 23-32 contain instructions to determine the pivot element for full pivoting. The code to (sequentially) load PE memories is stored at locations 20-21. The applicability of MAP in determing the pivot element is apparent from the data. Operation code frequencies are not given here, since they have little meaning to those unfamiliar with the machine instruction set.

Figure 4 is the PE activity histogram where activity is correlated with instruction counter contents. The data indicates that no PEs are active for significant portions of the time in locations 46-55; however, all PEs are active at locations 48-49 within the code to eliminate coefficients. PE utilization is low on the average, but all 11 PEs are required for some parts of the computation. This is a common occurrence in an SIMD program, i.e. all PEs are not used all of the time.

As might be expected, the plot of PE activity versus processing time has so many status changes that in order to plot the data to show all such changes, the plot loses its descriptive value since it is too large. Although others have used this method to indicate PE utilization, we find it not too be very helpful.

Table 1 shows the lengths of activity sequences, $|\omega(T)|$, for all 11 tasks, and Table 2 shows $d(T,T')$. Here, $t_0=0$ and $t_{2m+1}=24{,}290$, thus the average number of PEs active is $\bar{\alpha}=8.74$ PEs, or about 79% of the PEs are expected to be active at any given time. Using Tables 1 and 2,

S(T,T') can be computed for each pair of PEs. Some sample values, where $T_{11}$ is the righthand side vector and $t_i (1 \leq i \leq 10)$ is the $i^{th}$ column vector of the coefficient matrix, are

$$S(T_1,T_2) = 0.53$$

$$S(T_1,T_{11}) = 0.57$$

S(T,T') for both cases implies that merging tasks onto a single PE will degrade performance. This is also observable directly by noticing that

$$d(T_i,T_j) \leq \max(|\omega(T_i)|,|\omega(T_j)|) \quad \text{for } 1 \leq i,j \leq 11.$$

We have ignored context switching time, (it could only make the inequality more severe).

The conclusion based on monitor data is that no two tasks should be executed on one PE, and that code/PE optimization efforts should be concentrated on the portions of the program stored in Main Memory locations 23-32 and 46-55.

As a second example, a program to compute a shortest weighted path between to nodes of a graph (where each edge has a weight) was monitored. The particular execution of the program searched a graph composed of 64 nodes, and required 64 PEs. The program used 114 Main Memory locations for instructions and required 72971 cycles to execute. In this program, the instruction counter histogram indicates that 38% of the references were to 15 locations (20-35) and 50% of the references were to another 19 locations (44-63). The first locality was to load the graph description into PE memories, and upon inspecting this code it was discovered that a relatively inefficient approach was taken to perform this loading. This proves to be costly computation since PE utilization is very low during loading (i.e., it is essentially a sequential process). The other heavily-used portion of code does take good advantage of PEs.

The expected number of PEs active at any given time was $\bar{\alpha}$=11.63 PEs, or about 18%. Even though the expected number of active PEs is only 11.63, only a few cases were observed where S(T,T') exceeds 1, e.g.

$d(T_1,T_{55})=9,953$

$\omega(T_1) = 1,239$

$\omega(T_{55})=9,778$

thus $S(T_1,T_{55})=1.02$.

This program represents a case where one would not likely combine tasks on a single PE at any rate, because the nature of the problem makes it impossible to detect which arbitrary tasks should be combined because of data dependencies. The power of the SIMD architecture is being used for a search process while subsequent computation will naturally result in a small fraction of PEs being active.

## SUMMARY

Several aspects of measuring user programs for a SIMD processor have been discussed. Some measures used in sequential processor programs have been shown to be useful for SIMD programs. Additional measures of PE utilization have been introduced to help the programmer understand how effectively this resource is being used. In order to convince the reader that the required measurements can be taken without undue difficulty, an outline of a hardware monitor has been provided.

Some conditions under which a given program operating on a given set of data were investigated, and it was shown how to determine if the program could have as effectively been executed on a sequential

processor. Although this represents an application of hindsight, it can help one recognize cases where more effective utilization of computing resources can be applied if patterns of utilization tend to repeat themselves. In some cases it will allow the SIMD programmer to detect cases where his code should be executed on a sequential processor.

ACKNOWLEDGEMENT

The author wishes to thank the National Science Foundation for support of this work under Grant Number MCS74-08328 A01 (formerly designated at Grant Number GJ-42251).

REFERENCES

[1] Saltzer, J. H. and J. W. Gintell, "The Instrumentation of Multics", Communications of the ACM, Vol. 13, No. 8, pp 495-500, August, 1970.

