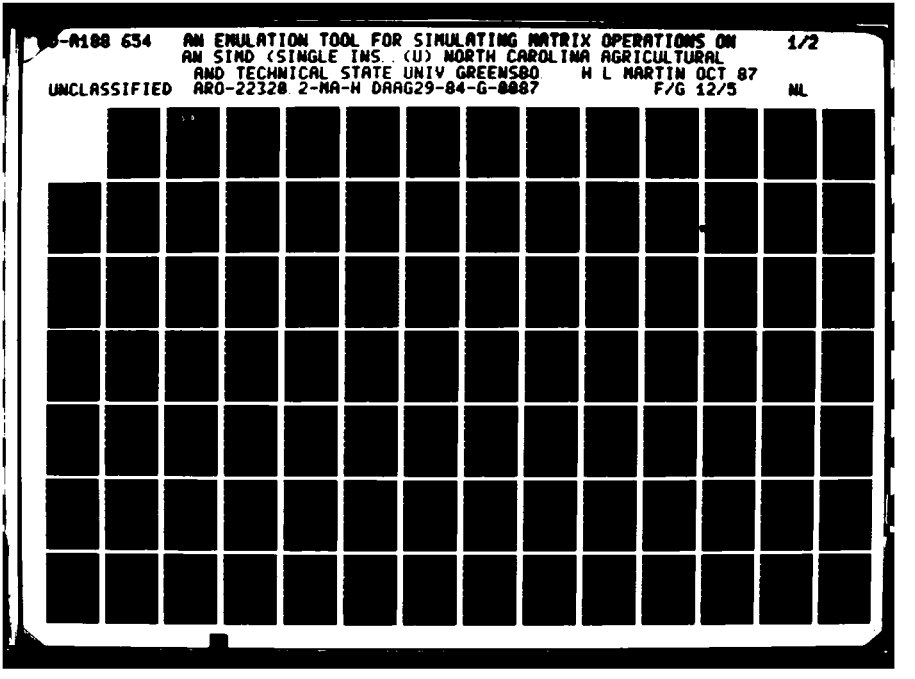
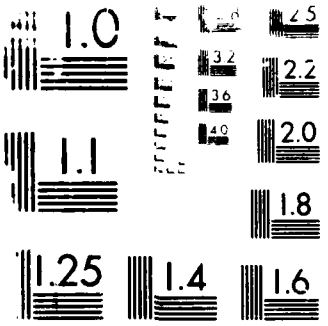

D-A188 654 AN EMULATION TOOL FOR SIMULATING MATRIX OPERATIONS ON AN SIMD (SINGLE INS... (U) NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIV GREENSBO W L MARTIN OCT 87
1/2
UNCLASSIFIED ARO-22328.2-MA-H DAAG29-84-G-0087 F/G 12/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
3.2
3.6
4.0

UNC
SECURIT

K COPY - FOR REPRODUCTION PURPOSES

(2)

AD-A188 654

DTIC
ELECTE
DEC 1 1 1987
S D

DTIC FILE COPY

DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S): ARO 22320.2-MA-H
6a. NAME OF PERFORMING ORGANIZATION: N.C. Agricultural & Tech. State Univ. / 6b OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code): Greensboro, NC 27411	7b. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING / SPONSORING ORGANIZATION: U. S. Army Research Office / 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: DAAG29-84-H-0087
8c. ADDRESS (City, State, and ZIP Code): P. O. Box 12211, Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

An Emulation Tool for Simulating Matrix Operations on an SIMD Multiprocessor

12. PERSONAL AUTHOR(S)

Harold L. Martin

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 8/15.84 TO 8.14/87	Oct 87	35

16. SUPPLEMENTARY NOTATION

The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Software, Algorithms, Emulators, System Architectures, Code Generators

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The MJH1 is a very good software tool to use in the analysis of the correctness of various partitioning algorithms. It has the potential to be a forerunner in the discipline of Computer Engineering at North Carolina A & T State University. This is possible because the emulator is easily adaptable to fit system architectures other than SIMD. Through

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

87 12 9 097

20. ABSTRACT CONTINUED

simulation, different architectures can be implemented, as well as various algorithms for exclusive partitioning of matrices to perform simple matrix operations in a parallel or multiprocessor environment.

The MJH1 Emulator has the potential to be expanded tremendously. The instruction set could grow immensely. Also, the emulator could be made into a smarter, more intelligent machine by combining some existing commands and making its language of a higher level.

ARO 22320.2-MA-H

FINAL REPORT
U. S. ARMY RESEARCH OFFICE
Grant No: DAAG29-84-G-0087

An Emulation Tool For Simulating Matrix Operations On An SIMD Multiprocessor

Dr. Harold L. Martin
Department of Electrical Engineering
N. C. A and T State University
Greensboro, N.C. 27411

FINAL REPORT
U. S. ARMY RESEARCH OFFICE
Grant No: DAAG29-84-G-0087

An Emulation Tool For Simulating Matrix Operations On An SIMD Multiprocessor

Dr. Harold L. Martin
Department of Electrical Engineering
N. C. A and T State University
Greensboro, N.C. 27411

CONTENTS

List of Illustrations

List of Tables

CHAPTER ONE

OVERVIEW OF REPORT

1.0 INTRODUCTION

By exploiting the parallel nature of matrix computations, a system configuration has been proposed for performing the basic matrix computations of matrix addition, subtraction, scalar multiplication, matrix multiplication, and matrix inversion. Algorithms have been developed for partitioning the matrix structure on the basis of this inherent parallelism. A single instruction multiple data stream machine, the Martin-Jones-Hughes Version One (the MJH1) has been developed for the efficient allocation of computational tasks in a parallel processing environment as prescribed by specific matrix operation partitioning algorithms.

The concept of processing data in parallel arose from the need to decrease computation time without decreasing the amount of computed data [1]. Novel architectures have been developed, i.e., Staran, Cray-1, etc., in order to capitalize on the concept. These architectures facilitate the strategy of distributing data evenly to a pre-determined number of processing elements (PE's) operating concurrently (in parallel) as opposed to sequentially (one after the other) and then routing all of the intermediate results to a specified location for further PE

distribution, if required. This method of computing data is both efficient and effective.

This idea permeates the concept of the parallel machine realized in this report. The MJH1 is a multiprocessor that is capable of partitioning matrices for specified matrix operations. After successfully partitioning and allocating the matrices, the computation can be performed in a minimum amount of time.

Traditionally, the von Neumann computer has been the model for all computers. However, due to rapid innovations in technology, processing speed has become a factor. Since the von Neumann machine is a sequential control flow machine, computations must be done in series, even though they may be done quicker by dividing the overall job among various sections of a machine's memory. Many novel parallel architectures have been invented because of the sluggishness of the von Neumann-like machine.

The concept of parallelism, or concurrency, yields efficiency. In circuit theory, circuits can be solved by defining currents via mesh or loop equations. The unknown currents may be solved by using Cramer's Rule, an implementation of the matrix structure. The loop method is a very general method for defining specific unknowns. It follows that loop equations can be implemented to describe parallelism. Loop equations can be used to solve matrices. Hence, matrices exhibit some inherent parallelism. Because this is true, matrix operations are easily adaptable to parallel architectures in

order to perform matrix computations. Because of the inherent parallelism of the solution techniques, matrices can be split up or partitioned among several processors, operations can be performed, and then their results written back to some central location.

1.1 REPORT OBJECTIVE

The objective of this research was to develop an emulator tool for verifying the matrix partitioning algorithms developed by Sadhasivan [2]. The emulator that performs these operations is the MJH1 [3].

1.2 REPORT ORGANIZATION

An analysis of the parallelism exhibited in the matrix multiplication operation is presented in this report. Algorithms developed for efficient partitioning of the matrix structure and the allocation of parallel computational tasks in a multiprocessing environment will be simulated. Chapter Two discusses various system architectures and their relative interconnection networks. Chapter Three defines the partitioning algorithms used to perform the various matrix computations. Chapter Four expounds on the specifics of the multiple processor architecture, including the control processing unit, the arithmetic processors (PE's) and the I/O capabilities. Also, the emulator itself will be discussed. Finally, Chapter Five includes a summation of the work done to date, as well as suggestions for future work.

CHAPTER TWO

SYSTEM ARCHITECTURE

2.0 INTRODUCTION

The idea of connecting N computers together to form one big super computer is the basic idea behind parallel processing [4]. If realized, the throughput potential of N computers can be N times that of one single computer. Thus, processing data in parallel yields great "computing power," which can approach a linear gain in overall computation speed. Thus, the idea of dividing and conquering yields faster throughput [5,6]. A basic hardware model of concurrent or parallel processing systems follows.

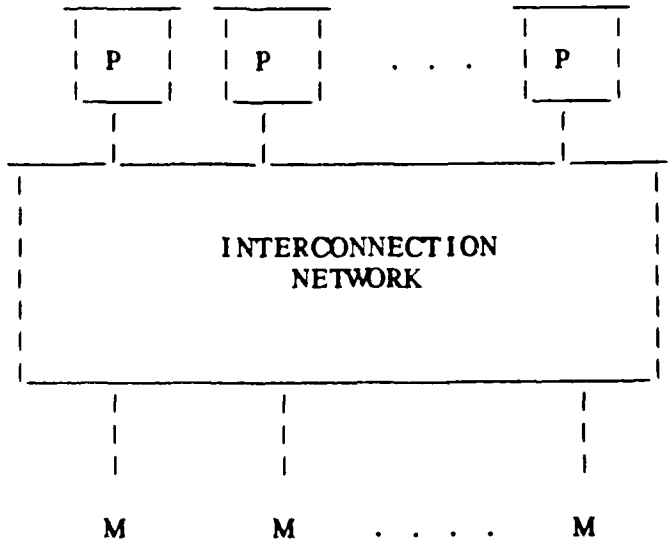


Figure 2.1

HARDWARE MODEL OF CONCURRENT PARALLEL SYSTEMS

Parallel computer architectures can be grouped into four distinct classes [7,8]. These four classes are distinguished by the parallelism afforded by the instructions coupled with the parallelism afforded by the data. Instructions or instruction streams can be distinguished in the same manner. That is, data streams can either be singular or multiple.

The classic von Neumann machine is classified as a Single Instruction Stream Single Data Stream (SISD) machine. A block diagram of the SISD architecture is pictured in Figure 2.2. This sequential machine can only execute one instruction on one piece of data at one time. Figure 2.3 shows the Array Processor or the Single Instruction Stream Multiple Data Stream (SIMD) machine which executes one instruction which acts on many pieces of data simultaneously. Usually, these data are located in arithmetic processors (APs) or in processing elements (PE's). These processors that contain the data are controlled by some type of control processor. This architecture class is easily adaptable to performing computations that can be broken into a series of vector operations, i.e. matrix operations. Perhaps the rarest configuration in practice is the Multiple Instruction Stream Single Data Stream (MISD) machine as shown in Figure 2.4. This machine exhibits parallelism in the instruction stream only and allows several instructions to be executed on the same datum. The fourth and final class of parallel computer architectures is the Multiple Instruction Stream Multiple Data Stream (MIMD) machine. This configuration allows for parallelism in both streams, instruction and data. A block diagram of MIMD

architecture is in Figure 2.5.

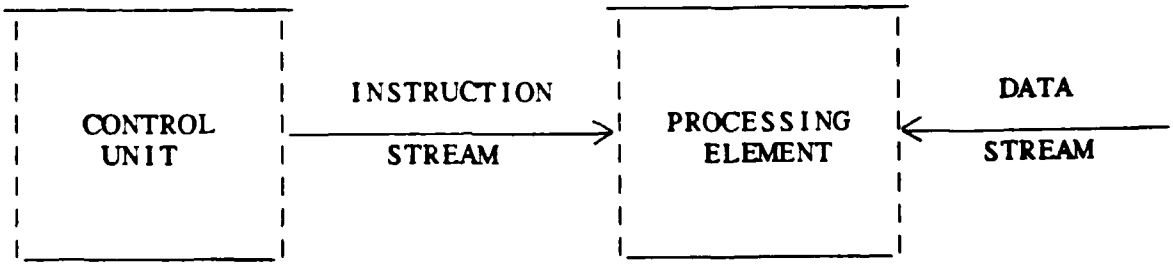
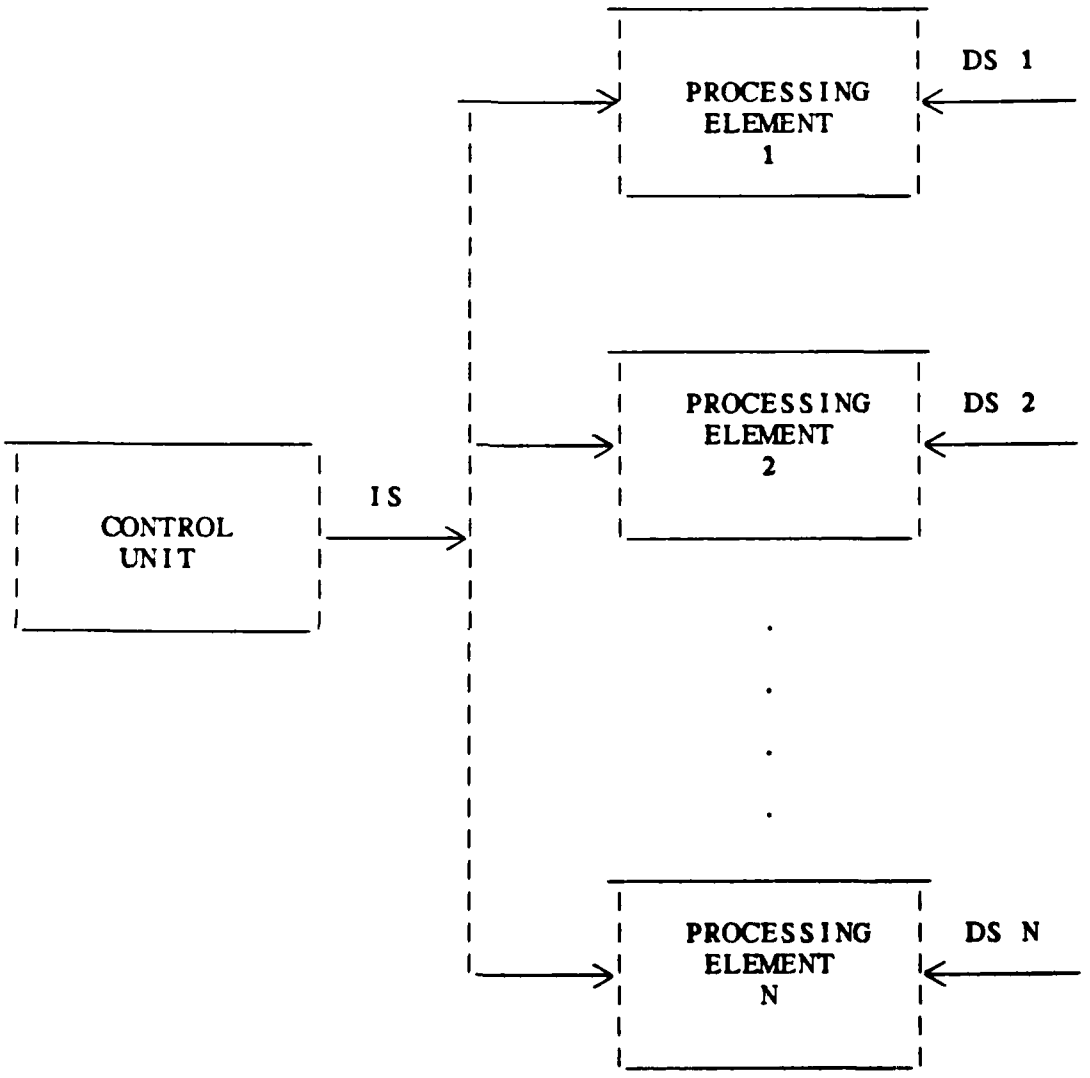


Figure 2.2

SISD BLOCK DIAGRAM



IS = INSTRUCTION STREAM
DS = DATA STREAM

Figure 2.3

SIMD BLOCK DIAGRAM

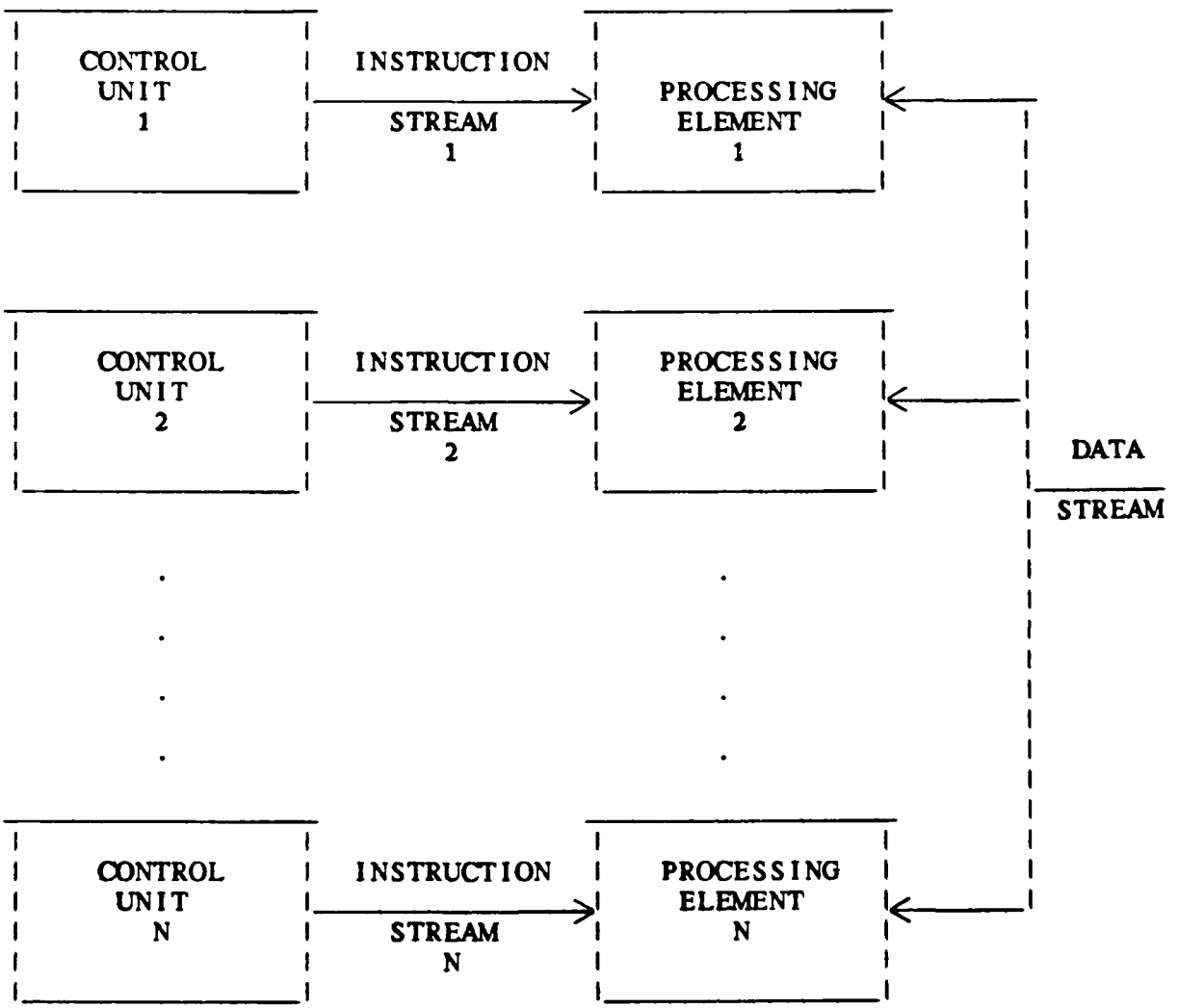


Figure 2.4

MISD BLOCK DIAGRAM

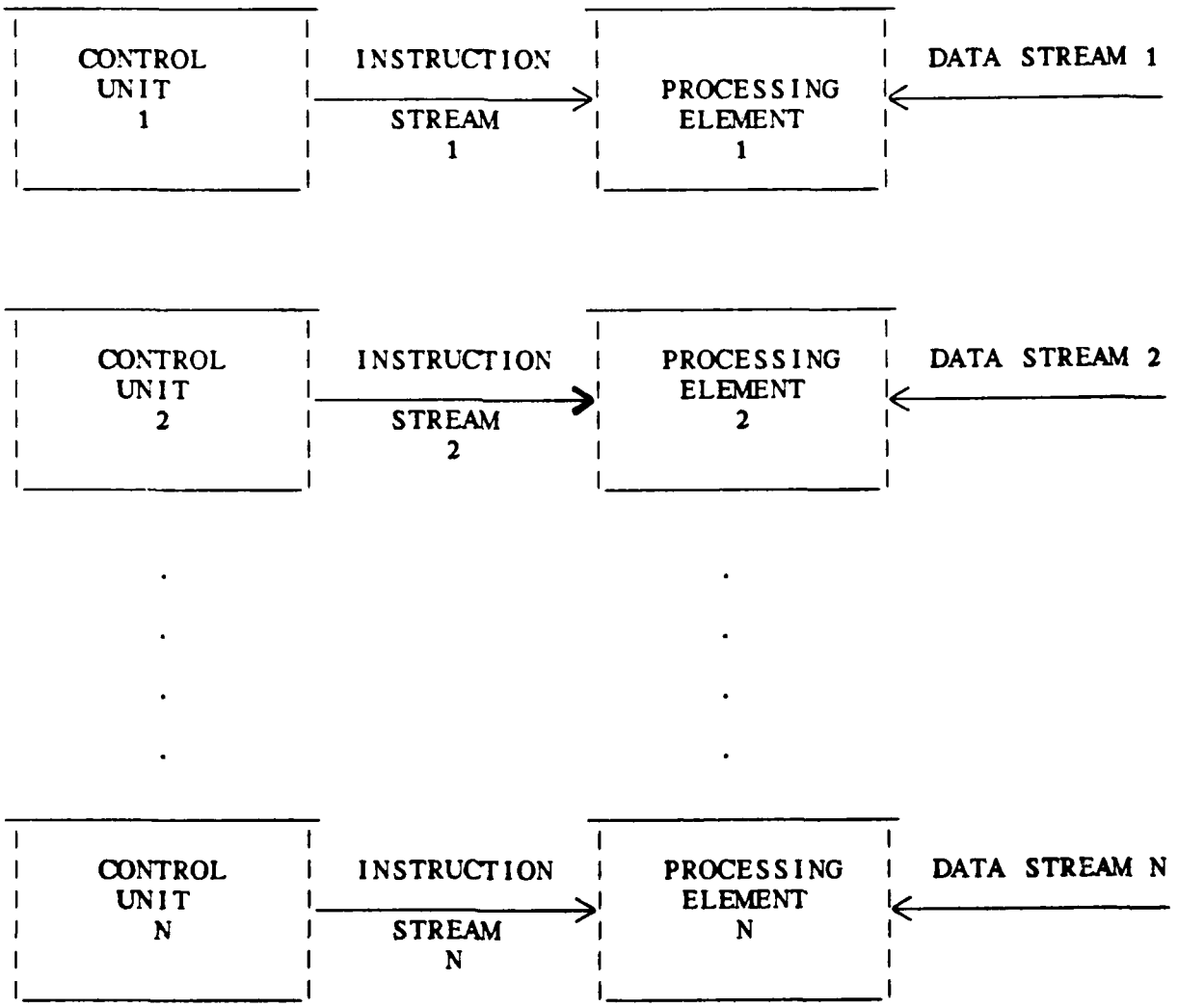


Figure 2.5

MIMD BLOCK DIAGRAM

Of the four classes of architectures, two can be used to realize parallelism with respect to data manipulations. These two are SIMD and MIMD machines.