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

[3] Lawrie, D. H., "Access and Alignment of Data in an Array Processor", IEEE Transactions on Computers, Vol. C-24, No. 12, pp 1145-1155, December, 1975.

[4] Nutt, G. J., "A Parallel Processor for Evaluation Studies", to appear in AFIPS Proceedings of the NCC, Vol. 45, 1976.

[5] 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, 50 pages, June, 1975.

[6] Barnes, G. H., R. M. Brown, M Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV Computer", IEEE Transactions on Computers, Vol. C-17, No. 8, pp 746-757, August, 1968.

[7] Bouknight, W. J., S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Samed, and D. L. Slotnick, "The ILLIAC IV System", Proceedings of the IEEE, Vol. 60, No. 4, pp 369-388, April, 1972.

[8] Githens, J. A., "A Fully Parallel Computer for Radar Data Processing", NAECON '70 Record, pp 290-297, 1970.

[9] Nutt, G. J., "Computer System Monitors", IEEE Computer, Vol. 8, No. 11, pp 51-61, November, 1975.

[10] Drummond, M. E., Jr., Evaluation and Measurement Techniques for Digital Computer Systems, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

shorter than the cycle time

programming to tailor these

[11] Lucas, H. C., Jr., "Performance Evaluation and Monitoring", ACM Computing Surveys, Vol. 3, No. 3, pp 80-91, September, 1971.

[12] Lloyd, G. R. and R. E. Merwin, "Evaluation of Performance of Parallel Processors in a Real-time Environment", AFIPS Proceedings of the NCC, Vol. 42, pp 101-108, 1973.

[13] Thornton, J. E., Design of a Computer: The Control Data 6600, Scott, Foresman and Company, Glenview, Illinois, 1970.

| i | $\lvert\omega(T_i)\rvert$ |
|---|---|
| 1 | 21,038 |
| 2 | 20,112 |
| 3 | 20,744 |
| 4 | 20,923 |
| 5 | 19,674 |
| 6 | 17,966 |
| 7 | 20,521 |
| 8 | 19,285 |
| 9 | 18,739 |
| 10 | 17,207 |
| 11 | 15,959 |