In the class of SIMD machines, we distinguish between vector and array computers. This distinction is based primarily on the way data are communicated to elements of the system. Processors in array computers typically access data from their own memories and those of their nearest neighbors (through some type of shift network). Vector computation has been supported through pipelining or streaming and by synchronous multiprocessing. Figure 2.6 summarizes this classification of parallel computers.

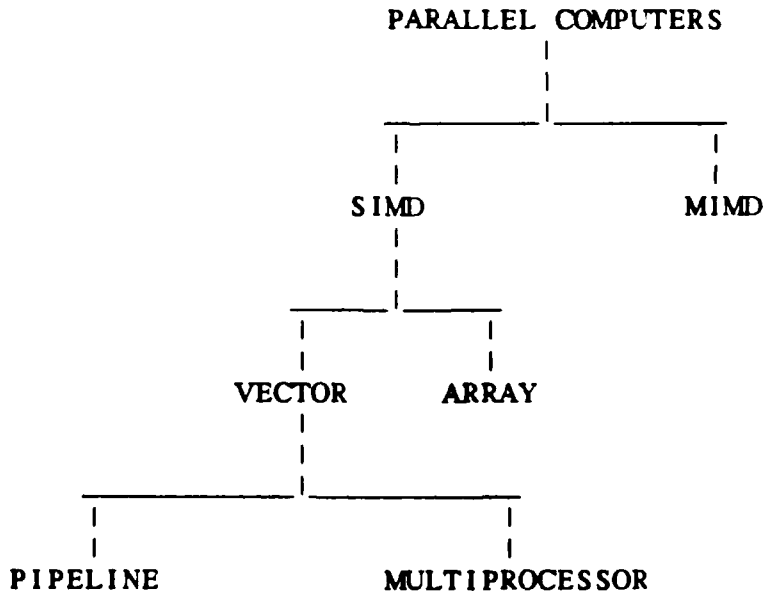


FIGURE 2.6

PARALLEL COMPUTERS

2.1 SIMD ARCHITECTURE

The Array Processor, Single Execution Array, or SIMD array is a machine that executes one instruction per array of data. Distinguishing architectural features include: a control unit that houses a local memory and can broadcast one single instruction to all processing elements (arithmetic processors); a predefined number of processing elements; processor memories (one per processing element); a communication system among processors and external data sources.

The main memory contains the combined memory of the N processor elements (PE's) and all of the instructions are stored there. These are connected via a high speed data bus with a bandwidth N times that of the individual memories and compatible with both the processor and the I/O bandwidths.

The number of available arithmetic processors and the number of computations required per processor for the complete execution of the computation, determine the instruction stream broadcast to the arithmetic processors by the control processor. In other words, the system configuration determines the broadcasted stream of information to the enabled parocessors. The processor must know prior to runtime how the system is configured.

The instructions are fetched from the memory in blocks into a buffer. These instructions can be of two types: control instructions executed by the control unit, or vector instructions executed by the arithmetic processor. A listing of the instructions for the MJH1 is included in a subsequent chapter.

Instructions of two different kinds leads to processing idling which results in the inefficient use of processing times. Processing idling can result when the controller executes a scalar instruction, and the arithmetic processors are idle waiting for the next operation. Alternatively, when computations require more than one cycle of parallel processor operation, all the processors may not be needed for the last cycle of operation. This results in idle processors. Although processor idling can occur, program execution and data manipulation in parallel is by far faster than the sequential execution and manipulation of data.

This problem can be alleviated, however, by buffering the instruction stream. That is, fetching several instructions at once and then distributing them evenly and efficiently; instruction stream pipelining via the control processor; and overlapping the instruction fetch sequence with data manipulations not requiring main memory usage.

Since there is only one control processor, the processing elements must be synchronized by that single processor. There is only one clock per machine, one timing signal or clock controls everything. This makes conditional branching to route data past certain idle processors virtually impossible.

PE routing is achieved, instead, via the control processor testing, setting, or resetting the mask bits associated with each. Various PE's only receive the broadcasted instruction or data if they are enabled, i.e. if their mask bits are high.

Mask bits are set at the completion of an instruction fetch cycle, prior to the execution of the next instruction. Mask bits can only be set or reset by the control processor. However, this mask checking slightly degrades the parallelism of the SIMD machine.

2.2 INTERCONNECTION NETWORKS

Access and interprocessor communications are major problems in the programming and design of SIMD machines. A number of different networks have been proposed to achieve fast, efficient communications at a reasonable cost. Some of these interconnection networks are included in subsequent sections [9].

2.2.1 CYCLIC-SHIFT NETWORK

Also called the exchange network or the uniform shift alignment network, the cyclic-shift interconnection network involves several processors connected cyclically to facilitate bidirectional data transfer between adjacent processors in a unit shift time of one cycle. A cyclic shift is achieved by a sequence of cyclic shifts of a unit amount. Between processors i and j there are ((i-j-1) mod p) processors through which the data must be shifted while moving from processor i to processor j.

The main advantage of the cyclic interconnection is the constant number of connections it supports per processor in an N processor system. Every processor is connected to its adjacent neighbor. However, the network is very inefficient for algorithms requiring noncyclic data permutations and may

sometimes require N mask-and-shifts involving N alignment cycles to route data to the appropriate processor for further manipulation. An illustration of this interconnection follows in Figure 2.7.

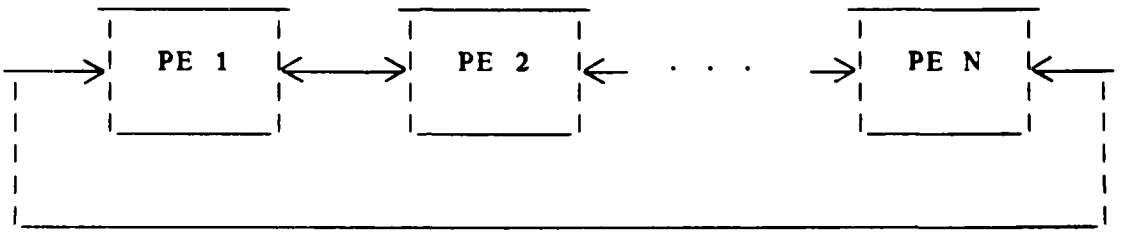


FIGURE 2.7

CYCLIC-SHIFT NETWORK

2.2.2 CROSSBAR NETWORK

This interconnection network in its simplest form can be positioned between the source units and destination units to implement the data movement in one transfer instruction. This network, known for its simplicity, is an N by N array of switches, with the N source units connected to the rows and the N destination units connected to the columns. The network is bidirectional. Each processor and memory has an input and output connection to the network and simultaneous conflict-free connections from any source to any destination for a one-to-one mapping.

A major disadvantage of the crossbar network is that its size grows by N squared with the subsequent addition of PE's. This increase is quadratic in nature and is not compatible to the typical linear increase found in parallel processing. Thus, the crosbar network is best suited for systems that confiure a small number of source and destination units. A block diagram of the crossbar network follows. This configuration consists of four processing elements.

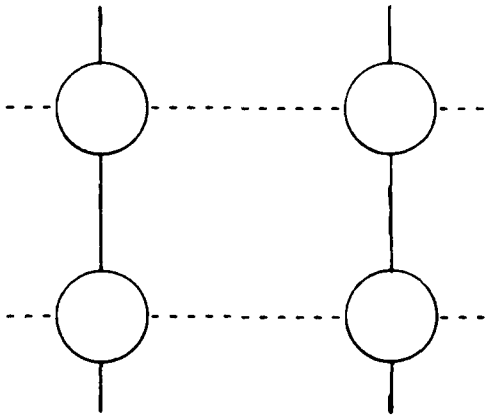


FIGURE 2.8
CROSSBAR NETWORK

2.2.3 PERFECT SHUFFLE NETWORK

This network derives its name from the shuffling of a deck of cards. The basic idea involves splitting N processors into equal halves and interlacing them. The ith processor of the unshuffled deck is bidirectionally connected to the ith processor of the shuffled deck. The perfect shuffle connection pattern routes data from position P to position P'.

The above mentioned interconnection networks are typical in many SIMD machines. However, processors may be dedicated to highly specialized algorithms, and their interconnections may be designed specifically for the implementation of these algorithms, and may be different from the ones mentioned in this thesis. SIMD machines tend to exhibit more speed of computations. This is primarily due to the fact that SIMD machines do not require as much synchronization, task scheduling, and system software as does the MIMD machine. Also, high reliability of the SIMD configurations can be attributed to the redundancy of the data manipulations.

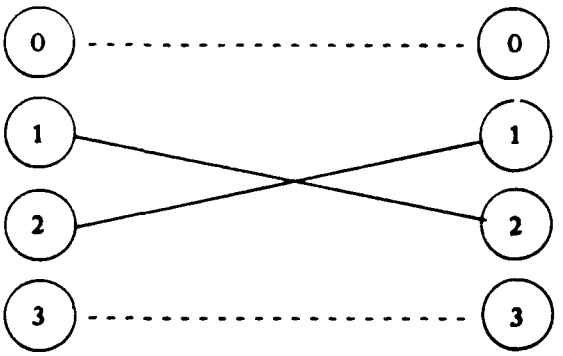


FIGURE 2.9

PERFECT SHUFFLE NETWORK

2.3 MIMD MACHINES

As stated before, MIMD machines are computers with one or more general purpose processor, each capable of executing a separate stream of instruction on a stream of data that is housed in central memory. In essence, the advantage of the MIMD machine is its capability to share memory, I/O units, and several computing units. Because MIMD architecture is centered around the concept of shared resources, MIMD machines are broadly classified into two categories: tightly coupled systems, and loosely coupled systems.

Tightly coupled systems consist of several centrally located processors that share the same memory and data structures and are supervised by a single operating system. Loosely coupled systems are comprised of independent computer systems that are physically distributed at several locations and supervised by a distributed operating system. Each processor has its own system software and data structures, and shares a common data base with the other systems via slow speed communication lines.

There are two methods of managing multiprocessor systems. One approach involves a hierarchical organization with functional division amongst the processors and supervisor controlling the specialized functional units. An alternate approach involves a network of independent computing systems communicating with each other on an equal basis. In both methods, the individual processors execute the individual instruction streams assigned to them sequentially.

Since the MIMD system is more flexible than the SIMD machine, it is more suitable for a large class of computations. However, flexibility comes with a price. That price is synchronization and allocation problems. The partitioning of the physical problem into several processes that can be executed in parallel is of major concern in the MIMD machine. However, MIMD systems are configured for ease, overall reliability, with functional specialization for overall throughput. Despite this, the problems of resource allocation, synchronization, and problem partitioning will not allow the realization of MIMD computers with several processors. Coupling refers to the ability of the various processors to share resources. Figure 2.10 shows a breakdown of the MIMD class of parallel computer.

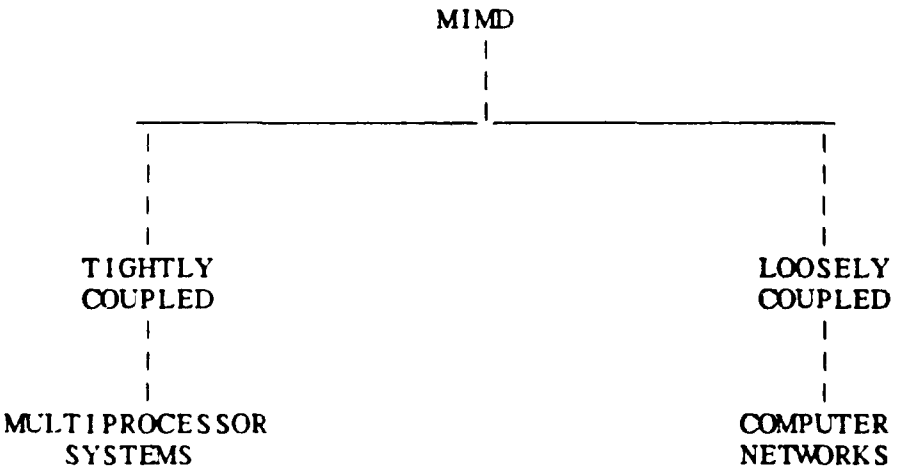


FIGURE 2.10

MIMD MULTIPLE PROCESSOR ORGANIZATION

2.4 SYSTEM ARCHITECTURE FOR MATRIX COMPUTATIONS

The system realized via this thesis is designed specifically for the solution of various matrix computations. The SIMD configuration was chosen to satisfy the special purpose requirements for the matrix operations of solution of simultaneous equations and matrix transpose. All these operations can be done in parallel.

The system architecture configures a system of K arithmetic processors, each equipped with local memories interconnected in a bidirectional cyclic-shift fashion. Each processor is individually connected to the control processor, and to central memory. Communication to the controller is achieved via handshaking signals. To avoid bus contention, the data required for the computations are assigned to each processor's local memory during the initial transfer of data, prior to runtime. The controller also broadcasts the intended operation to be performed along with the data values. What is not broadcast, is the operand addresses since the processors execute the specified operations on the data that is stored from a specific starting address in their local memories. To further avoid bus contention, the controller monitors the state of each arithmetic processor and keeps track of the number of computations tp be performed by each processor. The controller can test and set mask bits in order to shut out idle or unnecessary processors.

Interconnection between arithmetic processors involving data transfer is as follows. Data to be transferred is put onto the shift port. Then, through unit cyclic shifts, the data is passed from processor to processor until an enabled (or strobed) processor receives the data from the port and puts it into its shift memory. The controller strobes the appropriate destination processor. Note that the controller only has access to the computed results at the end of all task executions.

The allocation algorithms reside in the main memory of the controller. The algorithm is system based, so once the number of arithmetic processors in the configuration has been defined, the data allocation and data transfer operations are expressed as only a function of the input matrix dimensions.

The system architecture discussed above is implemented in software by using Pascal. The data allocation and control algorithms and the specific details of the system hardware and related software, implementing the synchronization, execution and coordination of the various matrix computations in parallel, are explained in detail in the chapter that expounds on the MJH1.

In conclusion, SIMD and MIMD systems are the most prevalent architectures among parallel machines today. It has been noted that SIMD systems are well suited for matrix computations due to the inherent parallelism found in matrices. This is true because there is not a synchronization problem like the one found in MIMD architectures, since all PE's are controlled by the processor.

However, in SIMD systems, the system software ignores the parallel architecture and reveals it to the user, allowing maximum benefit without any system software overhead. This in turn passes the burden of programming required to exploit the inherent parallelism to the user.

CHAPTER THREE

PARTITIONING ALGORITHMS FOR MATRIX COMPUTATIONS

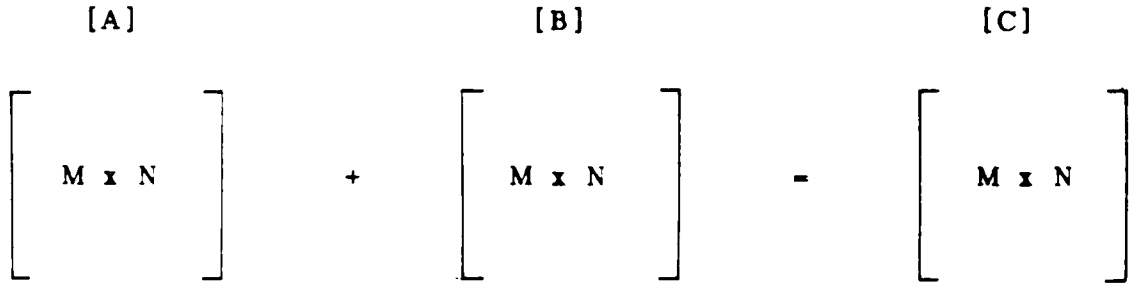
3.0 INTRODUCTION

In this chapter, the actual algorithms for partitioning the matrix structure for basic matrix computations are presented. More specifically, the partitioning algorithms for matrix addition/subtraction and scalar multiplication (each share identical algorithms), matrix multiplication, and matrix inversion are stated as allocation theorems. These theorems were developed and proven by Sadhasivan. For a more indepth study of the actual algorithms, including theorems, corollaries, and their proofs, the author refers the reader of this document to the thesis written by Sadhasivan [2].

The allocation theorems for each basic matrix computation will merely be stated in this chapter for clarification purposes, as well as for completeness. They are the basis for the proposal of performing matrix computations on an emulated microprocessor-based SIMD multiprocessor.

3.1 PARTITIONING ALGORITHMS FOR MATRIX ADDITION/SUBTRACTION AND SCALAR MULTIPLICATION

Let the matrices involved in these computations be,



Note that m is the number of rows and n is the number of columns of [A], [B], and [C]. Assuming the number of arithmetic processors to be k, the partitioning algorithms can be defined for the different occurances of parallelism in the matrix structure and its operations as shown below.

1.a. If m=k, then the allocation of [A], [B], matrix values are done according to the following theorem:

Theorem: One row of [A] and the corresponding row of [B] are allocated to each processor.

1.b. If m is a multiple of k, then the allocation of matrix values is done according to the following theorem.

Theorem: (m/k) rows of [A] in order, and corresponding rows of [B] are allocated to each processor.

2.a. If n=k, then the allocation of matrix values is done according to the following theorem.

Theorem: One column of [A] and the corresponding column of [B] are allocated to each processor.

2.b. If n is a multiple of k, then the allocation of matrix values is done according to the following theorem.

Theorem: (n/k) columns of [A] in order, and the corresponding

columns of [B] are allocated to each processor.

3. If t=(mn) and if t>0 then,

a. if t=k or t is a submultiple of k and,

i. if m<=n, then the allocation of the input matrix values is done according to the following theorem.

Theorem: For i=1 to m

{ for j=1 to n

{(ni+j-n)th processor is allocated the values A[i,j] and B[i,j], to compute the result matrix value C[i,j].

} /* for j */

} /* for i */

ii. If m>n, then the allocation is done according to the following theorem.

Theorem: For j=1 to n

{ for i=1 to m

{(mj+i-m)th processor is allocated the values of A[i,j] and B[i,j] to compute the result matrix value C[i,j].

} /* for i */

} /* for j */

3.b. If t is a multiple of k, such that t=jk and,

i. if m<=n i.e. m is a submultiple of k such that k=qm, then the allocation of [A] and [B] values is done column wise according to the following algorithm.

Theorem: For i=1 to q

{ for r=1 to m

{for c=0 to (j-1)

```
                { ((i-1)m+r)th processor is allocated the
                  A[r,(cq+i)] and B[r,(cq+i)] values to
                  compute the result matrix value C[r,(cq+i)]
                } /* for c */
              } /* for r */
            } /* for i */
```

ii. If m>n, i.e. n is a submultiple of k such that k=qn (m=jq), then the allocation of [A] and [B] values are done row wise according to the following theorem.

```
Theorem: For i=1 to q
          { for c=1 to n
            { for r = 0 to (j-1)
              { ((i-1)n+c)th processor is allocated the
                A[(rq+i),c] and B[(rq+i),c] values to compute
                the result matrix value C[(rq+i),c]
              } /* for r */
            } /* for c */
          } /* for i */
```

4. If t is not related to k, then letting j=⌊t/k⌋ and,

a. if m is a submultiple of k such that k=mq and letting i=jq (also note that if t>k then n>q), the allocation of [A] and [B] values are done according to the following theorem.

```
Theorem:  For l=1 to (n-1)
           { for r=1 to m
             { ((l-1)m+r)th processor is allocated the A[r,l] and
               B[r,l] values and,
               if t>k then
               For c=1 to j
```

```
                {A[r,(cq+1)] and B[r,(cq+1)] values are also
                 allocated to the same processor
            } /* for c */
 to compute the corresponding values of the result matrix [C]
            } /* for r */
            } /* for l */
             if t>k then
             for l=(n-i+1) to q
              { for r=1 to m
                { for c=0 to (j-1)
                  { ((l-1)m+r)th processor is allocated the
                    A[r,(cq+1)] and B[r,(cq+1)] values to
                    compute the result matrix value C[r,cq+1)]
                   } /* for c*/
                   } /* for r */
                   } /* for l */
```

4.b. If n is a submultiple of k such that k=qn, and letting i=jq (if t>k, m>q), the allocation of [A] and [B] matrix values are done according to the following theorem.

```
Theorem: For l=1 to (m-1)
          { for c=1 to n
            { ((l-1)n+c)th processor is allocated the A[l,c] and
              B[l,c] values and,
              if t>k then,
              for r=1 to j
              { A[(rl+1),c] and B[(rq+1),c] values are also
                allocated to the same processor
```

```
} /* for r */
to compute the corresponding values of the result
matrix [C]
 } /* for c */
 } /* for l */
 if t>k then
 for l=(m-i+1) to q
 { for c=1 to n
   { for r=0 to (j-1)
     { ((l-1)n+c)th processor is allocated the
       A[(rq+1),c] and B[(rq+1),c]
     } /* for r */
     } /* for c */
     } /* for l */
```

5. If the total number of elements of [C] to be computed, t = mn, is a totally random value, then letting q=⌊t/k⌋ and r=qk, the partitioning and allocation of the input matrix values can be done according to the following theorem.

1. The positions of [A] and [B] matrix values are first individually modified to represent them as linear lists such that, the value A[i,j] or B[i,j] occupies the (ni+j-n)th position in the linked lists.
2. For p=1 to (t-r)

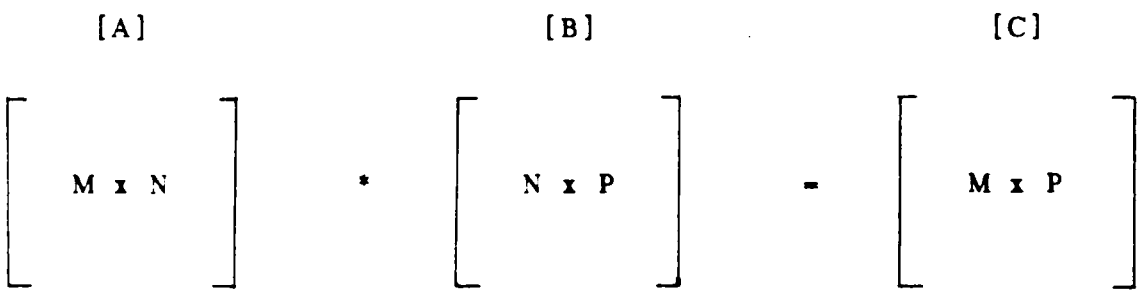
```
{ processor p is allocated the pth values of [A] and [B]
  from their linear lists and,
   if t>k then
   for i=1 to q
```

```
    { (ik+p)th values in the linked lists of [A] and [B] are
      also allocated to it
    } /* for i*/
  to compute the corresponding values of the result matrix
  [C]
    } /* for p */
     if t>k then,
     for p=(t-r+1) to k
     { for i=0 to (q-1)
       { pth processor is allocated the (ik+p)th values of
         [A] and [B] from their linear lists to compute the
         corresponding values of the [C] linear list.
       } /* for i */
     } /* for p */
```

The partitioning algorithms for scalar multiplication of an input matrix are the same as those suggested for matrix addition/subtraction except that now instead of two input matrices, only one input matrix and the scalar value are involved in the allocation. The computation involved in this case is obviously the multiplication operation instead of addition or subtraction and the computation time of the result is consequently expressed in terms of multiplication time for the evaluation of an element of the result matrice instead of addition/subtraction time required to compute the same element.

3.2 PARTITIONING ALGORITHMS FOR MATRIX MULTIPLICATION

For the discussion that follows, let the matrices involved in the computation be defined as shown below,



Note that m is the number of rows of [A] and [C], n is the number of columns of [A] and rows of [B], and p is the number of columns of [B] and [C]. If intercommunication between arithmetic processors is to be avoided during the matrix multiplication operation, then all the values of a specific row of [A] and all the values of an appropriate column of [B] required to compute a product element of the result matrix [C], should be allocated to a single processor, to compute that product. The evaluation time of each element of the product matrix [C], is referred to as the computation unit time throughout our discussion. With these assumptions and using the above mentioned representation for the matrix structure, the partitioning algorithms for multiplication can be defined.

1. If the number of values of the result matrix, t = mp, is equal to the number of arithmetic processors, k or is a submultiple of k, then the allocation of [A] and

[B] values is done as shown,

Theorem: One row of values of [A] and one column of values of [B] are allocated to each processor, according to the following theorem,

```
For r=1 to m
{ for c=1 to p
  { ((r-1)p+c)th processor is allocated the
    rth row of values of [A] and the cth
    column of values of [B] to compute the
    value C[r,c] of the result matrix
  } /* for c */
} /* for r */
```

2.a. If the number of columns of the product mattrix, p, equals k, then the allocation is done as follows,

Theorem: The entire matrix [A] and one column of values of [B] are allocated to each processor.

b. If p is equal to a multiple of k, then the allocation is done as follows,

Theorem: The entire matrix [A] and (p/k) columns of values of [B] in order, are allocated to each processor.

3.a. If the number of rows of matrix [A], m equals k, then the allocation of input matrix values is done according to the following theorem.

Theorem: One row of values of [A] and the entire matrix [B] are allocated to each processor.

b. If m is a multiple of k, then the allocation of [A] and [B] matrix values are done according to the following theorem.

Theorem: (m/k) rows of values of matrix [A] and the entire [B] matrix are allocated to each processor.

4. If t = mp, is a multiple of k, t = jk and,

a. if m<=p i.e. m is a submultiple of k, k = qm, then the allocation [A] and [B] values is done as follows.

```
Theorem: For r=1 to m
         { for c=1 to q
           { ((r-1)q+c)th processor is assigned the rth row
             values of [A] and cth string of j columns of
             [B] in order.
           } /* for c */
         } /* for r */
```

b. If m>p i.e. p is a submultiple of k such that k=rp, then the result matrix [C] is computed row wise according to the following theorem,

```
Theorem: For q=1 to p
         { for c=1 to r
           { ((q-1)r+c)th processor is assigned the
             cth string of j rows of values of [A]
             in order and qth column of values of
             [B]
           } /* for c */
         } /* for q */
```

5. If t = mp is > k, but not related to k and,

a. if the number of columns of [A], n is equal to k, then the allocation of [A] and [B] values is done according to the theorem presented next. Note that interprocessor communication for the addition

of partial products generated in each of these processors, to give a final product element of [C].

Theorem: One column of values of [A] and one row of values of [B] in order, are allocated to each processor. Each processor calculates each of the n or k partial products by using the column of values of [A] and relevant row of values of [B] stored in it. The final product is obtained by adding the n partial products by means of unit shifts between the processors followed by parallel additions of these partial products.

b. If n is a multiple of k, the allocation of [A] and [B] values is done according to the following theorem. (Note that interprocessor communication takes place through the cyclic shift interconnection provided in the system architecture).

Theorem: (n/k) columns of [A] and (n/k) rows of [B] in order are allocated to each processor.

c. If n is a submultiple of k and,

i. the number of elements in [A], (mn) is a multiple of k, then the allocation of [A] and [B] values are done according to the following theorem.

Theorem: For r=1 to n

{ for l=0 tp (k/n-1)

{ (ln+r)th processor is assigned (l+1)th, (mn/k) values of column r of [A] and

```
                the rth row of values of [B]
              } /* for l */
            } /* for r */
```

ii. if the number of elements in [B], (np) is a multiple of k, then the allocation of [A] and [B] values are done as shown below.

```
Theorem:  For c=1 to n
          { for l=0 to (k/n-1)
            { (ln+c)th processor is assigned the cth
              column values of [A] and (l+1)th string
              of (np/k) values of row c of [B]
            } /* for l */
          } /* for c */
```

d. if n is a submultiple of k, but the number of elements in [A] and in [B] are not a multiple of k but,

i. m=>p; then letting j=⌊mn/k⌋ and i=jk/n, the allocation of [A] and [B] values is done according to the following theorem.

```
Theorem:  For c=1 to n
          { for l=0 to (k/n-1)
            { (ln+c)th processor is assigned the
              (ln+1)th string of (in/k) values of
              column c of [A] and cth row of values
              of [B]
            } /* for l */
          } /* for c */
```

Also the additional values of [A] are allocated

such that,

```
for r=(i+1) to m
{ for c=1 to n
  { ((r-(i+1))n+c)th processor is
    allocated the cth column value of the
    rth row of [A]
  } /* for c */
} /* for r */
```

Each final product is obtained by shifting and adding the n partial products between the string of n processors. For synchronization purposes, the i rows of [C] are first evaluated column wise after which, the remaining (m-i) rows of [C] are computed.

ii. if m<p, then letting j=⌊np/k⌋ and i=jk/n, the allocation of [A] and [B] values is done according to the following theorem.

Theorem:

```
For c=1 to n
{ for l=0 to (k/n-1)
  { (ln+c)th processor is assigned the
    cth column of [A] and (l+1)th string
    of (in/k) values of row c of [B]
  } /* for l */
} /* for c */
```

Also the additional values of [B] are allocated such that,

```
for c=(i+1) to p
{ for r=1 to n
```

{ ((c-(i+1))n+r)th processor is allocated
the rth row value of cth column of [B]
} /* for r */
} /* for c */

Each final product is obtained by shifting and adding the n partial products between the string of n processors. For synchronication purposes, the i columns of [C] are first evaluated, followed by the eveluation of the remaining (p-i) columns of [C].

e. if m is a submultiple of k, then letting j=⌊mp/k⌋ and i=(p-jk/m), the allocation of [A] and [B] values is done according to the following theorem.

Theorem: For s=1 to i
{ sth string of m processors is allocated
m rows in order of [A] and,
for c=0 to j
{ (ck/m+s)th column of values of [B]
} /* for c */
} /* for s */

In addition, the following allocation is made,
for s=(i+1) to k/m (if s<p)
{ sth string of m processors is allocated
the m rows of [A] in order and,
for c=0 to (j-1)
{ (ck/m+s)th column of values of [B]
} /* for c */
} /* for s */

Note that interprocessor communication

is not required in this scheduling algorithm.

f. if p is a submultiple of k, then letting j=⌊mp/k⌋ and i=(m-jk/p), the allocation of [A] and [B] values can be done according to the following theorem.

Theorem: For s=1 to i

```
{ sth string of p processors is allocated
  the p columns of [B] in order, and
  for r=0 to j
  { (rk/p+s)th row of values of [A]
  } /* for r */
} /* for s */
```

In addition, the following allocation is made:

```
for s=(i+1) to k/p   (if s<m)
 { sth string of p processors is allocated p
   columns in order of [B] and,
   for r=0 to (j-1)
    { (rk/p+s)th row of values of [A]
    } /* for r */
 } /* for s */
```

Note that no interprocessor communication is required for this scheduling algorithm.

6. If the values of m, n, and p are totally arbitrary, defying any trace of parallelism, then letting t=(mp), q=⌊t/k⌋ and r=(qk), the allocation of [A] and [B] values is done by first establishing a linear relationship

between the individual values of product matrice [C] such that, the element C[i,j] occupies the position (pi+j-p)th location in the linear list. With this assumption, the allocation of [A] and [B] values among the processors can be done according to the following theorem.

```
Theorem:  For s=1 to (t-r) /* parallel sequence */
          { for l=0, and if (t>k) then, /* serial
            1 to q                        sequence */
            { sth processor computes the (lk+s)th
              value in the linked list of [C]
            } /* for l */
          } /* for s */
          If (mp)>k, then
           for s=(t-r+1) to k /* parallel sequence */
            { for l=0 to (q-1) /* serial sequence */
              { sth processor computes (lk+s)th
                value in the linked list of [C]
              } /* for l */
            } /* for s */
```

3.3 SUMMARY

The preceding theorems are the formulae for partitioning matrices efficiently for the speedy performance of the basic operations of addition, subtraction, and scalar and matrix multiplication. There are some algorithms developed for matrix inversion, also, but they are not encompassed within the scope of

this report. In fact, this report deals explicitly with verifying the partitioning algorithms for matrix multiplication. However, the emulator can presently handle matrix addition, subtraction, and scalar multiplication. For that reason, they were included in this section. Martin and Sadhasivan [10,11,12] have researched other relevant issues that are aroused by these algorithms. Their findings have been published, and may be referred to in order to enhance one's knowledge or this subject.

It is hoped that the reader will be able to follow the aforementioned theorems, coupled with the instructions for programming the MJH1 to successfully partition and execute a basic matrix computation via the MJH1 emulator.

CHAPTER FOUR

THE MARTIN-JONES-HUGHES VERSION ONE

4.0 INTRODUCTION

The Martin-Jones-Hughes Version One, as previously stated, is an emulated multiprocessor designed specifically to perform matrix computations. A software description of the MJH1 is written in the structured programming language of Pascal. The partitioning algorithms of Chapter Three were developed exclusively for mapping the matrix operations of addition, subtraction, scalar multiplication, and matrix multiplication onto a multiple processor architecture such as the MJH1. These algorithms have been shown to provide a much faster execution time when implemented on an appropriate multiple processor system. The MJH1 simply verifies the partitioning algorithms and allows for the intuitive evaluation of computational times associated with each algorithm.

The specifics of the MJH1 will be discussed at length in the subsequent sections of this chapter.

4.1 SYSTEM ARCHITECTURE

The MJH1 is classified as a Single Instruction-Stream Multiple Data-Stream machine. It exemplifies the following attributes:

- -a single controller
- -an array of 8 processing elements
- -an Input/Output scheme

-central memory

In the subsections that follow, the components of the multiprocessor will be thoroughly discussed.

4.2 MAJOR COMPONENTS OF THE MULTIPLE PROCESSOR ARCHITECTURE

A block diagram of the system architecture is given in Figure 4.1. A single control processor is connected to a maximum of eight processing elements (PE's). A data bus is accessible to all eight PE's via a Status Bit that is associated with each PE configuring the system. The bits are numbered from bit zero to bit seven, with the most significant bit of the status vector referring to PE 0 and the least significant bit of the status vector referring to the last (highest numbered) PE in the *configuration*. Instead of having an address bus that contains the address of specific PE's, they (PE's) are identified or are activated when their respective status bit is turned on (with a logical 1). The PE's are capable of reading from and writing to the central memory. Overall system I/O is done via the central memory. That is, individual processing elements do not have access to the outside world. They communicate with central memory, which in turn interacts with the outside world. Operand matrices are read into the PE's, the operations performed, and the results returned to the central memory of the control unit.

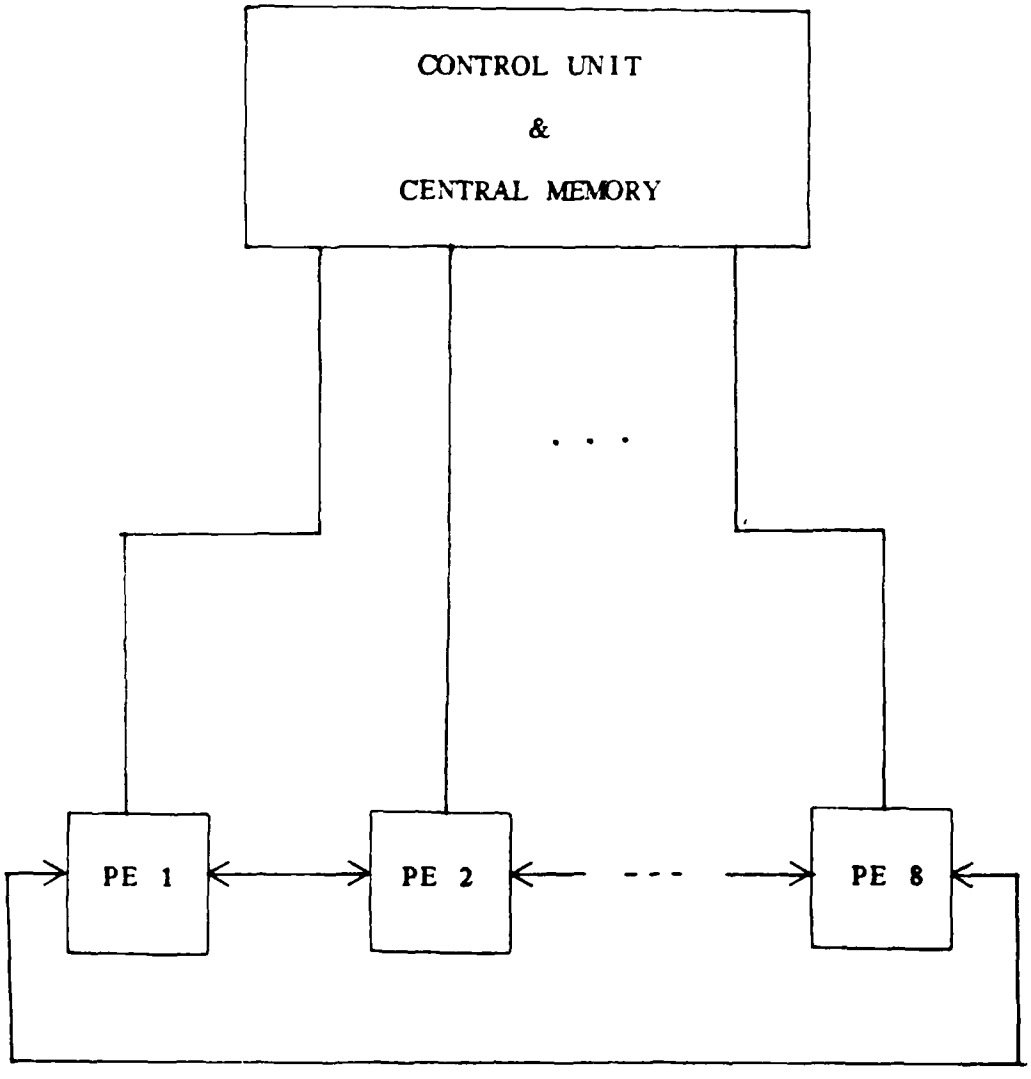


Figure 4.1

BLOCK DIAGRAM OF MULTIPROCESSOR ARCHITECTURE

4.2.1 The Controller

The controller is responsible for initializing all PE's at runtime. Initializing processing elements involves zeroing out all memory in the PE's and calculating their shift neighbors (should a shift function need to be performed). The controller is a general purpose computer with central memory and input/output capability.

The control processor is also responsible for program execution. A typical instruction execution proceeds in the following manner. The control processor fetches an instruction from central memory and broadcasts it to all PE's. Only the enabled PE's will be affected by the broadcasted instruction. The instruction is then decoded and executed by the appropriate PE's. The memory address register of the controller is then incremented and the next instruction is fetched, broadcasted, and executed. The cycle continues until a CPHalt instruction is invoked.

4.2.2 The Processing Element

Each processing element is identical to one another. A typical PE has an ALU of its own that can perform addition, subtraction, division, scalar multiplication, and matrix multiplication. A typical PE has its own memory, which is referred to as local memory. This local memory is partitioned into three sections: memory A (MA), memory B (MB), and memory R (MR), where A, B, and R refer to the operand matrices (A and B) and the result matrix R. Also, each enabled PE executes

instructions that are broadcasted to it by the controller. This execution is referred to as concurrent or parallel processing. A block diagram of the PE is shown in Figure 4.2.

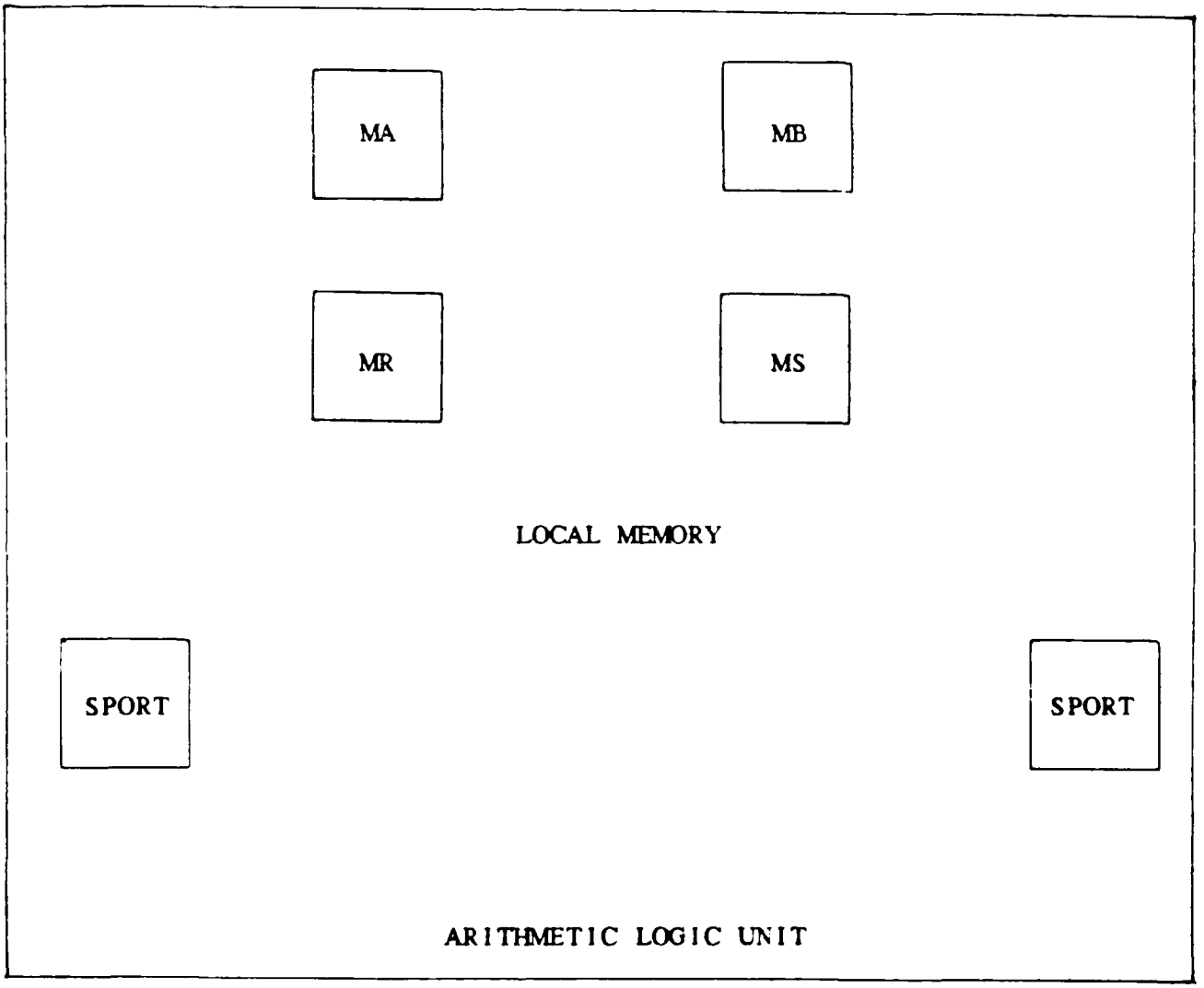


Figure 4.2

BLOCK DIAGRAM OF PROCESSING ELEMENT

4.2.3 I/O SCHEME

The Input/Output (I/O) of data is performed by the control processor. Data can either be written into central memory by various active processing elements, or it can be accessed by those same active processors. That is, each active processing element can write to as well as read from central memory. This data traverses along a common bidirectional data bus.

Along with reading and writing matrices, each PE is capable of sending and receiving data to or from other neighboring PE's via a shift network. The shift network consists of a shift port (SPORT) and a shift memory. This shift memory (MS) is also located within the local memory of each PE.

4.2.4 CENTRAL MEMORY

The central memory (CM) contains the instruction sequence needed to perform a particular computation. Also contained in central memory are the operand matrices, A and B. After the result matrix, R, has been generated via the active PE's, it, too, is stored into central memory, where it awaits to be accessed and written out to the outside world.

4.3 THE MJH1 EMULATOR

The MJH1 Emulator is a software tool that emulates the multiprocessor architecture that has been previously described. This emulator was first written by E. Jones [3], and later modified by the author of this thesis. The emulator is written in Pascal mainly because it is a structured programming language

and is more conducive to describing the various modules that are necessary in defining this SIMD architecture in software. Also, Pascal is a fairly simple language. A program listing of the emulator is included in Appendix A, however, the major components of the emulator will be discussed here.