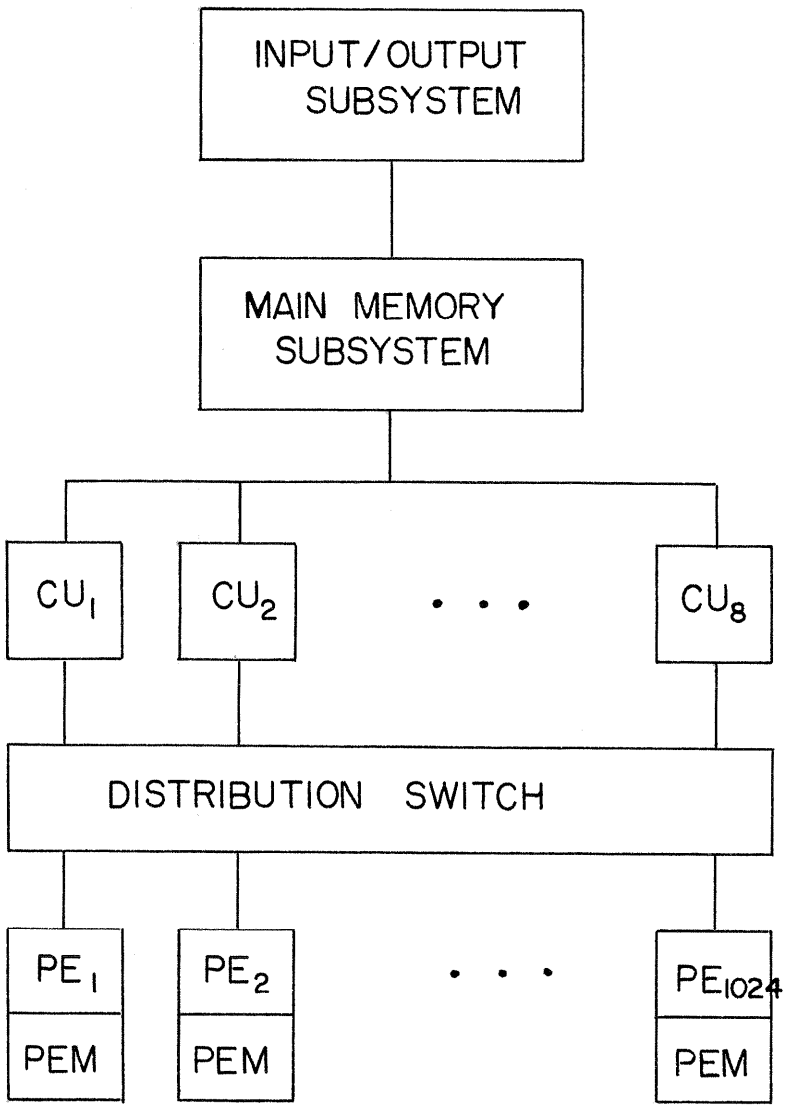
$\lvert\omega(T)\rvert$ in CU Cycles

Table 1

| | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ | $T_6$ | $T_7$ | $T_8$ | $T_9$ | $T_{10}$ | $T_{11}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $T_1$ | 0 | 1,678 | 1,100 | 903 | 2,006 | 3,624 | 1,499 | 2,597 | 3,135 | 4,327 | 5,079 |
| $T_2$ | | 0 | 1,178 | 1,633 | 1,240 | 2,748 | 1,037 | 1,631 | 2,279 | 3,431 | 4,153 |
| $T_3$ | | | 0 | 1,055 | 1,848 | 3,404 | 1,005 | 2,277 | 2,925 | 4,039 | 4,785 |
| $T_4$ | | | | 0 | 1,801 | 3,547 | 1,262 | 2,492 | 2,958 | 4,164 | 4,964 |
| $T_5$ | | | | | 0 | 2,318 | 1,567 | 1,233 | 1,757 | 2,915 | 3,715 |
| $T_6$ | | | | | | 0 | 3,161 | 1,901 | 1,325 | 1,255 | 2,007 |
| $T_7$ | | | | | | | 0 | 1,978 | 2,548 | 3,782 | 4,562 |
| $T_8$ | | | | | | | | 0 | 1,354 | 2,594 | 3,326 |
| $T_9$ | | | | | | | | | 0 | 1,970 | 2,780 |
| $T_{10}$ | | | | | | | | | | 0 | 1,248 |
| $T_{11}$ | | | | | | | | | | | 0 |