Since the MJH1 emulates a machine that is capable of executing instructions to perform the partitioning of matrices, it is, in essence a simulated computer. Thus, in order to show that this emulator is indeed practical, a discussion of the type of instructions that must be included in a practical computer follows.

A computer should have a set of instructions that allows the user to formulate any conceivable data processing task. To ensure this, the computer (or in this case, the emulator) must include a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions.
2. Instructions for moving information to and from memory and processor registors.
3. Instructions that check status information to provide decision making capabilities.
4. Input and output instructions.
5. The capability of stopping the computer.

Since the MJH1 provides instructions from each category listed, it is a functionally complete machine [13].

4.3.1 MAJOR COMPONENTS OF THE MJH1

The MJH1 is composed of several modules. One such module defines the global constants that are used in defining the actual configuration of active processors. A definition of the instruction set comprises another module of the emulator. Next, a declaration of the data types used in defining variables used in the emulator are included. Variables required to successfully run the emulator are contained in yet another module. The module which has the most impact, however, is the processor module. This module contains code for executing the overall instruction set, which consists of both control processor and processing element instructions. This section also contains the Pascal code for processor initialization, as well as for processor-state dumping into the trace file. These modules will be further discussed in the subsections that follow.

4.3.1.1 GLOBAL CONSTANTS

Table 4.1 contains a listing of the global constants that are used throughout the emulator. These constants define the actual configuration of each active processing element.

TABLE 4.1

```
(* Global Constants: These define the actual configuration *)

const

   MAXNP = 8 ;          { Max Number of arithmetic processors.  }
   NPM1 = 7;            { Number of processors minus one.       }
   MEMMAX = 256;        { Size of each AP local memory.         }
   CPMEMMAX = 2048;     { Size of central memory (CM).          }
   SHREGMAX = 1;        { Size of shift-register memory.        }
   MAXDATAVAL = 25600;   { Hypothetical overflow value.        }

   zeroval = 0;         { Constant for integer data type.       }
   trueval = '1';       { Char representation of logical "true".}
   falseval = '0';      { Char repr'n of logical "false".       }
```

Overall central memory of the multiprocessor architecture is simulated here, along with the individual processors.

4.3.1.2 THE INSTRUCTION SET

The MJH1 consists of forty instructions. These instructions are of two types: Processing Element (PE) instructions and Control Processor (CP) instructions. Two instruction sets are necessary due to the nature of the multiprocessor architecture that is being emulated. The instruction set for the MJH1 is summarized in Table 4.2. Note that PE instructions have two digit opcodes, while CP opcodes have three digit opcodes.

TABLE 4.2

```
(* PE INSTRUCTION SET MNEMONICS AND OPCODES.                    *)

     { MISCELLANEOUS INSTRUCTIONS. }
NOOP = 0;          { No Op: Allows documentation of code.  }

     { ADDRESSING INSTRUCTIONS. }
ADBASE = 10;       { Advance LM base register.             }
SETB = 11;         { Set LM base register to vector origin.}
ALLOC = 12;        { Allocate c more words in LM.          }
SETMA = 15;        { Set MAR to address in CM.             }
ADMA = 16;         { Advance MAR by specified value.       }

     { CM ACCESS and SHIFT INSTRUCTIONS. }
LOADCM = 20;       { Load into LM from CM.                 }
STORCM = 21;       { Store into CM from LM.                }
FCSHF = 22;        { Forward circular shift.               }
BCSHF = 23;        { Backward circular shift.              }
PSSHF = 24;        { Perfect shuffle shift.                }
SHFIN = 25;        { Receive shifted data.                 }
SHIFX = 26;        { Put data on shift port.               }

     { DATA MOVEMENT INSTRUCTIONS. }
MOVA = 30;         { Move data to MA from another LM.      }
MOVB = 31;         { Move data to MB from another LM.      }
MOVR = 32;         { Move data to MR from another LM.      }
MOVS = 33;         { Move data to MS from another LM.      }

     { SCALAR ARITHMETIC INSTRUCTIONS. }
SADD = 40;         { Add scalar values in MA and MB.       }
SSUB = 41;         { Subtract scalar values in MA and MB.  }
SMPY = 42;         { Multiply scalar values in MA and MB.  }
SDIV = 43;         { Divide scalar values in MA and MB.    }
SMSA = 44;         { Add MS[i] to a LM.                    }

     { VECTOR-SCALAR ARITHMETIC INSTRUCTIONS. }
     {    NOTE: Vector in MA, Scalar in MB.   }
VSADD = 50;        { Add scalar to vector.                 }
VSSUB = 51;        { Subtract scalar from vector.          }
VSMPY = 52;        { Multiply vector by scalar.            }
VSDIV = 53;        { Divide vector by scalar.              }

     { VECTOR-VECTOR ARITHMETIC INSTRUCTIONS. }
PVADD = 60;        { Pair-wise vector addition.            }
PVSUB = 61;        { Pair-wise vector subtraction.         }
PVMPY = 62;        { Pair-wise vector multiplication.      }
INPRD = 65;        { Vector inner product.                 }

     { STATUS-CHECKING INSTRUCTIONS. }
TST = 70;          { Test statusword for error condition.  }
```

```
(* CP INSTRUCTION SET MNEMONICS AND OPCODES.                    *)

     { CONTROL INSTRUCTIONS. }
ENABLE = 100;        { Enable processors specified by mask. }
PUSHM = 101;         { Push current active mask onto MSTACK.}
PULLM = 102;         { Restore (pull) mask from MSTACK.      }
SETT = 103;          { Set MJH1 TRACE level.                 }
CPHALT = 255;        { Shut down Control Processor.          }

     { I/O INSTRUCTIONS. }
MREAD = 110;         { Read M x N matrix into CM.            }
MREADB = 113;        { READ M x N matrix [B] into CM         }
MWRITE = 111;        { Write M x N matrix stored in CM.      }

     { DATA MANAGEMENT INSTRUCTIONS. }
SETV = 120;          { Set range of CM locations to value.   }
```

Each instruction consists of an opcode followed by three operands. An illustration of the instruction format is shown below in Figure 4.3.

OPCODE OP1 OP2 OP3

FIGURE 4.3

These operands, in general specify addressing modes. For example, the following symbols can be used for operand one depending upon the particular instruction desired.

```
m := local memory specifier
     1 => MA
     2 => MB
     3 => MR
     4 => MS

v := an immediate value

c := a count

i := an index, or immediate value

p := a specific processing element

k := # of column values for matrix multiplication
```

IA := index for MA

t := trace level
0 => no trace
1 => processor state only
2 => full processor state

Symbols for operand two are similar to those of operand one. However, all symbols used in operand one are not used in operand two. A listing of these operands follow.

i := index or immediate value

v:= an immediate value

c := a count

p := PE number

m := column dimension of a matrix

IB := index for MB

There is only one specified symbol for operand three. Despite this fact, an operand must be supplied in each field for each instruction that is to be executed. For that reason, zeros must be supplied in a particular field when no opcode is required. The defined symbol for operand three follows.

m := column dimension of a matrix

Consider, for example, the instruction, 12 2 2 0. This instruction says to allocate two spaces in local memory B. Operands two and three correspond to m and c, and operand three is a zero because it is not needed. The programmer is referred to Appendix A, more specifically to pages A-10 through A-14, and A-16, for the correct placement of these symbols in a specific instruction.

The emulator has the capability of performing both vector and scalar operations. Vector operations are those that involve streaming data to one processor (serially), then to the next, and so on. Scalar operations involve interleaving or performing operations concurrently on all processors simultaneously. Upon examining the instruction set of the MJH1, the user will be able to distinguish the vector instructions from the scalar instructions by their groupings.

4.3.1.3 DATA TYPES

A listing of the various types of data used in the emulator are shown in Table 4.3. These "type" statements are similar to the declaration statements used in Fortran.

TABLE 4.3

DATA TYPES

```
dataval = integer;                    { Data Type of array     }
matrixdim = 0..31;                    { Max   rows / cols      }
matrix = array[matrixdim,matrixdim]
           of dataval;
                                      {    data values.        }
PEmask = record                       { Bit string indicating  }
           bit: array[0..NPM1]        {   active PEs.          }
                  of boolean;
         end;
maskstack = array[1..10]  { Stack of active PE masks.     }
           of PEmask;
maskstr = array[0..NPM1]  { String version of PE mask.    }
           of char;

statusword = array[0..3]              { 4-bit Status word.     }
               of boolean;            { See Below.             }
   { bit zero:  1 = enabled/noncompletion.                     }
   { bits 1-2:  00 = no exception.                             }
   {            01 = arithmetic exception:                     }
   {                 bit 3: 0 = zero divide;                   }
   {            10 = machine exception.                        }
   {            11 = operand addressing exception.             }
   {                 bit 3: 0 = address range;                 }
statustr = array[0..3]   { String version of status word.}
             of char;
datamem = array[0..MEMMAX]            { local A,B,R memories.  }
             of dataval;
shiftmem = array[0..SHREGMAX]         { Shift register memory.}
             of dataval;
cpmem = array[0..CPMEMMAX]            { The Central Memory.    }
             of dataval;

instruction = record
                opcode:integer; { Operation code.         }
                op1:integer;    { First operand.          }
                op2:integer;    { Second operand.         }
                op3:integer;    { Third operand.          }
              end;

opcodeset = set of 0..CPHALT;     { Valid processor
opcodes.}
```

```
processor = record
              PROCID:integer;      { Procesor ID.                }
              ACC:dataval;         { Accumulator.                }
              MAR:integer;         { Memory address reg.         }

                                   { Processor ID for shifting.  }
              FCSID:integer;       { Forward circular.           }
              BCSID:integer;       { Backward circular.          }
              PSSID:integer;       { Perfect Shuffle.            }

                                   { Local Memories.             }
              MA:datamem;          { A-operand Memory.           }
              MB:datamem;          { B-operand Memory.           }
              MR:datamem;          { Result Memory.              }
              MS:shiftmem;         { Shift registers.            }
              SPORT:shiftmem;      { Inter-processor shift       }
                                   { port.

                                   { Base Registers for Local Mem's}
              MAB:integer;         { Base register for MA.       }
              MBB:integer;         { Base register for MB.       }
              MRB:integer;         { Base register for MR.       }

                                   { Current Bounds-Reg for LM's.  }
              MAH:integer;         { Hi in-use MA address.       }
              MBH:integer;         { Hi in-use MB address.       }
              MRH:integer;         { Hi in-use MR address.       }

              STATUS:statusword; { condition code bits.          }

            end; { processor record. }
```

4.3.1.4 REQUIRED VARIABLES

Table 4.4 contains a listing of the variables required to run the emulator. Some of these variables must be supplied by the programmer. These variables are the four files: code, operand, trace, and result; the number of processors in the multiprocessing configuration, and the level of trace desired. The other variables listed, unless contained in the files supplied by the user, are generated by the emulator. These entities will be further discussed later.

TABLE 4.4

REQUIRED VARIABLES

```
codefile:text;          { File containing MJH1 code along  }
                        {   with immediate data.           }
oprndfile:text;         { File containing matrix values.   }
tracefile:text;         { File to contain full processor   }
                        {   execution trace.               }
resultfile:text;        { File to contain "result" (i.e.,  }
                        {   data transmitted by "SEND"     }
                        {   instructions.                  }
TRACE:integer;          { Trace enable flag.               }

IR:instruction;         { Broadcasted instr. register.     }
PE: array[0..NPM1]      { The MJH1 network of PE's.        }
    of processor;
NP:integer;             { Number of processors in use.     }
NAP:integer;            { Number of currently active PEs.  }
A,B,R:matrix;           { Operand and result matrices      }
CM:cpmem;               { The CP Central Memory.           }
CMHI:integer;           { Highest in-use CM address.       }
ACTIVEMASK:PEmask;      { Current active processor mask.   }
MSTACK:maskstack;       { Stack of active PE masks.        }
MSTKTOP:integer;        { Stack pointer for MSTACK.        }
COMPLETED:boolean;      { Flag set when active PEs         }
                        {   complete instruction.          }
PEOPCODES:opcodeset;    { Set of valid PE opcodes.         }
CPOPCODES:opcodeset;    { Set of valid CP opcodes.         }
i:integer;              { Work variable.                   }
P:INTEGER;
```

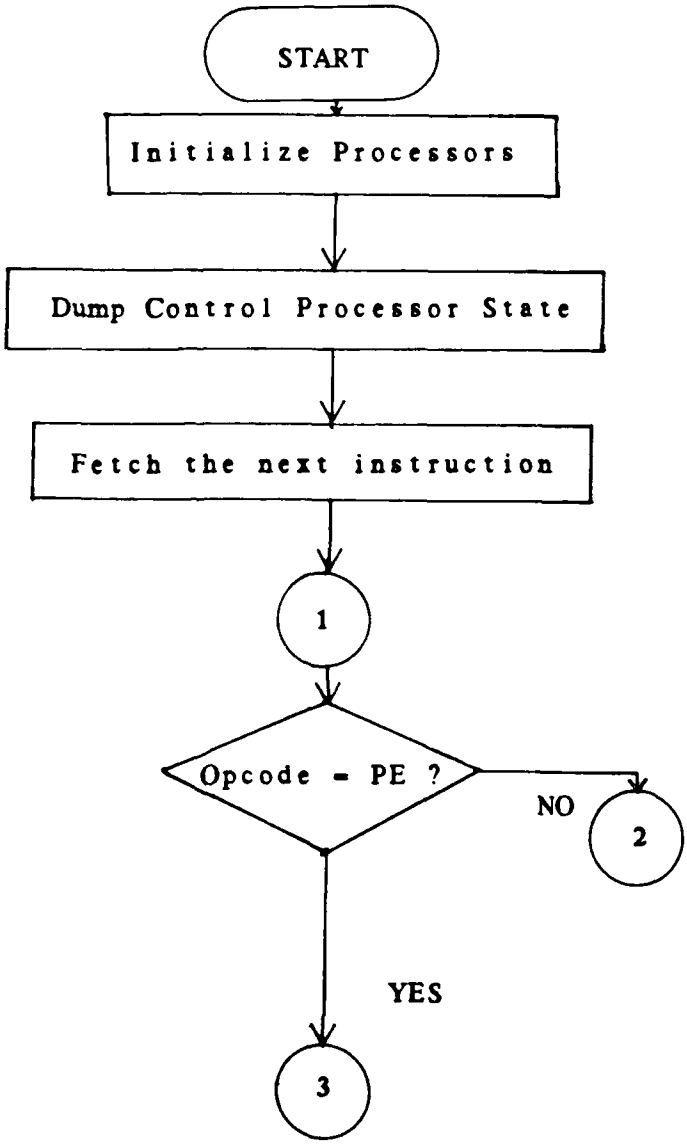
4.3.1.5 PROCESSOR MODULE

This module is analogous to the control processor of the multiprocessor architecture in that it is the heart of the emulator. Like the controller, the processor module is responsible for clearing all processor memories and identifying all shift neighbors for each processing element configured in the multiprocessing network. Along with this initialization, this module also performs all instruction execution. That includes the I/O scheme, also, since there are specific instructions for reading and writing data among processors, central memory, and the outside world. This function is also compatible with that of

the controller, in that the controller is responsible for all data I/O.

Instruction execution proceeds as follows. Upon initialization, the memory address register (MAR) of the controller is set to a negative one (-1). This is done so that once the MAR is incremented, it will point to the current instruction. The MAR is incremented at the end of each instruction decode and execution cycle. The current instruction is fetched, and then compared first to the Processing Element opcodes, and then to the Control Processor opcode, in the event that the instruction was not a PE opcode. Once a match has been made and the appropriate instruction has been identified, then it is decoded as per the detailed lines of code included in the emulator. The MAR is incremented and the next instruction is fetched, decoded, and executed. This process continues until a CPHalt instruction is issued.

Major sections of Pascal code from the processor module perform the initialization of processors, instruction decoding, and processing dumping. The relevant procedures, as they are called in Pascal, are named appropriately, and can be further studied in Appendix A. A basic flow chart of the overall processor module is included in Figure 4.4. The actual procedures are included in Appendix A, where the entire emulator is located.


START
Initialize Processors
Dump Control Processor State
Fetch the next instruction
1
Opcode = PE ?
NO
2
YES
3

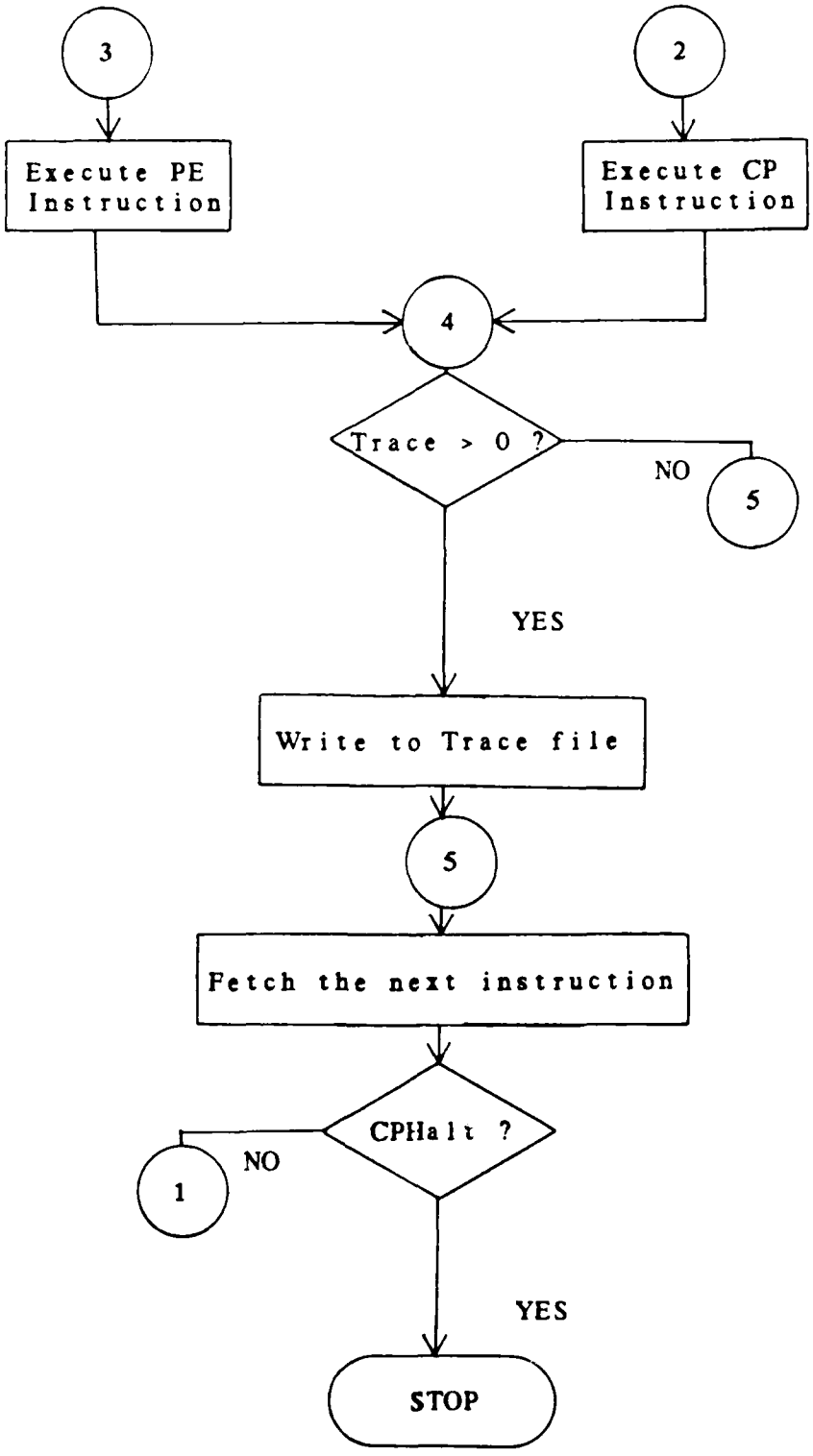


FIGURE 4.4

FLOW CHART OF PROCESSOR MODULE

With each instruction fetch (Ifetch), pertinent information is written into the trace file by the processor module if the level of trace is greater than 1. This trace file will prove invaluable as the complexity of the functions performed by the emulator increases.

4.3.2 PROGRAMMING THE MJH1

Before invoking the emulator and successfully performing various matrix operations, the user must first become familiar with specifics of the emulator itself. Following are details of the emulator that should aid the programmer/user in obtaining favorable results.

There are four files that must be assigned to the MJH1 before a successful run can be made. These files are the operand file, code file, trace file, and the result file. The first two are input files that must be created prior to runtime, while the latter two are output files that are generated by the emulator itself.

4.3.2.1 OPERAND FILE

The operand file, oprndfile, is the file which contains the two operand matrices A and B. In this thesis, matrix multiplication will be looked at exclusively. Thus, the dimensions of A must be M by N, while matrix B must be N by P, yielding a result matrix with the dimensions M by P. A sample operand file follows.

```
8
6
                    Matrix A (4 x 1)
4
2

10   20             Matrix B (1 x 2)
```

Notice how the data file (whose extension is .mtx) is written in matrix form.

4.3.2.2 CODE FILE

The code file contains the instructions to be executed by the emulator. As stated before, there are two types of instructions, Processing Element (PE) instructions and Control Processor (CP) instructions. Two instruction sets are necessary due to the nature of the multiprocessor architecture that is being emulated. A listing of the MJH1 instruction set is shown in Table 4.2.

The instruction format of the emulator is shown in Figure 4.3. The programmer must always supply three operands with an opcode. This is true even if all three fields are not being used. In this case, zeros must be supplied in the appropriate operand field. In general, the PE opcodes contain only 2 digits, while CP opcodes contain 3 digits. Operands 1 and 2 generally specify the source of data, while operand 3 specifies its destination after some operation has been performed. In the emulator, however, required operands are coded as follows:

m := local memory specifier
1 => MA
2 => MB
3 => MR
4 => MS

v := an immediate value

c := a count

i := an index, or immediate value

IA, IB, IR := indices for MA, MB, MR, respectively

In order to enable processors, the ENABLE insruction must be used. Operand one of this instruction is the decimal equivalent of the binary string of processors comprising the system. A '1' indicates that a certain processor is turned on, or enabled, while a '0' indicates that a certain processor is turned off. In this binary string of processors, the most significant bit represents processing element 0 while the least significant bit represents the highest numbered PE in the configuration. For example, if two PE's comprise a system, then

100 2 0 0

means to enable PE 0 only, while

100 3 0 0

means to enable all processors. Remember that the binary representation of 2 is 10, and the binary representation of 3 is 11.

4.3.2.3 TRACE FILE

The trace file is a file that contains a detailed account of all activity within the emulator on each clock cycle. There are varying levels of tracing available:

1) no trace
2) processor state only
3) full processor trace

Level 0 is the lowest, simplest level of trace, while level 2 is the most comprehensive trace level. Although the full trace is lengthy and consumes a significant amount of space, it is a major feature of the emulator, in that it allows one to see what happens to every active processor during each clock cycle. In case of an erroneous result, the source can easily be traced.

4.3.2.4 RESULT FILE

The result file contains the operand matrices along with the result file that is created by the MJH1 after simulation.

4.4 FILE NAMING CONVENTIONS

Following is a table that describes the way that files should be named, Table 4.5.

TABLE 4.5

FILE NAMING CONVENTIONS

EXTENSION	DESCRIPTION
.mtx	Operand file containing two matrices A & B. Prefix of filename should specify dimensions or algorithm with which it is to be used. eg., MEQKNP4, a file containing 2 matrices with M equal to number of processors (K), and PE = 4.
.cod	Code file that is read as input by the emulator. Note that the code must be in absolute form--no symbolic coding is allowed.
.trc	Trace file that is created while the emulator is executing instructions. The trace level (0,1,2), which determines the volume of trace output, can be set at the start of execution by the user, or it can be set by the programmer using the SETT instruction.
.res	Result file that is created by MJH1. This file contains both operand matrices, followed by the result matrix.

4.5 CREATING THE CODE FILE

This is perhaps the most important file that must be submitted to the MJH1. This is the file that performs the matrix partitioning based on the partitioning algorithms of Chapter Three. This file can be easily generated by hand, although the programmer must be familiar with the assembly code that is specific to the MJH1. Thus, some background in assembly language programming, i.e. microprogramming, is very helpful in the programming of this machine.

Before the code can be written, however, the programmer must know several "design" parameters. They are as follows. He must know the number of processors that will be active in the configuration and he must also know the dimensions of the operand

matrices. From this basic information, the correct partitioning algorithm can be chosen for the most efficient allocation of values to the active processors.

The code file contains three sections as shown in Figure 4.5. The first section involves allocating space in central memory for three matrices: matrix A, matrix B, and matrix R. The second section defines the partitioning of the operand matrices. The third and final section of the code file performs matrix multiplication (or whatever matrix operation desired), and writes the result matrix back to central memory. A sample code file with comments follows in Figure 4.6. This sample is an implementation of partitioning algorithm for matrix multiplication when the number of rows, m, is a multiple of the number of active processors. This is the algorithm numbered 3b which can be referenced in Chapter Three, Section Three under the same, 3b. Note that the first instruction is a no-op whose comment statement describes the way in which each martix is partitioned and distributed among each active processor.

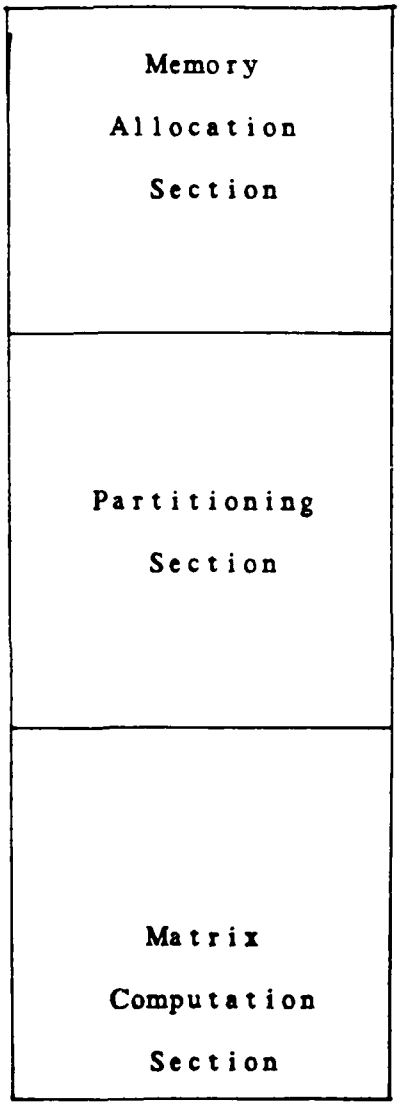


FIGURE 4.5

SECTIONS OF THE CODE FILE

FIGURE 4.6

SAMPLE CODE FILE

ALG3B: 4 x 1 BY 1 x 2; NP=2. [MMULTK]

```
  0   0   0   0    (m/k) rows of [A] and entire [B]
120  16   0   0    Clear 16 words in CM
110   0   4   1    Read matrix A                          [MEMORY
111   0   4   1    Echo A                                 ALLOCATION]
110   4   1   2    Read matrix B and echo
111   4   1   2    Echo B

  0   0   0   0    Start partitioning
100   3   0   0    Enable ALL PE's to load B
 15   4   0   0    PEs at B[1,1]
 12   2   2   0    Allocate space in MB for 2 values
 20   2   2   0    Move 2 values into MB
100   2   0   0    Enable PE 0
 15   0   0   0    Set address to A[1,1]
100   1   0   0    Enable PE 1                            [PARTITIONING]
 15   2   0   0    Set address to A[3,1]
100   3   0   0    Enable ALL PEs
103   3   0   0    Start TRACE
 12   1   2   0    Allocate space for A
 20   1   2   0    Load various rows into MA
  0   0   0   0    Prepare to multiply
 11   1   0   0    Set MA to 0
 11   2   0   0    Set MB to 0
 11   3   0   0    Set MR to 0
 12   3   4   0    Allocate room in MR for 2 values

 65   0   1   0    Multiply
 11   1   0   0    Reset B
 11   2   1   0
 65   1   1   0
 11   1   1   0
 11   2   0   0
 65   2   1   0
 11   1   1   0
 11   2   1   0
 65   3   1   0                                          [MATRIX
100   2   0   0    Enable P0                              COMPUTATION]
 15   6   0   0    Set MAR to 1st result
100   1   0   0    Enable P1
 15  10   0   0    Set MAR to 2nd result
100   3   0   0    Enable ALL PEs
 11   3   0   0
 21   4   0   0    Store 1st results
111   6   4   2    PRINT
103   0   0   0    STOP TRACE
255   0   0   0    CPHALT
```

Note that special care must be taken when writing the result matrix back to the central memory. The MJH1 architecture is defined such that data is read and stored in a linear, one-dimensional array. In other words, each data word is stored in sequential memory locations. Also, matrices are read in row order. If matrix A contained two rows and three columns, it would be read into central memory by rows. The first element of row 1 would be followed by the second element of row 1, etc., until the third element of row 2 was read (matrix A has the dimensions of 2 by 3). Thus, after the operations have been performed by all active PE's, the programmer must be aware of which processor computed what result, then make sure that the result is written back to the proper location in central memory.

After the code file has been created in accordance to the particular algorithm, the MJH1 may be invoked, thereby testing the correctness of the partitioning algorithm, and of the code file created by the programmer. Remember that the burden of programming to exploit the inherent parallelism in such an environment is placed solely on the programmer. Thus, the correctness of the codefile determines the correctness of the simulation.

4.6 RUNNING THE MJH1

The MJH1 emulator is very easy to operate. The user/programmer must first have access to an account on the VAX 11/780. In order to obtain an account, see the system operator. After one has an account and has successfully logged onto the

system, the emulator must be made accessible to the user's account. This can be done by setting the default to the residence of the emulator as follows:

SET DEF [HUGHES.EMULATOR]

This command allows the programmer access to the MJH1, and to previously created files relative to the operation of the emulator. The user should feel free to explore and examine the existing files by typing them out onto the terminal. This will further familiarize the user with the machine and its operation. The command: RUN MJH1EM invokes the emulator. Note that all files being run on or sent to the emulator should reside in [HUGHES.EMULATOR]. Therefore, the user must either create both the operand file and code file in that directory, or must copy them over to [HUGHES.EMULATOR] before invoking the emulator.

The emulator will prompt the user for the various filenames, the number of active procesors, and for the level of tracing as follows:

```
ENTER NAME OF OPERAND FILE :
ENTER NAME OF CODE FILE :
ENTER NAME OF RESULT FILE :
ENTER NAME OF TRACE FILE :
ENTER NUMBER OF PROCESSORS :
ENTER PROCESSING TRACING LEVEL
        0) NO TRACE
        1) PROCESSOR STATE ONLY
        2) FULL PROCESSOR TRACE
```

After these parameters have been entered, the emulator goes to work and executes each instruction. In other words, it does

exactly what it is programmed to do. When the CPHalt command has been executed, the emulator stops, and the message

MJH1 SHUT-DOWN!!

is displayed on the screen. To see what has just transpired, the result file and the trace file should be either typed or printed. Upon examination of these files, one will be able to tell if the run was successful or not.

A demonstration of the MJH1 Emulator is found in Appendix C. Algorithm 3B of Chapter Three is implemented here, the same algorithm that is used in Figure 4.6. This demonstration includes a sample operand file, code file, result file, and trace file.

CHAPTER FIVE

SUMMARY AND SUGGESTIONS

5.0 SUMMARY

In summation, the MJH1 is a very good software tool to use in the analysis of the correctness of various partitioning algorithms. It has the potential to be a forerunner in the discipline of Computer Engineering at North Carolina A & T State University. This is possible because the emulator is easily adaptable to fit system architectures other than SIMD. Through simulation, different architectures can be implemented, as well as various algorithms for exclusive partitioning of matrices to perform simple matrix operations in a parallel or multiprocessor environment.

The MJH1 Emulator has the potential to be expanded tremendously. The instruction set could grow immensely. Also, the emulator could be made into a smarter, more intelligent machine by combining some existing commands and making its language of a higher level.

5.1 SUGGESTIONS

Suggestions for future work include devising a way to generate code necessary for the operation of the MJH1 in a systematic, automatic fashion. Upon careful examination of all existing code files, one can see the existence of a pattern. Due to time constraints, the author did not have time to decipher the pattern, however, she acknowledges the fact that a pattern does

exist. Thus, a code generator should be looked into in order to make the MJH1 a more attractive entity. The programmer can see, by examining some of the existing codes, that programming the MJH1 by hand can be excrutiatingly long and frustrating.

Also, the algorithms for matrix inversion should be looked into and possibly implemented on the emulator. After these two feats have been successfully accomplished, the author feels that the emulator should be expanded to be able to handle more PE's. Currently, the maximum number of PEs in a system configuration is only eight. However, before adding more PEs in the configuration, the controller should be expanded to include error detection and/or error correction.

BIBLIOGRAPHY

1. Rodrigue, Garry, Parallel Computations, Academic Press, 1982, pp 1 - 49.

2. Sadhasivan, S., "Multiprocessing Configuration for Matrix Computations," Thesis, North Carolina A. & T. State University, 1984.

3. Martin, Sr., Harold L., and Jones, Ed, "Analysis of Parallel Matrix Computations on an Emulated Multiprocessor-Based SIMD Multiprocessor," Report, North Carolina A. & T. State University, August 1985.

4. Dertougas, M. L., "The Multiprocessor Revolution: Harnessing Computers Together," Technology Review, Vol. 89, Feb/March 1986, pp 44 - 54.

5. Deposito, J.,"Working Faster Together," Computers and Electronics, Vol. 22, May 1984, pp 73 - 74.

6. Port, O. "Superfast Computers: You Ain't Seen Nothin' Yet," Business Week, August 26. 1985, pp 91-92.

7. Stone, H. S., "Parallel Computers," Introduction to Computer Architecture, Science Research Associates, Chicago, 1975, pp 318 - 374.

8. Browne, J., C., "Parallel Architectures for Computer Systems," Physics Today, Vol. 37, May 1984, pp 28 - 35.

9. Feng, Tse-yun, "A Survey of Interconnection Networks," Computer, December 1981, pp 12 - 27.

10. Martin, Sr., Harold L. and Sadhasivan, S., "A Parallel MIMD Machine for Matrix Computations," Proc. of the Sixteenth Southeastern Symp. on System Theory, March 1984, pp 58 - 61.

11. Martin, Sr., Harold L. and Sadhasivan, S.,"Partitioning Algorithms for Matrix Computations in a Multiprocessing Environment," Proc. of the Third International Conference on Systems Engineering, Sept. 1984, pp 287 - 294.

12. Martin, Sr., Harold L. and Sadhasivan, S., "An Array Processor for Matrix Computations," Proc. of the Eighteenth Annual Asilomer Conference on Circuits, Systems, and Computers, Nov. 1984.