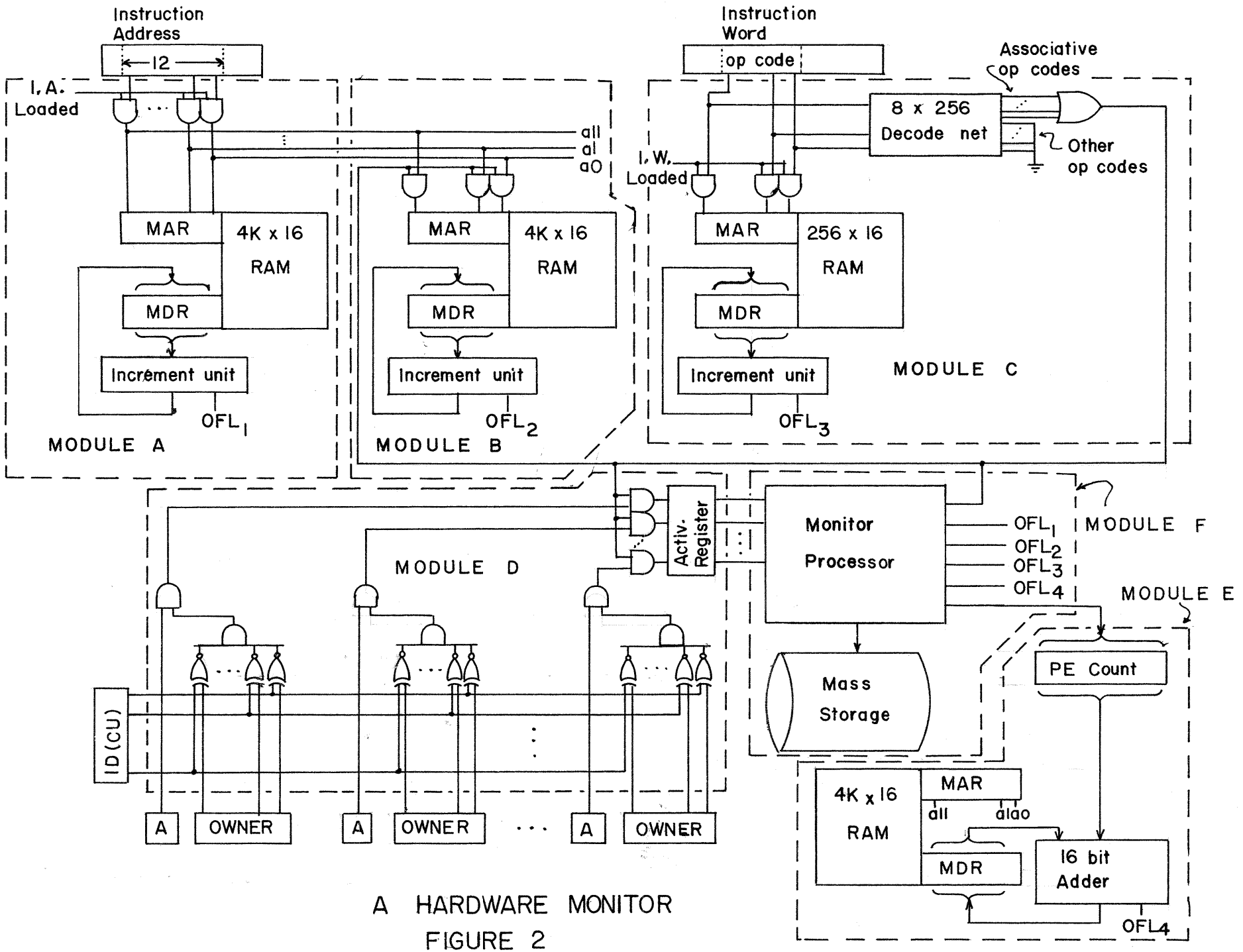
d(T,T') in CU Cyles

Table 2



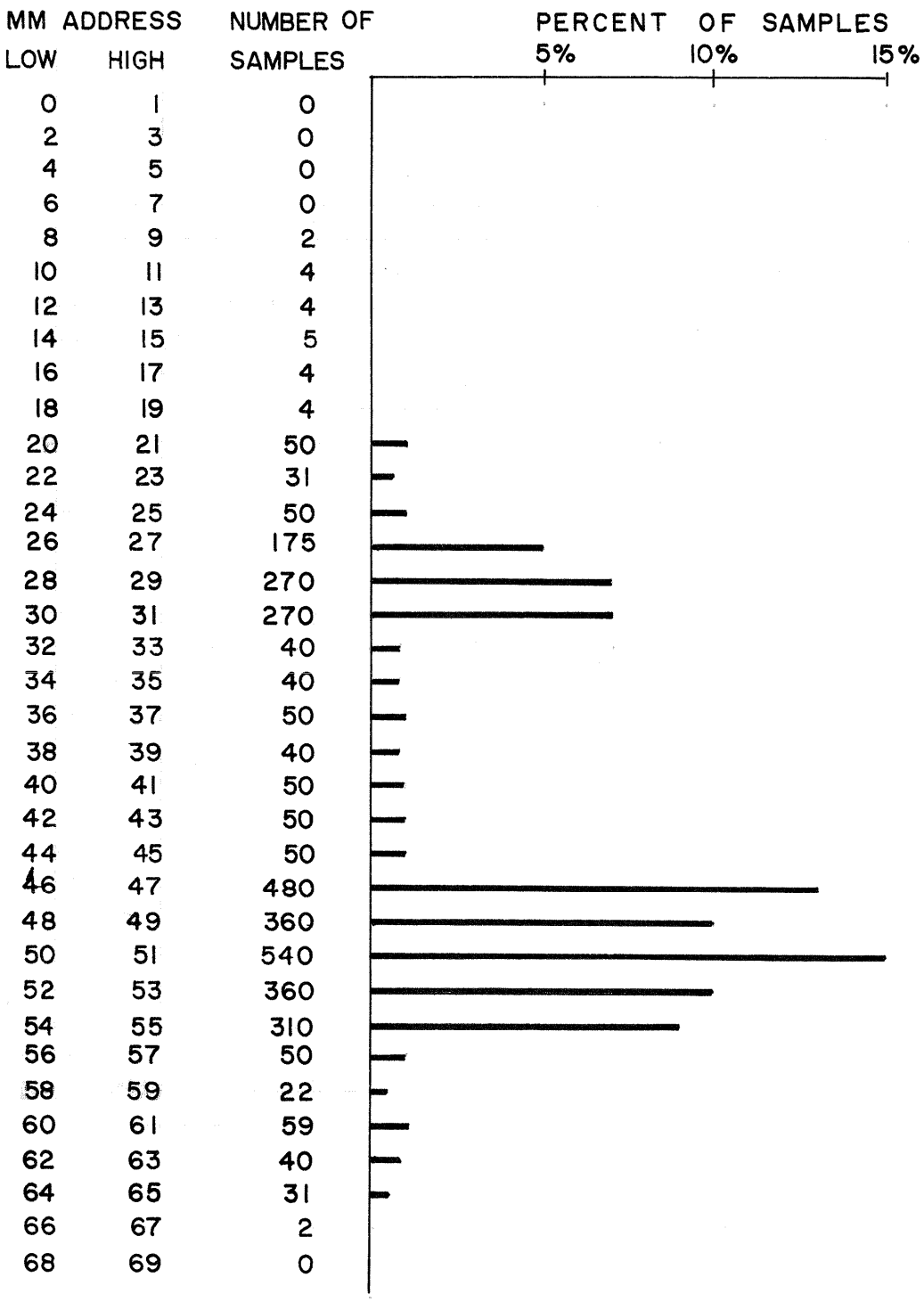

BLOCK DIAGRAM

FIGURE 1



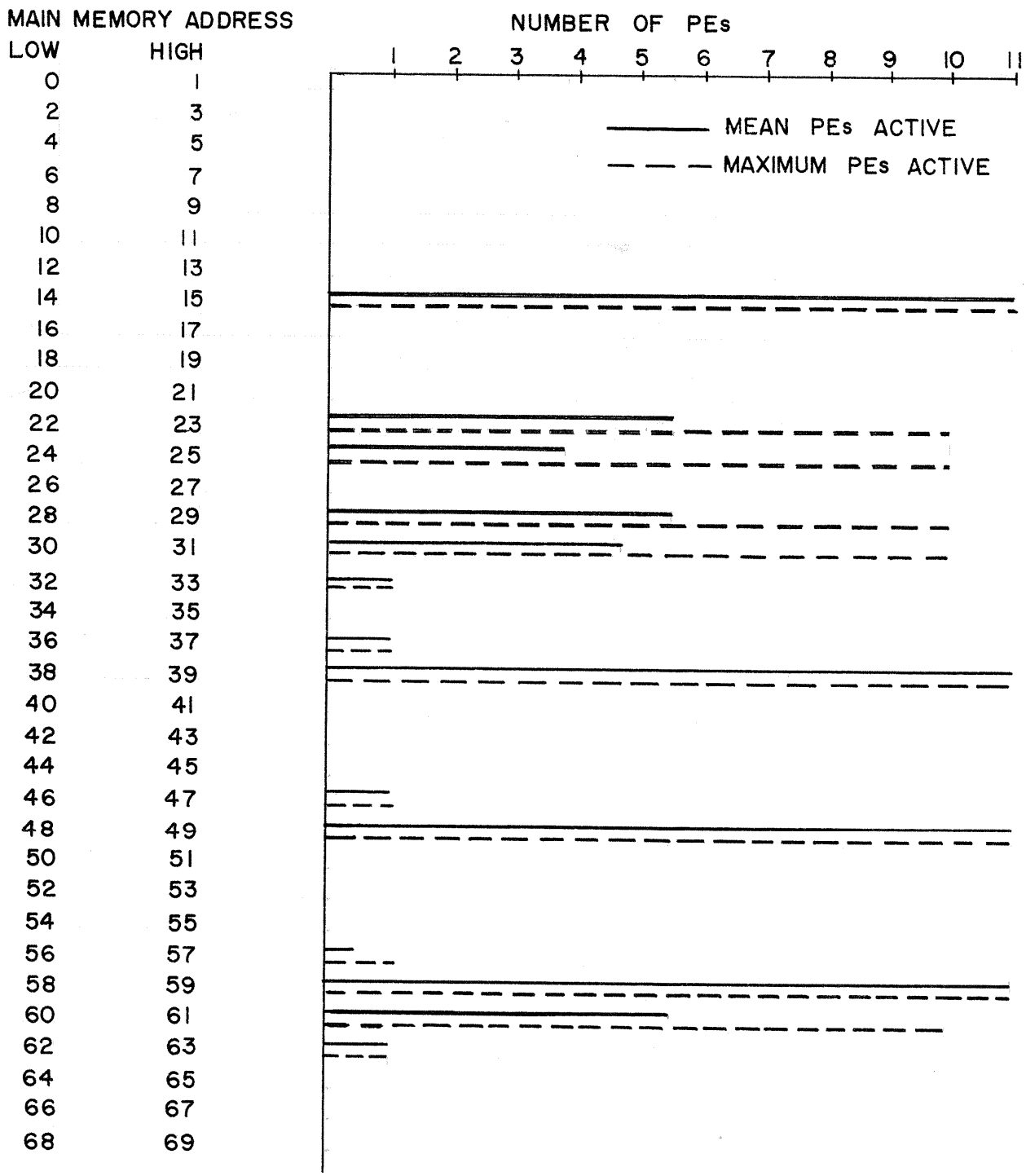

A HARDWARE MONITOR

FIGURE 2




INSTRUCTION COUNTER HISTOGRAM

FIGURE 3




PE ACTIVITY HISTOGRAM

FIGURE 4