13. Hayes, John P., Computer Architecture and Organization, McGraw-Hill, Inc., 1978, pp 160 - 171.

APPENDIX A

PROGRAM LISTING OF THE MJH1 EMULATOR

```
program
MJH1(input,output,codefile,oprndfile,tracefile,resultfile);

    { INCLUDE MJH1.VAR }

    (* Declaration of MJ-1 Arithmetic Processor State Register. *)
    (*                                                          *)
    (* Global Constants: These define the actual configuration  *)
    (*                                                          *)

    const

        MAXNP = 8 ;          { Max Number of arithmetic processors.  }
        NPM1 = 7;            { Number of processors minus one.       }
        MEMMAX = 256;        { Size of each AP local memory.         }
        CPMEMMAX = 2048;     { Size of central memory (CM).          }
        SHREGMAX = 1;        { Size of shift-register memory.        }
        MAXDATAVAL = 25600;   { Hypothetical overflow value.        }

        zeroval = 0;         { Constant for integer data type.       }
        trueval = '1';       { Char representation of logical "true".}
        falseval = '0';      { Char repr'n of logical "false".       }


        (* PE INSTRUCTION SET MNEMONICS AND OPCODES.                 *)

            { MISCELLANEOUS INSTRUCTIONS. }
        NOOP = 0;            { No Op: Allows documentation of code.  }

            { ADDRESSING INSTRUCTIONS. }
        ADBASE = 10;         { Advance LM base register.             }
        SETB = 11;           { Set LM base register to vector origin.}
        ALLOC = 12;          { Allocate c more words in LM.          }
        SETMA = 15;          { Set MAR to address in CM.             }
        ADMA = 16;           { Advance MAR by specified value.       }

            { CM ACCESS and SHIFT INSTRUCTIONS. }
        LOADCM = 20;         { Load into LM from CM.                 }
```

```
    STORCM = 21;        { Store into CM from LM.                    }
    FCSHF = 22;         { Forward circular shift.                   }
    BCSHF = 23;         { Backward circular shift.                  }
    PSSHF = 24;         { Perfect shuffle shift.                    }
    SHFIN = 25;         { Receive shifted data.                     }
    SHIFX = 26;         { Put data on shift port.                   }
        { DATA MOVEMENT INSTRUCTIONS. }
    MOVA = 30;          { Move data to MA from another LM.          }
    MOVB = 31;          { Move data to MB from another LM.          }
    MOVR = 32;          { Move data to MR from another LM.          }
    MOVS = 33;          { Move data to MS from another LM.          }

        { SCALAR ARITHMETIC INSTRUCTIONS. }
    SADD = 40;          { Add scalar values in MA and MB.           }
    SSUB = 41;          { Subtract scalar values in MA and MB.      }
    SMPY = 42;          { Multiply scalar values in MA and MB.      }
    SDIV = 43;          { Divide scalar values in MA and MB.        }
    SMSA = 44;          { Add MS[i] to a LM.                        }

        { VECTOR-SCALAR ARITHMETIC INSTRUCTIONS. }
        {    NOTE: Vector in MA, Scalar in MB.   }
    VSADD = 50;         { Add scalar to vector.                     }
    VSSUB = 51;         { Subtract scalar from vector.              }
    VSMPY = 52;         { Multiply vector by scalar.                }
    VSDIV = 53;         { Divide vector by scalar.                  }

        { VECTOR-VECTOR ARITHMETIC INSTRUCTIONS. }
    PVADD = 60;         { Pair-wise vector addition.                }
    PVSUB = 61;         { Pair-wise vector subtraction.             }
    PVMPY = 62;         { Pair-wise vector multiplication.          }
    INPRD = 65;         { Vector inner product.                     }

        { STATUS-CHECKING INSTRUCTIONS. }
    TST = 70;           { Test statusword for error condition.      }


(* CP INSTRUCTION SET MNEMONICS AND OPCODES.                        *)

        { CONTROL INSTRUCTIONS. }
    ENABLE = 100;       { Enable processors specified by mask.
}
    PUSHM = 101;        { Push current active mask onto MSTACK.
}
    PULLM = 102;        { Restore (pull) mask from MSTACK.
}
    SETT = 103;         { Set MJH1 TRACE level.
}
    CPHALT = 255;       { Shut down Control Processor.
}

        { I/O INSTRUCTIONS. }
    MREAD = 110;        { Read M x N matrix into CM.
}
    MREADB = 113;       { READ M x N matrix [B] into CM
```

```
}
        MWRITE = 111;       { Write M x N matrix stored in CM.
}

        { DATA MANAGEMENT INSTRUCTIONS. }
        SETV = 120;         { Set range of CM locations to value.
}


    type
        dataval = integer;                      { Data Type of array    }

        matrixdim = 0..31;                      { Max   rows / cols     }

        matrix = array[matrixdim,matrixdim]
                    of dataval;

                                                {    data values.       }
        PEmask = record                         { Bit string indicating }
                    bit: array[0..NPM1]         {   active PEs.         }
                            of boolean;
                 end;
        maskstack = array[1..10]  { Stack of active PE masks.    }
                    of PEmask;
        maskstr = array[0..NPM1]  { String version of PE mask.   }
                    of char;

        statusword = array[0..3]                { 4-bit Status word.    }
                        of boolean;             { See Below.            }
            { bit zero:  1 = enabled/noncompletion.                     }
            { bits 1-2:  00 = no exception.                             }
            {            01 = arithmetic exception:                     }
            {                 bit 3: 0 = zero divide;                   }
            {            10 = machine exception.                        }
            {            11 = operand addressing exception.             }
            {                 bit 3: 0 = address range;                 }
        statustr = array[0..3]   { String version of status word.}
                    of char;

        datamem = array[0..MEMMAX]              { local A,B,R memories. }
                    of dataval;
        shiftmem = array[0..SHREGMAX]           { Shift register memory.}
                    of dataval;

        cpmem = array[0..CPMEMMAX]              { The Central Memory.   }
                    of dataval;

        instruction = record
                        opcode:integer; { Operation code.       }
                        op1:integer;    { First operand.        }
                        op2:integer;    { Second operand.       }
                        op3:integer;    { Third operand.        }
                      end;
```

```
        opcodeset = set of 0..CPHALT;     { Valid processor opcodes.
}

        processor = record
                        PROCID:integer;     { Procesor ID.          }
                        ACC:dataval;        { Accumulator.          }
                        MAR:integer;        { Memory address reg.   }

                                  { Processor ID for shifting.      }
                        FCSID:integer;      { Forward circular.     }
                        BCSID:integer;      { Backward circular.    }
                        PSSID:integer;      { Perfect Shuffle.      }

                                  { Local Memories.                 }
                        MA:datamem;         { A-operand Memory.     }
                        MB:datamem;         { B-operand Memory.     }
                        MR:datamem;         { Result Memory.        }
                        MS:shiftmem;        { Shift registers.      }
                        SPORT:shiftmem;     { Inter-processor shift }
                                            { port.                 }

                                  { Base Registers for Local Mem's}
                        MAB:integer;        { Base register for MA. }
                        MBB:integer;        { Base register for MB. }
                        MRB:integer;        { Base register for MR. }

                                  { Current Bounds-Reg for LM's.    }
                        MAH:integer;        { Hi in-use MA address. }
                        MBH:integer;        { Hi in-use MB address. }
                        MRH:integer;        { Hi in-use MR address. }

                        STATUS:statusword; { condition code bits.   }

                    end; { processor record. }



    { INCLUDE MJH1.PEC }
    (* ********* REQUIRED VARIABLES *********
*)
    (* ------------------------------------------------------------
*)

    var
        codefile:text;          { File containing MJH1 code along  }
                                {   with immediate data.           }
        oprndfile:text;         { File containing matrix values.   }
        tracefile:text;         { File to contain full processor   }
                                {   execution trace.               }
        resultfile:text;        { File to contain "result" (i.e.,  }
                                {   data transmitted by "SEND"     }
                                {   instructions.                  }
        TRACE:integer;          { Trace enable flag.               }
```

```
        IR:instruction;          { Broadcasted instr. register.    }
        PE: array[0..NPM1]       { The MJH1 network of PE's.       }
            of processor;

        NP:integer;              { Number of processors in use.    }
        NAP:integer;             { Number of currently active PEs. }

        A,B,R:matrix;            { Operand and result matrices     }

        CM:cpmem;                { The CP Central Memory.          }
        CMHI:integer;            { Highest in-use CM address       }

        ACTIVEMASK:PEmask;       { Current active processor mask.  }
        MSTACK:maskstack;        { Stack of active PE masks.       }
        MSTKTOP:integer;         { Stack pointer for MSTACK.       }
        COMPLETED:boolean;       { Flag set when active PEs        }
                                 {   complete instruction.         }

        PEOPCODES:opcodeset;     { Set of valid PE opcodes.        }
        CPOPCODES:opcodeset;     { Set of valid CP opcodes.        }

        i:integer;               { Work variable.                  }
      P:INTEGER;



(* Processor Module: Contains code for executing the instruction
*)
(*    Set, processor initialization, processor-state dumping.
*)
(* ************************************************************
*)

(* Initialize processor registers.  *)
procedure initprocessor(var
PE:processor;pid,fcs,bcs,pss:integer);

   var i:integer;     { Work variable.  }

   begin { initprocessor }

      with PE do
      begin

         PROCID := pid; ACC := zeroval; MAR := -1;
         FCSID := fcs; BCSID := bcs;  PSSID := pss;
         MAB := 0;  MBB := 0;  MRB := 0;
         MAH := 0;  MBH := 0;  MRH := 0;

         for i := 0 to MEMMAX-1 do
         begin
            MA[i] := zeroval; MB[i] := zeroval; MR[i] := zeroval;
         end;
```

```
            for i := 0 to SHREGMAX-1 do
            begin
               MS[i] := zeroval; SPORT[i] := zeroval;
            end;

            for i := 0 to 3 do STATUS[I] := false;

         end; { with PE }

      end;  { initprocessor }

   (* Convert 4-bit status vector into character array. *)
   procedure convertstatus2str(var S:statusword;var C:statustr);

      var i:integer;       { Work variable.                    }

      begin
         for i := 0 to 3 do
              if S[i]
                 then C[i] := trueval
                 else C[i] := falseval;
      end; { convertstatus2str }

   (* Convert processor mask into character array. *)
   procedure convertmask2str(NP:integer; var MASK:PEmask;
                             var MSTR:maskstr);

      var i:integer;       { Work variable.                    }

      begin
         for i := 0 to NP-1 do
              if MASK.bit[i]
                 then MSTR[i] := trueval
                 else MSTR[i] := falseval;
      end; { convertmask2str }


   (* Dump the Control Processor State. *)
   procedure DumpCPState(NP,NAP,TRACE:integer; var
   ACTIVEMASK:PEmask);

      var
           i:integer;              { Work variable.
   }
           mstr:maskstr;           { String version of ACTIVEMASK.
   }
           nvals:integer;          {  vals on current line: skip
   when}
                                   {       12.
   }

      begin

         convertmask2str(NP,ACTIVEMASK,mstr);
```

```
        writeln(tracefile);
        write(tracefile,'CP STATE ==> ','TRACE =',TRACE:2,
                        '  NP =',NP:3,'  NAP =',NAP:3,
                        '  ACTIVEMASK =');
        for i := 0 to NP-1 do
            write(tracefile,mstr[i]:2);
        writeln(tracefile);

        if TRACE > 0 then
           begin
              nvals := 0;
              write(tracefile,' DUMP OF CM:',
                              CMHI:4,' WORDS IN USE.');
              writeln(tracefile);
              for i := 0 to CMHI do
              begin
                   if nvals = 12 then
                      begin
                         writeln(tracefile,' ':8);
                         nvals := 0;
                      end;
                  write(tracefile,CM[i]:6);
                  nvals := nvals + 1;
              end;
           end;

     end; { DumpCPState }

  (* Dump the processor state record. *)
  procedure dumprocstate(var PE:processor);

     var ST:statustr;    { Work array for displaying Status bits. }

     begin { dumprocstate }

        WITH PE do
        begin

           convertstatus2str(STATUS,ST);

           writeln(tracefile);
           writeln(tracefile,' ID ACC MAR FC BC PS  MS  SP',
                             ' MAB MBB MRB MAH MBH MRH  STATUS');
           writeln(tracefile,PROCID:3,ACC:4,MAR:4,
                             FCSID:3,BCSID:3,PSSID:3,
                             MS[0]:4,SPORT[0]:4,
                             MAB:4,MBB:4,MRB:4,MAH:4,MBH:4,MRH:4,
                             ST[0]:3,ST[1]:2,ST[2]:2,ST[3]:2);
           writeln(tracefile);

        end; { WITH PE }

     end;  { dumprocstate }

  (* Dump the specified local memory *)
```

```
procedure dumplocalmem(MEMTYPE:char;var
MEM:datamem;MEMHI:integer);
   var i:integer;    { Work variable. }

   begin { dumplocalmem }

      writeln(tracefile);

      case MEMTYPE of

        'A': write(tracefile,'DUMP OF MA:');

        'B': write(tracefile,'DUMP OF MB:');

        'R': write(tracefile,'DUMP OF MR:');

      end; { case of MEMTYPE }

      writeln(tracefile,MEMHI:4,' WORDS IN USE.');
      writeln(tracefile);
      for i := 0 to MEMHI-1 do
          write(tracefile,MEM[i]:4);

   end;  { dumplocalmem }


(* Dump processor state and memory. *)
procedure dumprocessor(var PE:processor;TRACE:integer);

   begin

      if TRACE >= 1
         then dumprocstate(PE);

      if TRACE >= 2 then
         with PE do
         begin
            dumplocalmem('A',MA,MAH);
            dumplocalmem('B',MB,MBH);
            dumplocalmem('R',MR,MRH);
         end;

   end; { dumprocessor }

   (* Shift value V into I'th position in processor number P. *)
   procedure shift(P,I:integer; V:dataval);

      begin

         with PE[P] do
              SPORT[I] := V;

      end; { shift }
```

```
(* Fetch the next instruction from the Instruction Stream. *)
procedure Ifetch(var IR:instruction);

   begin
      writeln(output);
      write(output,' IR>  ');
      with IR do
      begin
         readln(codefile,opcode,op1,op2,op3);
         if TRACE >= 0 then
            begin
               writeln(output,opcode:2,op1:3,op2:3,op3:3);
               writeln(tracefile);
               writeln(tracefile,' IR>
',opcode:2,op1:3,op2:3,op3:3);
            end;
      end;
   end; { Ifetch }


(* Execute the current instruction on processor PE. *)
procedure PEexecute(var IR:instruction; var PE:processor);

   var
       i:integer;          { Local work variable. }
       error:boolean;      { Error flag.          }

   (* Move C data values starting at SM[SB] to TM[TB]. *)
   procedure movedata(C:integer;
                      var SM:datamem;SB:integer;
                      var TM:datamem;TB:integer);
      var i:integer;   { Work variable. }

      begin { movedata }

         for i := 0 to C-1 do
             TM[TB+i] := SM[SB+i];

      end;  { movedata }

   (* Move C data values from MS into TM starting at TB. *)
   procedure movefromMS(C:integer; var MS:shiftmem;
                        var TM:datamem; TB:integer);
      var i:integer;

      begin { movefromMS }
         for i := 0 to C-1 do
             TM[TB+i] := MS[i];
      end;  { movefromMS }


   (* Move C data values into MS from SM, starting at SB. *)
   procedure movetoMS(C:integer; var MS:shiftmem;
                      var SM:datamem; SB:integer);
      var i:integer;
```

```
    begin
       for i := 0 to C-1 do
           MS[i] := SM[SB+i];
    end;  { movetoMS }

begin { PEexecute }

    error := false;

    WITH PE do
    BEGIN

    STATUS[1] := false; STATUS[2] := false; STATUS[3] := false;

    case IR.opcode of

      00: { NOOP }
          begin
          end;

      10: { ADBASE m,i }
          begin
             if IR.op1 = 1 then MAB := MAB + IR.op2
             else if IR.op1 = 2 then MBB := MBB + IR.op2
             else if IR.op1 = 3 then MRB := MRB + IR.op2;
          end;

      11: { SETB m,v }
          begin
             if IR.op1 = 1 then MAB := IR.op2
             else if IR.op1 = 2 then MBB := IR.op2
             else if IR.op1 = 3 then MRB := IR.op2;
          end;

      12: { ALLOC m,c }
          begin
             if IR.op1 = 1 then
                MAH := MAH + IR.op2
             else if IR.op1 = 2 then
                MBH := MBH + IR.op2
             else if IR.op1 = 3 then
                MRH := MRH + IR.op2;
          end;

      15: { SETMA v }
          begin
             MAR := IR.op1;
          end;

      16: { ADMA v }
          begin
             MAR := MAR + IR.op1;
          end;

      20: { LOADCM m,c }
```

```
            begin
               if IR.op1 = 1 then
                  for i:= 0 to IR.op2-1 do
                      MA[MAB+i] := CM[MAR+i]

               else if IR.op1 = 2 then
                  for i := 0 to IR.op2-1 do
                      MB[MBB+i] := CM[MAR+i];
            end;

         21: { STORCM c }
            begin
               for i := 0 to IR.op1-1 do
                   CM[MAR+i] := MR[MRB+i];
            end;

         22: { FCSHF c }
            begin
               for i := 0 to IR.op1-1 do
                   shift(FCSID,i,MS[i]);
            end;

         23: { BCSHF c }
            begin
               for i := 0 to IR.op1-1 do
                   shift(BCSID,i,MS[i]);
            end;

         24: { PSSHF c }
            begin
               for i := 0 to IR.op1-1 do
                   shift(PSSID,i,MS[i]);
            end;

         25: { SHFIN c }
            begin
               for i := 0 to IR.op1-1 do
                   MS[i] := SPORT[i];
            end;

         26: { SHIFX c,p}
            begin
            p := IR.op2;
               for i := 0 to IR.op1-1 do
                 shift(p,i,MS[i]);
            end;

         30: { MOVA m,c }
            begin
               if IR.op1 = 2 then movedata(IR.op2,MB,MBB,MA,MAB)
               else if IR.op1 = 3 then
movedata(IR.op2,MR,MRB,MA,MAB)
               else if IR.op1 = 4 then
movefromMS(IR.op2,MS,MA,MAB);
            end;
```

```
          31: { MOVB m,c }
              begin
                 if IR.op1 = 1 then movedata(IR.op2,MA,MAB,MB,MBB)
                 else if IR.op1 = 3 then
movedata(IR.op2,MR,MRB,MB,MBB)
                 else if IR.op1 = 4 then
movefromMS(IR.op2,MS,MB,MBB);
              end;

          32: { MOVR m,c }
              begin
                 if IR.op1 = 1 then movedata(IR.op2,MA,MAB,MR,MRB)
                 else if IR.op1 = 2 then
movedata(IR.op2,MB,MBB,MR,MRB)
                 else if IR.op1 = 4 then
movefromMS(IR.op2,MS,MR,MRB);
              end;

          33: { MOVS m,c }
              begin
                 if IR.op1 = 1 then movetoMS(IR.op2,MS,MA,MAB)
                 else if IR.op1 = 2 then movetoMS(IR.op2,MS,MB,MBB)
                 else if IR.op1 = 3 then
movetoMS(IR.op2,MS,MR,MRB);
              end;

          40: { SADD IA,IB,IR }
              begin
                 MR [MRB+IR.op3] := MA [MAB+IR.op1] + MB
[MBB+IR.op2];
              end;

          41: { SSUB IA,IB,IR }
              begin
                 MR[MRB+IR.op3] := MA[MAB+IR.op1] - MB[MBB+IR.op2];
              end;

          42: { SMPY IA,IB,IR }
              begin
                 MR[MRB+IR.op3] := MA[MAB+IR.op1] * MB[MBB+IR.op2];
              end;

          43: { SDIV IA,IB,IR }
              begin
                 if MB[MBB+IR.op2] <> zeroval
                    then MR[MRB+IR.op3] := MA[MAB+IR.op1]
                                           DIV MB[MBB+IR.op2]
                 else begin
                         error := true; STATUS[1] := true;
                         writeln(output,'ZERO DIVIDE IN
PE-',PROCID:2);
                      end;
              end;

          44: {SMSA        }
```

```
        begin
           if IR.op1 = 1 then
               MR[MRB + IR.op3] := MA[MAB + IR.op2] + MS[i]
           else if IR.op1 = 2 then
               MR[MRB + IR.op3] := MA[MBB + IR.op2] + MS[i]
           else if IR.op1 = 3 then
               MR[MRB + IR.op3] := MA[MRB + IR.op2] + MS[i]
        end;

    50: { VSADD i,c }
        begin
           for i := 0 to IR.op2-1 do
               MR[MRB+i] := MA[MAB+i] + MB[MBB+IR.op1];
        end;

    51: { VSUB i,c }
        begin
           for i := 0 to IR.op2 - 1 do
               MR[MRB+i] := MA[MAB+i] - MB[MBB+IR.op1];
        end;

    52: { VSMPY i,c }
        begin
           for i := 0 to IR.op2 - 1 do
               MR[MRB+i] := MA[MAB+i] * MB[MBB+IR.op1];
        end;

    53: { VSDIV i,c }
        begin
           if MB[MBB+IR.op1] <> zeroval then
              for i := 0 to IR.op2 - 1 do
                  MR[MRB+i] := MA[MAB+i] DIV MB[MBB+IR.op1]
           else begin
                   error := true; STATUS[1] := true;
                   writeln(output,'ZERO DIVIDE IN PE[',
                                  PROCID:1,']');
                end;
        end;

    60: { VPADD c }
        begin
           for i := 0 to IR.op1-1 do
               MR[MRB+i] := MA[MAB+i] + MB[MBB+i];
        end;

    61: { VPSUB c }
        begin
           for i := 0 to IR.op1-1 do
               MR[MRB+i] := MA[MAB+i] - MB[MBB+i];
        end;

    62: { VPMPY c }
        begin
           for i := 0 to IR.op1-1 do
               MR[MRB+i] := MA[MAB+i] * MB[MBB+i];
```

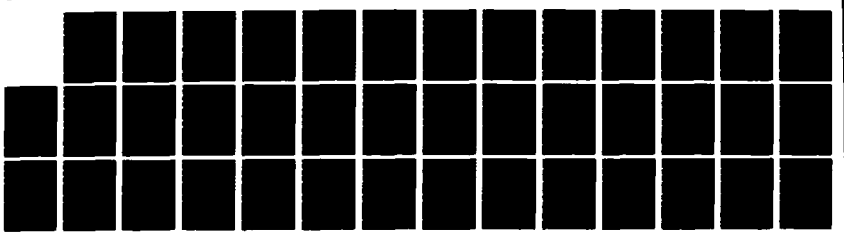
```
            end;

        65: { INPRD k,c}
            begin
               ACC := 0;
               for i := 0 to IR.op2-1 do
                   ACC := ACC + MA[MAB+i] * MB[MBB+i];
               MR[MRB+IR.op1] := ACC;
            end;

        70: { TST }
            begin
               STATUS[0] := STATUS[1] OR STATUS[2] OR STATUS[3];
            end;

      end; { case }

      { If no error occurred, clear the enable bit to indicate }
      {    completion of the instruction execution.            }
      if NOT error then STATUS[0] := false;

      END; { WITH PE }

   end;  { PEexecute }


   (* Read M x N matrix into CM, starting at BASE address. *)
   procedure readmatrix(BASE,M,N:integer;var CM:cpmem);

       var i,j,k:integer;     { Work variables. }

       begin { readmatrix }

          k := BASE;
          for i := 1 to M do
              for j := 1 to N do
              begin
                  read(oprndfile,CM[k]);
                  A[i,j] := CM[k];
                  k := k + 1;
              end;

       end;  { readmatrix }

   procedure Breadmatrix(BASE,M,N:integer;var CM:cpmem);

       var i,j,k:integer;     { Work variables. }

       begin { Breadmatrix }

          k := BASE;
          for i := 1 to M do
              for j := 1 to N do
              begin
```

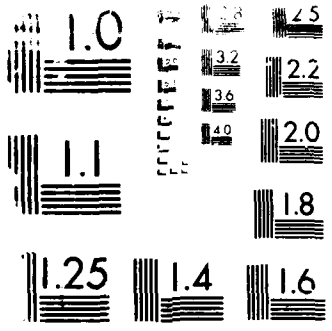
```
                read(oprndfile,CM[k]);
                B[i,j] := CM[k];
                k := k + 1;
            end;

        end;  { Breadmatrix }


    (* Write M x N matrix from CM, starting at BASE address. *)
    procedure writematrix(BASE,M,N:integer;var CM:cpmem);

        var i,j,k:integer;     { Work variables. }

        begin { writematrix }

            writeln(resultfile); writeln(resultfile);
            k := BASE;
            for i := 1 to M do
            begin
                writeln(resultfile);
                for j := 1 to N do
                begin
                    A[i,j] := CM[k];
                    write(resultfile,A[i,j]);
                    k := k + 1;
                end;
            end;
            writeln(resultfile); writeln(resultfile);

        end;  { writematrix }


    (* Convert integer-coded mask into boolean-string mask. *)
    procedure convertnum2mask(NP,NUM:integer; var MASK:PEmask);

        var i:integer;    { Work variable.
}
            q,r:integer; { quotient, remainder for conversion algm.
}
            v: integer;  { Work variable: successive quotients.
}

        begin
            v := NUM;
            for i := NP-1 downto 0 do
            begin
                r := v MOD 2;
                q := v DIV 2;
                if r = 0
                    then MASK.bit[i] := false
                    else MASK.bit[i] := true;
                v := q;
            end;
        end; { convertnum2mask }
```

```
(* Execute CP Instruction.                                    *)
procedure CPExecute(var IR:instruction);

   var i:integer;     { Work variable. }

   begin { CPExecute }

      case IR.opcode of

        100: { ENABLE maskcode }
             begin
                convertnum2mask(NP,IR.op1,ACTIVEMASK);
             end;

        101: { PUSHM }
             begin
                MSTKTOP := MSTKTOP + 1;
                MSTACK[MSTKTOP] := ACTIVEMASK;
             end;

        102: { PULLM }
             begin
                ACTIVEMASK := MSTACK[MSTKTOP];
                MSTKTOP := MSTKTOP-1;
             end;

        103: { SETT tracelevel }
             begin
                TRACE := IR.op1;
             end;

        110: { MREAD base,m,n }
             begin
                readmatrix(IR.op1,IR.op2,IR.op3,CM);
             end;

        113: { MREADB base,m,n }
             begin
                Breadmatrix(IR.op1,IR.op2,IR.op3,CM);
             end;

        111: { MWRITE base,m,n }
             begin
                writematrix(IR.op1,IR.op2,IR.op3,CM);
             end;

        120: { SETV base,count,val }
             begin
                if IR.op1+IR.op2 > CMHI
                   then CMHI := IR.op1 + IR.op2;
                for i := 0 to IR.op2-1 do
                    CM[IR.op1+i] := IR.op3;
             end;

      end; { case }
```

AD-A188 654 AN EMULATION TOOL FOR SIMULATING MATRIX OPERATIONS ON AN SIMD (SINGLE INS (U) NORTH CAROLINA AGRICULTURAL AND TECHNICAL STATE UNIV GREENSBO H L MARTIN OCT 87 2/2

UNCLASSIFIED ARO-22320.2-MA-H DAAG29-84-G-0087 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
3.2
3.6
4.0

```
        end;  { CPExecute }


      (* Perform Emulator initialization. *)
      procedure MJH1Initialize;
    type chrlen = 1..12;
        var
            titleline:packed array[1..72]  { Computation title. }
                                of char;
     filenm:packed array[chrlen] of char;
        begin

            writeln(output,'MJH1 START-UP!!');
     write(output,' ENTER NAME OF OPERAND FILE :');
     readln(input,filenm);
     open(oprndfile,filenm,old);
            write(output,' ENTER NAME OF CODE FILE :');
            readln(input,filenm);
            open(codefile,filenm,old);
     write(output,' ENTER NAME OF RESULT FILE :');
     readln(input,filenm);
     open(resultfile,filenm,new);
     write(output,' ENTER NAME OF TRACE FILE :');
            readln(input,filenm);
     open(tracefile,filenm,new);

            { Open files and copy codefile title line to all output
}
            {       files.
}
            reset(codefile); reset(oprndfile);
            rewrite(tracefile); rewrite(resultfile);
            readln(codefile,titleline);
            writeln(tracefile,titleline); writeln(tracefile);
            writeln(resultfile,titleline); writeln(resultfile);
            writeln(output);
            write(output,' ENTER NUMBER OF PROCESSORS: ');
            readln(input,NP);

            writeln(output);
            writeln(' ENTER PROCESSOR TRACING LEVEL: ');
            writeln('       0) NO TRACE.');
            writeln('       1) PROCESSOR STATE ONLY');
            writeln('       2) FULL PROCESSOR TRACE');
            readln(input,TRACE);
            for i := 0 to NP-1 do
            begin
               initprocessor(PE[i],i,(i+1) mod NP,(NP+i-1) mod NP,
                             (2 * i) mod NP  +  (2 * i) div NP );
               if TRACE > 0
                  then dumprocessor(PE[i],TRACE);
            end;


            { Set up opcodes for PEs and CP. }
```

```
        PEOPCODES := [NOOP,ADBASE,SETB,ALLOC,SETMA,ADMA,LOADCM,
STORCM,FCSHF,BCSHF,PSSHF,SHFIN,MOVA,MOVB,MOVR,
                      MOVS,SADD,SSUB,SMPY,SDIV,
                      VSADD,VSSUB,VSMPY,VSDIV,PVADD,PVSUB,PVMPY,
                      INPRD,TST];

        CPOPCODES := [ENABLE,PUSHM,PULLM,MREAD,MWRITE,SETT,SETV,
                      CPHALT];

        { Initialize active processor mask, and MSTACK. }
        for i := 0 to NP-1 do
            ACTIVEMASK.bit[i] := true;
        NAP := NP;
        MSTKTOP := 0;
        CMHI := 0;

    end; { MJH1Initialize }


  (* Perform MJH1 shut-down. *)
  procedure MJH1ShutDown;

    var i:integer;    { Work variable. }

    begin

        { Dump all processor states and local memories. }
        for i := 0 to NP-1 do
            dumprocessor(PE[i],TRACE+2);

        close(codefile); close(tracefile); close(resultfile);

        writeln(output,'MJH1 SHUT-DOWN!!');

    end; { MJH1ShutDown }


  (* Determine whether all processors completed last PE
instruction. *)
  procedure checkcompletion(NP:integer; var CURMASK:PEmask;
                            var COMPLETED:boolean);

    var i:integer;     { Work variable.    }

    begin
        COMPLETED := true;
        for i := 0 to NP-1 do
            COMPLETED := COMPLETED AND
                         (CURMASK.bit[i] AND NOT
PE[i].STATUS[0]);
    end; { checkcompletion }

  (* Update the active mask based on PE completion codes. *)
  procedure updatemask(NP:integer; var ACTIVEMASK:PEmask);
```

```
        var i:integer;     { work variable.    }

        begin
           for i := 0 to NP-1 do
               if ACTIVEMASK.bit[i]
                  then ACTIVEMASK.bit[i] := NOT PE[i].STATUS[0];
        end; { updatemask }


    (* Enable processors based on mask. *)
    procedure PEEnable(NP:integer; var CURMASK:PEmask;
                       var NAP:integer);

        var i:integer;     { Work variable.  }

        begin
           NAP := 0;
           for i := 0 to NP-1 do
           begin
               PE[i].STATUS[0] := CURMASK.bit[i];
               if CURMASK.bit[i] then NAP := NAP + 1;
           end;
        end; { PEEnable }


    begin { MJH1NEW }

        MJH1Initialize;
        DumpCPState(NP,NAP,TRACE,ACTIVEMASK);

        Ifetch(IR);
        while (IR.opcode <> CPHALT) AND (NAP > 0) do
        begin

           if IR.opcode IN PEOPCODES then
              begin

                 { Execute PE instruction on active processors. }
                 for i := 0 to NP-1 do
                     if ACTIVEMASK.bit[i] then
                        begin
                           PEexecute(IR,PE[i]);
                           if TRACE > 0
                              then dumprocessor(PE[i],TRACE);
                        end;

                 { Check for completion of all PEs before
proceeding. }
                 {   Disable PEs that did not complete.
}
                 {   NOTE: Calls to Error recovery code goes here.
}
                 checkcompletion(NP,ACTIVEMASK,COMPLETED);
                 if NOT COMPLETED
                    then updatemask(NP,ACTIVEMASK);
```

```
            end { PE Instruction}

         else if IR.opcode IN CPOPCODES
              then CPExecute(IR);

         PEEnable(NP,ACTIVEMASK,NAP);

         DumpCPState(NP,NAP,TRACE,ACTIVEMASK);

         Ifetch(IR);

      end; { Ex Cycle. }

      MJH1ShutDown;

end.   { MJH1 }
```

APPENDIX B

CODE FILES FOR THE MJH1

Following are existing code files that are currently operable on the MJH1. The first line of each program gives matrix dimensions, as well as a coded description of the algorithm used.

A listing of some variables follows:

M = # of rows of matrix A

N = # of cols of A and rows of matrix B

P = # of cols of result matrix

T = # of terms of result matrix

The code files are annotated with simple comments that are easily understandable. Careful examination of these files should aid the potential programmer in creating code files of his own.

```
ALG1: 2 x 3 BY 3 x 2; NP=4.  [TEQK]

  0   0   0 0    For r=1 to m
  0   0   0 0    for c=1 to p
  0   0   0 0    ((r-1)p+c)th PE = rth row of [A]
  0   0   0 0    & the cth col of [B]
120   0  20 0    Clear 20 words in CM
110   0   2 3    Read matrix A.
111   0   2 3    Echo A.
110   6   3 2    Read matrix B and echo.
111   6   3 2    Echo B.

  0   0   0 0    Prepare for partitioning.
100  12   0 0    Enable PE's 0 & 1 to load A[1,j]
 15   0   0 0    PEs 0,1 at A[1,1].
 12   1   3 0    Allocate space in MA for 3 values
 20   1   3 0    Move 3 values into MA A[1,j]
100   3   0 0    Enable PEs 2 & 3
 15   3   0 0    Set address to A[2,j]
 12   1   3 0    Allocate space in MA for 3 values
 20   1   3 0    Read A[2,j] into MA
100  10   0 0    Enable PEs 0,2
 15   6   0 0    Both PEs can access B[1,1] (col1)
100   5   0 0    Enable PEs 1,3
 15   7   0 0    Both PEs can access B[1,2] (col2)
100  15   0 0    Enable all PEs
 12   2   3 0    Allocate space for 3 col values of B
 11   2   0 0    Set MB base reg to word zero
 20   2   1 0    Load 1st value from CM into MB
 10   2   1 0    Increment MB by 1
 16   2   0 0    Increment MAR to access next word from CM
 20   2   1 0    Load 2nd value from CM into MB
 10   2   1 0    Increment LM (MB)
 16   2   0 0    Increment MAR to access next word from CM
 20   2   1 0    Load 3rd value from CM into MB
103   3   0 0    Get a TRACE snapshot
103   0   0 0

  0   0   0 0    Prepare to multiply
 11   1   0 0    Set LM base register to 1st word
 11   2   0 0
 11   3   0 0
 12   3   1 0    Allocate room in MR for 4 values
103   3   0 0    Start a TRACE snapshot
 65   0   3 0    Multiply rows by cols
  0   0   0 0    STORE 4 values from each PEs R memory
100   8   0 0    Enable P0
 15  12   0 0    Set MAR to 1st result
100   4   0 0    Enable P1
 15  13   0 0    Set MAR to 2nd result
100   2   0 0    Enable P2
 15  14   0 0    Set MAR to 3rd result
100   1   0 0    Enable P3
 15  15   0 0    Set MAR to 4th result
100  15   0 0    Enable ALL PEs
```

```
 21   1   0 0     Store values
103   0   0 0     END TRACE
111  12   2 2     PRINT
255   0   0 0     CPHALT
```

```
ALG2A:  2 x 3 BY 3 x 4; NP=4.  [PEQK]

  0   0   0 0    Entire matrix [A] and one col of vals
  0   0   0 0    of [B] in order.
120   0  30 0    Clear 30 words in CM
110   0   2 3    Read matrix A.
111   0   2 3    Echo A.
110   6   3 4    Read matrix B and echo.
111   6   3 4    Echo B.

  0   0   0 0    Prepare for Partitioning.
100  15   0 0    Enable PE's 0 & 1 to load A[1,j]
 15   0   0 0    PEs 0,1 at A[1,1].
 12   1   6 0    Allocate space in MA for 6 values
 20   1   6 0    Move 6 values into MA A[i,j]
100   8   0 0    Enable PE 0
 15   6   0 0    Set address to B[1,1]
100   4   0 0    Enable PE 1
 15   7   0 0    Set address to B[1,2]
100   2   0 0    Enable PE 2
 15   8   0 0    Set address to B[1,3]
100   1   0 0    Enable PE 1
 15   9   0 0    Set address to B[1,4]
100  15   0 0    Enable all PEs
103   3   0 0    Start TRACE
 12   2   3 0    Allocate space for 3 col values of B
 20   2   1 0    Load 1st value from CM into MB
 10   2   1 0    Increment MB by 1
 16   4   0 0    Increment MAR to access next word from CM
 20   2   1 0    Load 2nd value from CM into MB
 10   2   1 0    Increment LM (MB)
 16   4   0 0    Increment MAR to access next word from CM
 20   2   1 0    Load 3rd value from CM into MB
 10   2   1 0    Increment MB by 1

  0   0   0 0    Prepare to multiply
 11   1   0 0    Set LM base register to 1st word
 11   2   0 0
 11   3   0 0
 12   3   2 0    Allocate room in MR for 2 values
 65   0   3 0    Multiply rows by cols A[1,j]
100   8   0 0    Enable P0
 15  18   0 0    Set MAR to 1st result
100   4   0 0    Enable P1
 15  19   0 0    Set MAR to 2nd result
100   2   0 0    Enable P2
 15  20   0 0    Set MAR to 3rd result
100   1   0 0    Enable P3
 15  21   0 0    Set MAR to 4th result
100  15   0 0    Enable ALL PEs
 21   1   0 0    Store values
  0   0   0 0    Prepare to continue multiplication
 11   2   0 0    Set LM base register to 3rd word
 11   1   3 0    Set LM base reg A to 1st word
 65   0   3 0    Multiply rows by cols A[2,j]
```

```
100   8   0 0    Enable P0
 15  22   0 0    Set MAR to 5th result
100   4   0 0    Enable P1
 15  23   0 0    Set MAR to 6th result
100   2   0 0    Enable P2
 15  24   0 0    Set MAR to 7th result
100   1   0 0    Enable P1
 15  25   0 0    Set MAR to final result
100  15   0 0    Enable ALL
 21   1   0 0    Send values
111  18   2 4    PRINT
103   0   0 0    STOP TRACE
255   0   0 0    CPHALT
```

```
ALG2B:  2 x 3 BY 3 x 4; NP=2.  [PMULTK]

  0   0   0 0     Entire matrix [A] and p/k cols
  0   0   0 0     of vals of [B] in order
120  30   0 0     Clear 30 words in CM
110   0   2 3     Read matrix A.
111   0   2 3     Echo A.
110   6   3 4     Read matrix B and echo.
111   6   3 4     Echo B.

  0   0   0 0     Prepare for partitioning
100   3   0 0     Enable PE's 0 & 1 to load A[1,j]
 15   0   0 0     PEs 0,1 at A[1,1].
 12   1   6 0     Allocate space in MA for 6 values
 20   1   6 0     Move 6 values into MA A[i,j]
100   2   0 0     Enable PE 0
 15   6   0 0     Set address to B[1,1]
100   1   0 0     Enable PE 1
 15   8   0 0     Set address to B[1,3]
100   3   0 0     Enable both
103   3   0 0     Start TRACE
 12   2   6 0     Allocate space for 6 col values of B
 11   2   0 0     set MB to 0
 20   2   1 0     Load 1st value from CM into MB
 10   2   1 0     Increment MB by 1
 16   4   0 0     Increment MAR to access next word from CM
 20   2   1 0     Load 2nd value from CM into MB
 10   2   1 0     Increment LM (MB)
 16   4   0 0     Increment MAR to access next word from CM
 20   2   1 0     Load 3rd value from CM into MB
 10   2   1 0     Increment MB by 1
100   2   0 0     Enable PE 2
 15   7   0 0     Set MAR to B[1,2]
100   1   0 0     Enable PE 1
 15   9   0 0     Set MAR to B[1,4]
100   3   0 0     Enable ALL
 20   2   1 0     Read 4th val
 16   4   0 0     Increment MAR
 10   2   1 0     Increment MB
 20   2   1 0     Load 5th val
 16   4   0 0     Increment MAR
 10   2   1 0     Increment MB
 20   2   1 0     Load next word

  0   0   0 0     Prepare to multiply
 11   1   0 0     Set LM base register to 1st word
 11   2   0 0
 11   3   0 0
 12   3   4 0     Allocate room in MR for 4 values
 65   0   3 0     Multiply rows by cols A[1,j]
 11   1   0 0     Set LM to row 1
 11   2   3 0     Set LM to col 2
 65   1   3 0
 11   1   3 0     Set to row 2
 11   2   0 0     Set to col 1
```

```
 65   2   3 0
 11   1   3 0      Set to row 2
 11   2   3 0      Set to col 2
 65   3   3 0
100   2   0 0      Enable P0
 15  18   0 0      Set MAR to 1st result
100   1   0 0      Enable P1
 15  20   0 0      Set MAR to 2nd result
100   3   0 0      Enable all
 21   2   0 0      Store values
100   2   0 0
 15  22   0 0      Set to C[2,1]
100   1   0 0
 15  24   0 0      Set to C[2,1]
100   3   0 0
 11   3   2 0      Set to 3rd value in MR
 21   2   0 0
111  18   2 4      PRINT
103   0   0 0      STOP TRACE
255   0   0 0      CPHALT
```

```
ALG3A:  4 x 1 BY 1 x 2; NP=4.  [MEQK]

  0   0   0   0     One row of values of [A] and entire [B]

120  16   0   0     Clear 16 words in CM
110   0   4   1     Read matrix A.
111   0   4   1     Echo A.
110   4   1   2     Read matrix B and echo.
111   4   1   2     Echo B.

  0   0   0   0     Start alocation
100  15   0   0     Enable ALL PE's to load B
 15   4   0   0     PEs at B[1,1].
 12   2   2   0     Allocate space in MB for 2 values
 20   2   2   0     Move 2 values into MB
100   8   0   0     Enable PE 0
 15   0   0   0     Set address to A[1,1]
100   4   0   0     Enable PE 1
 15   1   0   0     Set address to A[2,1]
100   2   0   0     Enable PE 3
 15   2   0   0     A[3,1]
100   1   0   0     Enable PE 4
 15   3   0   0     A[4,1]
100  15   0   0     Enable ALL
103   3   0   0     Start TRACE
 12   1   1   0     Allocate space for A
 20   1   1   0     Load various rows into MA

  0   0   0   0     Prepare to multiply
 11   1   0   0     Set MA to 0
 11   2   0   0     Set MB to 0
 11   3   0   0     Set MR to 0
 12   3   2   0     Allocate room in MR for 2 values
 65   0   1   0     Multiply
 11   1   0   0
 11   2   1   0     Reset B
 65   1   1   0
100   8   0   0     Enable P0
 15   6   0   0     Set MAR to 1st result
100   4   0   0     Enable P1
 15   8   0   0     Set MAR to 2nd result
100   2   0   0     Enable PE 2
 15  10   0   0
100   1   0   0     Enable PE 1
 15  12   0   0     Set MAR to 4th result
100  15   0   0     Enable ALL
 11   3   0   0
 21   1   0   0     Store 1st result
100   8   0   0
 15   7   0   0
100   4   0   0
 15   9   0   0
100   2   0   0
 15  11   0   0
100   1   0   0
```

```
 15 13  0  0
100 15  0  0      Enable ALL
 11  3  1  0
 21  1  0  0      Store other vals of C
111  6  4  2      PRINT
103  0  0  0      STOP TRACE
255  0  0  0      CPHALT
```

```
ALG3B: 4 x 1 BY 1 x 2; NP=2.  [MMULTK]

  0  0  0  0    (m/k) rows of [A] and entire [B]
120 16  0  0    Clear 16 words in CM
110  0  4  1    Read matrix A.
111  0  4  1    Echo A.
110  4  1  2    Read matrix B and echo.
111  4  1  2    Echo B.

  0  0  0  0    Start alocation
100  3  0  0    Enable ALL PE's to load B
 15  4  0  0    PEs at B[1,1].
 12  2  2  0    Allocate space in MB for 2 values
 20  2  2  0    Move 2 values into MB
100  2  0  0    Enable PE 0
 15  0  0  0    Set address to A[1,1]
100  1  0  0    Enable PE 1
 15  2  0  0    Set address to A[3,1]
100  3  0  0    Enable ALL PEs
103  3  0  0    Start TRACE
 12  1  2  0    Allocate space for A
 20  1  2  0    Load various rows into MA
  0  0  0  0    Prepare to multiply
 11  1  0  0    Set MA to 0
 11  2  0  0    Set MB to 0
 11  3  0  0    Set MR to 0
 12  3  4  0    Allocate room in MR for 2 values

 65  0  1  0    Multiply
 11  1  0  0    Reset B
 11  2  1  0
 65  1  1  0
 11  1  1  0
 11  2  0  0
 65  2  1  0
 11  1  1  0
 11  2  1  0
 65  3  1  0
100  2  0  0    Enable P0
 15  6  0  0    Set MAR to 1st result
100  1  0  0    Enable P1
 15 10  0  0    Set MAR to 2nd result
100  3  0  0    Enable ALL PEs
 11  3  0  0
 21  4  0  0    Store 1st results
111  6  4  2    PRINT
103  0  0  0    STOP TRACE
255  0  0  0    CPHALT
```

```
ALG4A: 2x2 by 2x4; NP=4  [TMULTK]

  0   0   0   0    For r=1 to m
  0   0   0   0    { for c = 1 to q  (k=qm)
  0   0   0   0     { ((r-1)q+c)th PE = rth row [A]
  0   0   0   0       and cth string of j cols of [B]
  0   0   0   0       in order}}
120   0  24   0    Allocate space for A,B,R
110   0   2   2    Read A
111   0   2   2    Echo A
110   4   2   4    Read B
111   4   2   4    Echo

  0   0   0   0    Prepare to Partition
 12   1   2   0    Allocate 2 spaces in A
100  12   0   0    Enable PEs 1,2
 15   0   0   0    Set MAR to origin
 20   1   2   0    Read 1st row of A into PEs 1,2
100   3   0   0    Enable PE 3,4
 15   2   0   0    Set MAR to row 2 of A
 20   1   2   0    Read 2nd row of A into PE 3,4
  0   0   0   0    Prepare to load cols
100  10   0   0    Enable PE 1,3
 15   4   0   0    Set MAR to 4 B[1,1]
100   5   0   0    Enable PE 2,4
 15   6   0   0    Set MAR to 6 B[1,3]
100  15   0   0    Enable ALL PEs
 11   2   0   0    Set MB to 0
 12   2   4   0    Allocate 4 values in B
 20   2   1   0    Read 1 value
 10   2   1   0    Increment MB by 1
 16   4   0   0    Increment MAR by 4
 20   2   1   0    Read 1 value
 10   2   1   0    Advance MB
100  10   0   0    Enable PE 1,3
 15   5   0   0    Set MAR to 2nd col
100   5   0   0
 15   7   0   0
100  15   0   0
 20   2   1   0    Read in next value
 10   2   1   0    Increment LM B
 16   4   0   0    Advance MAR to next col value
 20   2   1   0    Read in next value

  0   0   0   0    Perform multiplication
 11   1   0   0
 11   2   0   0    Set LM base register to 1st word
 11   3   0   0
  0   0   0   0    Prepare to multiply
 65   0   2   0    Multiply 1st value
 11   2   2   0    Increment MB to 2nd col
 11   1   0   0    Do A again
 65   1   2   0    Multiply again (2nd val)
  0   0   0   0    Prepare to PRINT result
100   8   0   0    Enable PE 1
```

```
 15 12  0  0    Position for [1,1]
100  4  0  0    Enable PE 2
 15 14  0  0    Position for [1,3]
100  2  0  0    Enable PE 3
 15 16  0  0    Position for [2,1]
100  1  0  0    Enable PE 4
 15 18  0  0    Position for [2,2]
100 15  0  0    Enable ALL PEs
 21  2  0  0    Send values
111 12  2  4    PRINT
255  0  0  0    CPHALT
```

```
ALG4B:  4x2 by 2x2; NP=4    [PSUBK]

  0   0   0   0      for q = 1 to p
  0   0   0   0     { for c = 1 to r
  0   0   0   0       {((q-1)r+c)th PE =
  0   0   0   0        cth string of j rows of [A]
  0   0   0   0        in order and qth col of [B]
120  24   0   0     Allocate space for A,B,C
110   0   4   2      Read A
111   0   4   2     Echo
110   8   2   2      Read B
111   8   2   2     Echo

  0   0   0   0     Partition Matrices
100  10   0   0     Enable PEs 1,3
 15   0   0   0     Set MAR to origin
 12   1   4   0     Allocate 4 values in A
 20   1   4   0     Read 1st 2 rows of A
100   5   0   0      Enable PEs 2,4
 12   1   4   0     Allocate 4 more values
 15   4   0   0     Set MAR up by 4
 20   1   4   0     Read in 2nd 2 rows of A
100  12   0   0      Enable PEs 1,2
 15   8   0   0      Set MAR to B[1,1]
100   3   0   0     Enable PEs 3,4
 15   9   0   0     Set MAR to B[1,2]
100  15   0   0      Enable ALL
 12   2   2   0     Allocate 2 values in B
 20   2   1   0     Read in 1st col value
 16   2   0   0      Increment MAR by 2
 10   2   1   0     Increment MB by 1
 20   2   1   0      Read in 2nd col value

  0   0   0   0     Prepare to multiply
 11   1   0   0
 11   2   0   0     Set ALL LM bases to 0
 11   3   0   0
 12   3   2   0      Al'ote values in MR
 65   0   2   0     Multiply once for 1st row value
 11   1   2   0
 11   2   0   0     Reset MB to top of col
 65   1   2   0      Multiply again for 2nd col value
  0   0   0   0      Prepare to write out result
100   8   0   0      Enable PE 1
 15  12   0   0     Set MAR to 1st result in RESULT MATRIX
100   4   0   0      Enable PE 2
 15  16   0   0     Set MAR
100   2   0   0      Enable PE 3
 15  13   0   0      Set MAR
100   1   0   0     Enable PE 4
 15  17   0   0      Set MAR
100  15   0   0      Enable ALL
 11   3   0   0
 21   1   0   0      Send values
 16   2   0   0     Increment MAR by 2
```

```
 11   3   1   0
 21   1   0   0      Send values
111  12   4   2      Write (PRINT) result
255   0   0   0     CPHALT
```

```
ALG5A:  3x4 by 4x2; NP=4    [TGTKNEQK]

  0   0   0   0     one col of [A] and one row of [B] in order
  0   0   0   0      Load matrices
120  30   0   0     Allocate space in CM
110   0   3   4     Read in A
111   0   3   4      Echo
110  12   4   2      Read B
111  12   4   2     Echo

  0   0   0   0      Partition Matrices
100   8   0   0      Enable PE 1
 15   0   0   0      Set MAR to origin
100   4   0   0     Enable PE 2
 15   1   0   0      Set MAR to A[1,2]
100   2   0   0      Enable PE 3
 15   2   0   0      Set MAR to A[1,3]
100   1   0   0      Enable PE 4
 15   3   0   0      Set MAR to A[1,4]
100  15   0   0      Enable ALL
 12   1   3   0      Allocate 3 spaces in MA
 20   1   1   0     Read in 1st col value of A
 16   4   0   0      Increment MAR
 10   1   1   0      Increment MA
 20   1   1   0      Read 2nd col value of A
 16   4   0   0      Increment MAR
 10   1   1   0     Increment MA
 20   1   1   0      Read 3rd col value of A
100   8   0   0     Enable PE 1
 15  12   0   0      Set MAR to row 1 of B
100   4   0   0      Enable PE 2
 15  14   0   0      Set MAR to row 2 of B
100   2   0   0      Enable PE 3
 15  16   0   0     Set MAR to row 3 of B
100   1   0   0      Enable PE 4
 15  18   0   0     Set MAR to row 4 of B
100  15   0   0      Enable ALL
 12   2   2   0
 20   2   2   0     Read 1 row of B

  0   0   0   0     Prepare to Multiply
 11   1   0   0
 11   2   0   0     Set LM bases to 0
 11   3   0   0
 12   3   6   0      Allocate 6 values in MR
 65   0   1   0      Multiply to find 1st INPRD
 11   1   1   0      Increment MA
 11   2   0   0      Dec MB (back at origin)
 65   1   1   0     Multiply 2nd col value
 11   1   2   0      Increase MA to 3rd value
 11   2   0   0      Keep MB at origin
 65   2   1   0      Multiply 3rd col value of INPRD
 11   1   0   0      Dec MA to origin
 11   2   1   0      Inc MB to 2nd col
 65   3   1   0     Multiply 4th INPRD
```

```
11   1   1   0
11   2   1   0
65   4   1   0      Multiply 5th INPRD
11   1   2   0
11   2   1   0
65   5   1   0      Multiply 6th INPRD
 0   0   0   0      Prepare for INTERPROCESSOR COMMUNICATION
11   1   0   0
11   2   0   0      Reset LMs
11   3   0   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0     Add previous result to MS;Store in MR[0]
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0      Add previous result to MS
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0      Add previous result to MS
 0   0   0   0      Get 2nd FINAL RESULT
11   1   1   0
11   2   1   0      Reset LMs
11   3   1   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0     Add previous result to MS;Store in MR[1]
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0      Add previous result to MS
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0      Add previous result to MS
 0   0   0   0      Get 3rd FINAL RESULT
11   1   2   0
11   2   2   0      Reset LMs
11   3   2   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
```

```
40   0   0   0    Add previous result to MS;Store in MR[1]
30   3   1   0     Move result to MA
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0     Add previous result to MS
30   3   1   0     Move result to MA
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0    Put MS into MB
40   0   0   0     Add previous result to MS
 0   0   0   0     Get 4th FINAL RESULT
11   1   3   0
11   2   3   0     Reset LMs
11   3   3   0
30   3   1   0     Move MR to MA
33   3   1   0     Put one value on SPORT
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0    Add previous result to MS;Store in MR[1]
30   3   1   0     Move result to MA
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0     Add previous result to MS
30   3   1   0     Move result to MA
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0    Put MS into MB
40   0   0   0     Add previous result to MS
 0   0   0   0     Get 5th FINAL RESULT
11   1   4   0
11   2   4   0     Reset LMs
11   3   4   0
30   3   1   0     Move MR to MA
33   3   1   0     Put one value on SPORT
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0    Add previous result to MS;Store in MR[1]
30   3   1   0     Move result to MA
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0     Add previous result to MS
30   3   1   0     Move result to MA
22   1   1   0     Shift forward
25   1   1   0     Receive shifted data
31   4   1   0    Put MS into MB
40   0   0   0     Add previous result to MS
 0   0   0   0     Get 6th FINAL RESULT
11   1   5   0
11   2   5   0     Reset LMs
11   3   5   0
```

```
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
 30   3   1   0     Move result to MA
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0     Add previous result to MS
 30   3   1   0     Move result to MA
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0    Put MS into MB
 40   0   0   0     Add previous result to MS
100   8   0   0     Enable PE 1
 11   3   0   0
 15  20   0   0     Set MAR to 1st result
100   4   0   0     Enable PE 2
 11   3   3   0
 15  21   0   0     Set MAR to 2nd result
100   2   0   0     Enable PE 3
 11   3   1   0
 15  22   0   0     Set MAR to 3rd result
100   1   0   0     Enable PE 4
 11   3   4   0
 15  23   0   0     Set MAR to 4th result
100  15   0   0     Enable ALL
 21   1   0   0     Send 1 value
100  12   0   0     Enable PEs 1,2
 10   3   2   0
 16   4   0   0
 21   1   0   0
111  20   3   2     PRINT
255   0   0   0     CPHALT
```

```
ALG5B:  3x8 by 8x2; NP=4    [TGTKNMULTK]

  0   0   0   0     (n/k) cols of [A] and
  0   0   0   0     (n/k) rows of [B] in order
  0   0   0   0      Load matrices
120  48   0   0      Allocate space for 48 spaces
110   0   3   8      Read A
111   0   3   8      Echo
110  24   8   2      Read B
111  24   8   2      Echo

  0   0   0   0      Partition matrices
100   8   0   0      Enable PE 1
 15   0   0   0      Set MAR to origin
100   4   0   0      Enable PE 2
 15   2   0   0      Set MAR to A[1,3]
100   2   0   0      Enable PE 3
 15   4   0   0     Set MAR to A[1,5]
100   1   0   0     Enable PE 4
 15   6   0   0      Set MAR to A[1,7]
100  15   0   0      Enable ALL
 12   1   6   0      Allocate enough space for 6 values
 20   1   2   0      Read 1st 2 values
 16   8   0   0      Increment MAR
 10   1   2   0      Increment MA
 20   1   2   0      Read 2nd col value
 16   8   0   0      Increment MAR
 10   1   2   0      Increment MA
 20   1   2   0      Read 3rd col value
100   8   0   0      Enable PE 4
 15  24   0   0      Beginning of B[1,1]
100   4   0   0      Enable PE 3
 15  28   0   0
100   2   0   0      Enable 2
 15  32   0   0
100   1   0   0      Enable 1
 15  36   0   0
100  15   0   0      Enable ALL
 12   2   4   0      Allocate 4 values in B
 20   2   1   0      Read in 1 value
 10   2   1   0
 16   2   0   0
 20   2   1   0
100   8   0   0
 15  25   0   0
100   4   0   0
 15  29   0   0
100   2   0   0
 15  33   0   0
100   1   0   0
 15  37   0   0
100  15   0   0
 10   2   1   0
 20   2   1   0
 16   2   0   0
```

```
10   2   1   0
20   2   1   0

 0   0   0   0      Multiply scheme
11   1   0   0
11   2   0   0      Set LM to 0 (origin)
11   3   0   0
12   3   6   0      Allocate space in MR for 6 INPRD
65   0   2   0      Multiply 1st INPRD
11   1   2   0
11   2   0   0
65   1   2   0
11   1   4   0
11   2   0   0
65   2   2   0
11   1   0   0
11   2   2   0
65   3   2   0
11   1   2   0
11   2   2   0
65   4   2   0
11   1   4   0
11   2   2   0
65   5   2   0
 0   0   0   0      INTERPROCESSOR COMMUNICATION
11   1   0   0
11   2   0   0      Reset LMs
11   3   0   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0     Add previous result to MS;Store in MR[0]
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0      Add previous result to MS
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0      Add previous result to MS
 0   0   0   0      Get 2nd FINAL RESULT
11   1   1   0
11   2   1   0      Reset LMs
11   3   1   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0     Add previous result to MS;Store in MR[1]
30   3   1   0      Move result to MA
```

```
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0      Add previous result to MS
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0      Add previous result to MS
 0   0   0   0      Get 3rd FINAL RESULT
11   1   2   0
11   2   2   0      Reset LMs
11   3   2   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0     Add previous result to MS;Store in MR[1]
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0      Add previous result to MS
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0      Add previous result to MS
 0   0   0   0      Get 4th FINAL RESULT
11   1   3   0
11   2   3   0      Reset LMs
11   3   3   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0     Add previous result to MS;Store in MR[1]
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0      Put MS into MB
40   0   0   0      Add previous result to MS
30   3   1   0      Move result to MA
22   1   1   0      Shift forward
25   1   1   0      Receive shifted data
31   4   1   0     Put MS into MB
40   0   0   0      Add previous result to MS
 0   0   0   0      Get 5th FINAL RESULT
11   1   4   0
11   2   4   0      Reset LMs
11   3   4   0
30   3   1   0      Move MR to MA
33   3   1   0      Put one value on SPORT
```

```
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
 30   3   1   0     Move result to MA
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0     Add previous result to MS
 30   3   1   0     Move result to MA
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0    Put MS into MB
 40   0   0   0     Add previous result to MS
  0   0   0   0     Get 6th FINAL RESULT
 11   1   5   0
 11   2   5   0     Reset LMs
 11   3   5   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
 30   3   1   0     Move result to MA
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0     Add previous result to MS
 30   3   1   0     Move result to MA
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0    Put MS into MB
 40   0   0   0     Add previous result to MS
100   8   0   0     Enable PE 1
 11   3   0   0
 15  20   0   0     Set MAR to 1st result
100   4   0   0     Enable PE 2
 11   3   3   0
 15  21   0   0     Set MAR to 2nd result
100   2   0   0     Enable PE 3
 11   3   1   0
 15  22   0   0     Set MAR to 3rd result
100   1   0   0     Enable PE 4
 11   3   4   0
 15  23   0   0     Set MAR to 4th result
100  15   0   0     Enable ALL
 21   1   0   0     Send 1 value
100  12   0   0     Enable PEs 1,2
 10   3   2   0
 16   4   0   0
 21   1   0   0
111  20   3   2     PRINT
255   0   0   0     CPHALT
```

```
ALG5CI: 6X2 by 2X1; NP=4   [TGTKNSUBKAMK]

  0   0   0   0    For r=1 to n
  0   0   0   0    { for l=0 to (k/n-1)
  0   0   0   0      {(ln+r)th PE gets (l+1)th,
  0   0   0   0       (mn/k) vals of col r of [A]
  0   0   0   0       & rth row of values of [B]
  0   0   0   0     Load matrices
120  24   0   0     Allocate 24 spaces
110   0   6   2     Read matrix A
111   0   6   2     Echo
110  12   2   1     Read matrix B
111  12   2   1     Echo B

  0   0   0   0     Partition matrices
100   8   0   0     Enable PE 1
 15   0   0   0     Set MAR to A[1,1]
100   2   0   0     Enable PE 3
 15   6   0   0     Set MAR to A[1,1]
100   4   0   0     Enable PE 2
 15   1   0   0     Set MAR to A[2,1]
100   1   0   0     Enable PE 4
 15   7   0   0     Set MAR to A[2,4]
100  15   0   0     Enable ALL
 12   1   3   0     Allocate 3 values in MA
 20   1   1   0     Read 1st value
 10   1   1   0     Increment MA
 16   2   0   0     Increment MAR by 2
 20   1   1   0     Read 2nd col value
 10   1   1   0    Increment MA
 16   2   0   0     Increment MAR by 2
 20   1   1   0     Read 3rd col val
100  10   0   0     Enable PEs 1,3
 15  12   0   0     Set MAR to B[1,1]
100   5   0   0     Enable PEs 2,4
 15  13   0   0     Set MAR to B[2,1]
100  15   0   0     Enable ALL
 12   2   1   0     Allocate space in MB
 20   2   1   0     Read in 1 row for all PE's

  0   0   0   0     Prepare to multiply
 11   1   0   0
 11   2   0   0     Set ALL LM to 0
 11   3   0   0
 12   3   3   0     Allocate 3 values in MR
 65   0   1   0    1st INPRD
 11   1   1   0
 11   2   0   0
 65   1   1   0     2nd INPRD
 11   1   2   0
 11   2   0   0
 65   2   1   0    3rd INPRD
  0   0   0   0     Prepare for INTERPROCESSOR COMMUNICATION
 11   1   0   0
 11   2   0   0     Reset LMs
```

```
 11   3   0   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[0]
  0   0   0   0     Get 2nd FINAL RESULT
 11   1   1   0
 11   2   1   0     Reset LMs
 11   3   1   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
  0   0   0   0     Get 3rd FINAL RESULT
 11   1   2   0
 11   2   2   0     Reset LMs
 11   3   2   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
  0   0   0   0     Prepare to output
 11   3   0   0     Reset Result memory
100   4   0   0     Enable PE 2
 15  14   0   0     Set result to 1st location
100   1   0   0     Enable PE 4
 15  17   0   0    Set result to 2nd 3 col values
100   5   0   0     Enable 2 PEs 2,4
 21   3   0   0     Read 3 values
111  14   6   1
255   0   0   0     CPHALT
```

ALG5CII: 1X2 by 2X6; NP=4 [TGTKNSUBKBMK]

```
  0  0  0  0     Load matrices
120 24  0  0     Allocate space for matrices
110  0  1  2     Load matrix A
111  0  1  2     Echo
110  2  2  6     Load matrix B
111  2  2  6     Echo

  0  0  0  0     Partition A and B
100 10  0  0     Enable 1,3
 15  0  0  0     A[1,1]
100  5  0  0     Enable 2,4
 15  1  0  0     A[1,2]
100 15  0  0     Enable ALL
 12  1  1  0     Reserve 1 space for A
 20  1  1  0     Read in 1 value
100  8  0  0     Enable 1
 15  2  0  0     B[1,1]
100  2  0  0     Enable 3
 15  5  0  0     B[1,4]
100  4  0  0     Enable 2
 15  8  0  0     B[2,1]
100  1  0  0     Enable 4
 15 11  0  0     B[2,4]
100 15  0  0
 12  2  3  0     Allocate 3 values in B
 20  2  3  0     Read in 3 row vals of [B]

  0  0  0  0     Prepare to multiply
 12  3  3  0     Allocate 3 spaces in MR
 11  1  0  0
 11  2  0  0     Reset local memories
 11  3  0  0
 65  0  1  0     Find 1st INPRD
 11  1  0  0     Reset A
 10  2  1  0     Increment B
 65  1  1  0     Find 2nd INPRD
 11  1  0  0     Reset A
 10  2  1  0     Increment MR
 65  2  1  0     Find 3rd INPRD
  0  0  0  0     Prepare for INTERPROCESSOR COMMUNICATION
 11  1  0  0
 11  2  0  0     Reset LMs
 11  3  0  0
 30  3  1  0     Move MR to MA
 33  3  1  0     Put one value on SPORT
 22  1  1  0     Shift forward
 25  1  1  0     Receive shifted data
 31  4  1  0     Put MS into MB
 40  0  0  0    Add previous result to MS;Store in MR[0]
  0  0  0  0     Get 2nd FINAL RESULT
 11  1  1  0
 11  2  1  0     Reset LMs
 11  3  1  0
```

```
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
  0   0   0   0     Get 3rd FINAL RESULT
 11   1   2   0
 11   2   2   0     Reset LMs
 11   3   2   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
  0   0   0   0     Prepare to output
 11   3   0   0     Reset Result memory
100   4   0   0     Enable PE 2
 15  14   0   0     Set result to 1st location
100   1   0   0     Enable PE 4
 15  17   0   0    Set result to 2nd 3 col values
100   5   0   0     Enable 2 PEs 2,4
 21   3   0   0     Read 3 values
111  14   1   6
255   0   0   0     CPHALT
```

ALG5DI: 3X2 by 2X3; NP=4 [TGTKNSUBKBMGTP]

SO FAR, THIS WON'T WORK FOR THIS CASE. THERE SEEMS TO BE SOMETHING WRONG WITH THE ALGORITHM. IT IS NOT ALLOWING FOR ROW 2 OF MATRIX [A] TO BE ALLOCATED TO ANY PE.

```
ALG5DII: 3x2 by 2x5; NP = 4   [TGTKNSUBKMLTP]

  0   0   0   0     For c=1 to n
  0   0   0   0     { for l=0 to (k/n-1)
  0   0   0   0      { (ln+c)th PE gets cth col of [A] and
  0   0   0   0        (l+1)th string of (in/k) vals of row c
  0   0   0   0        of [B]
  0   0   0   0     ADD'L VALS:
  0   0   0   0     For c=(i+1) to p
  0   0   0   0     { for r=1 to n
  0   0   0   0      { ((c-(i+1))n+r)th PE gets rth row value of
  0   0   0   0        cth col of [B]
120  24   0   0      allocate space for matrices
110   0   3   2      Read [A]
111   0   3   2
110   6   2   5      Read [B]
111   6   2   5

  0   0   0   0      Prepare to Partition
100  10   0   0      Enable PE 1,3
 15   0   0   0      Set MAR to A[1,1]
100   5   0   0      Enable PE 2,4
 15   1   0   0      Set MAR A[1,2]
100  15   0   0      Enable ALL
 12   1   3   0      Allocate 3 values in MA
 20   1   1   0      Read 1 value
 11   1   1   0      Set LM A to 1
 16   2   0   0      Increment MAR by 2
 20   1   1   0      Read 1 more val
 11   1   2   0      Set LM A to 2
 16   2   0   0      Increment MAR by 2
 20   1   1   0      Read final A value
100   8   0   0      Enable 1
 15   6   0   0     B[1,1]
100   2   0   0      Enable 3
 15   8   0   0      B[1,3]
100   4   0   0      Enable 2
 15  11   0   0      B[2,1]
100   1   0   0      Enable 4
 15  13   0   0      B[2,3]
100  15   0   0      Enable ALL
 12   2   3   0      Allocate space for 3 values in MB
 20   2   2   0      Read in 2 row values
100   8   0   0      Enable PE 1
 15  10   0   0      Set MAR B[1,5]
100   4   0   0     Enable PE 2
 15  15   0   0      Set MAR B[2,5]
100  12   0   0
 10   2   2   0
 20   2   1   0

  0   0   0   0      Prepare to multiply
100  15   0   0      Enable ALL
 12   3   9   0      Allocate space for 9 values
 11   1   0   0
```

```
 11   2   0   0     Set LM's to 0
 11   3   0   0
 65   0   1   0     Multiply 1st INPRD
 10   1   1   0     Increment MA
 11   2   0   0     Reset B
 65   1   1   0     Multiply 2nd INPRD
 10   1   1   0     Increment MA
 11   2   0   0     Reset B
 65   2   1   0     Multiply again
 11   1   0   0     Reset A for 2nd col
 11   2   1   0    Reset B for 2nd col
 65   3   1   0
 10   1   1   0     Increment MA
 11   2   1   0     Reset B
 65   4   1   0     Multiply 2nd value
 10   1   1   0     Increment MA
 11   2   1   0     Reset B
 65   5   1   0     Last INPRD for regular allocation
  0   0   0   0     Now for additional values
100  12   0   0
 11   1   0   0     Reset A
 11   2   2   0     Set B
 65   6   1   0     Multiply
 11   1   0   0     Reset A
 11   2   3   0    Set [B]
 65   7   1   0     Last INPRD for additional values

  0   0   0   0     Prepare for INTEPROCESSOR COMMUNICATION
100  15   0   0     Enable ALL
 11   1   0   0
 11   2   0   0     Reset LMs
 11   3   0   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[0]
  0   0   0   0     Get 2nd FINAL RESULT
 11   1   1   0
 11   2   1   0     Reset LMs
 11   3   1   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
 22   1   1   0     Shift forward
 25   1   1   0     Receive shifted data
 31   4   1   0     Put MS into MB
 40   0   0   0    Add previous result to MS;Store in MR[1]
100  12   0   0
  0   0   0   0     Get 3rd FINAL RESULT
 11   1   2   0
 11   2   2   0     Reset LMs
 11   3   2   0
 30   3   1   0     Move MR to MA
 33   3   1   0     Put one value on SPORT
```

```
 22   1   1   0      Shift forward
 25   1   1   0      Receive shifted data
 31   4   1   0      Put MS into MB
 40   0   0   0     Add previous result to MS;Store in MR[1]
  0   0   0   0      Get 4th FINAL RESULT
 11   1   3   0
 11   2   3   0      Reset LMs
 11   3   3   0
 30   3   1   0      Move MR to MA
 33   3   1   0      Put one value on SPORT
 22   1   1   0      Shift forward
 25   1   1   0      Receive shifted data
 31   4   1   0      Put MS into MB
 40   0   0   0     Add previous result to MS;Store in MR[1]
  0   0   0   0      Get 4th FINAL RESULT
 11   1   4   0
 11   2   4   0      Reset LMs
 11   3   4   0
 30   3   1   0      Move MR to MA
 33   3   1   0      Put one value on SPORT
 22   1   1   0      Shift forward
 25   1   1   0      Receive shifted data
 31   4   1   0      Put MS into MB
 40   0   0   0     Add previous result to MS;Store in MR[1]
  0   0   0   0      Get 5th FINAL RESULT
 11   1   2   0
 11   2   2   0      Reset LMs
 11   3   2   0
 30   3   1   0      Move MR to MA
 33   3   1   0      Put one value on SPORT
 22   1   1   0      Shift forward
 25   1   1   0      Receive shifted data
 31   4   1   0      Put MS into MB
 40   0   0   0     Add previous result to MS;Store in MR[1]
  0   0   0   0      Prepare to output
 11   3   0   0      Reset Result memory
100   4   0   0      Enable PE 2
 15  16   0   0      Set result to 1st location
100   1   0   0      Enable PE 4
 15  17   0   0     Set result to 2nd 3 col values
100   2   0   0      Enable 3
 15  18   0   0
100   1   0   0
 15  19   0   0
100  15   0   0
 21   1   0   0      Read 1 value
 10   3   1   0
 16   5   0   0
 21   1   0   0
 10   3   1   0
 16   5   0   0
 21   1   0   0
100   4   0   0
 10   3   1   0
 15  20   0   0
```

```
 21   1   0   0
 10   3   1   0
 16   1   0   0
 21   1   0   0
 10   3   1   0
 16   1   0   0
 21   1   0   0
111  14   3   5
255   0   0   0        CPHALT
```

```
ALG5E: 2X3 by 3X3; NP=4   [TGTKMSUBK]

  0    0    0    0      For s=1 to i
  0    0    0    0      { sth string of m PEs get
  0    0    0    0       m rows in order of [A] and,
  0    0    0    0       for c=0 to j
  0    0    0    0       { (ck/m+s)th col of vals of [B]
  0    0    0    0      ADD'L VALS:
  0    0    0    0      for s=(i+1) to k/m (if s>p)
  0    0    0    0      { sth string of m processors get m rows
  0    0    0    0       of [A] in order and,
  0    0    0    0       for c=0 to (j-1)
  0    0    0    0       {(ck/m+s)th col of vals of [B]
  0    0    0    0      Load matrices
120   24    0    0      Allocate 24 values
110    0    2    3      Read A
111    0    2    3      Echo
110    6    3    3      Read B
111    6    3    3      Echo

  0    0    0    0      Prepare to partition A and B
100   15    0    0      Enable ALL
 15    0    0    0      A[1,1]
 12    1    6    0      Allocate 6 values
 20    1    6    0      Read in A
100   12    0    0      Enable PEs 1,2
 15    6    0    0      B[1,1]
100    3    0    0      Enable PEs 3,4
 15    8    0    0      B[1,3]
100   15    0    0      Enable ALL
 11    2    0    0
 12    2    6    0      Reserve 6 spaces in B
 20    2    1    0      Read 1st col value
 16    3    0    0      Increment to next value
 10    2    1    0      Increment MB
 20    2    1    0      Read 2nd val
 16    3    0    0      Increment MAR
 10    2    1    0      Increment MB
 20    2    1    0      Read another value
 10    2    1    0      Increment MB
100    3    0    0      Enable PEs 3,4
 15    7    0    0      B[1,2]
 20    2    1    0      Read in 2nd col of B
 16    3    0    0      Increment MAR
 10    2    1    0      Increment MB
 20    2    1    0      Read val
 16    3    0    0      Increment MAR
 10    2    1    0      Increment MB
 20    2    1    0      Read final value

  0    0    0    0      Prepare to multiply
100   15    0    0      Enable ALL
 11    1    0    0
 11    2    0    0      Set ALL LM to 0
 11    3    0    0
```

```
 65    0    3    0       Find 1st INPRD
 11    1    3    0       Advance to row 2
 11    2    0    0       Reset B
 65    1    3    0       Find 2nd INPRD
 15    3    0    0       Prepare for left overs
 11    1    0    0
 11    2    3    0
 65    2    3    0       1st of add'l values
 11    1    3    0
 11    2    3    0
 65    3    3    0       2nd of add'l values
  0    0    0    0       Prepare to write out values
100    8    0    0       Enable PE 1
 15   15    0    0       C[1,1]
100    2    0    0       Enable PE 3
 15   17    0    0       C[1,3]
100   10    0    0       Enable PEs 1,3
 11    3    0    0
 21    1    0    0       Send 1st col val
 10    3    1    0
 16    3    0    0
 21    1    0    0       Send 2nd col val
100    2    0    0       Enable PE 3 only
 15   16    0    0
 10    3    1    0       Advance MR
 21    1    0    0
 16    3    0    0       Increment to correct location
 10    3    1    0       Advance MR
 21    1    0    0
111   15    2    3
255    0    0    0
```

```
ALG5F: 3x2 by 2x2; NP=4   [TGTKPSUBK]

  0   0   0   0    For s=1 to i
  0   0   0   0    { sth string of p processors get
  0   0   0   0     p cols of [B] in order, and
  0   0   0   0     for r=0 to j
  0   0   0   0     { (rk/p+s)th row of vals of [A]
  0   0   0   0    ADD'L VALS:
  0   0   0   0    for s=(i+1) to k/p (if s<m)
  0   0   0   0    { sth string of p processors is allocated
  0   0   0   0     p cols in order of [B] and,
  0   0   0   0     for r=0 to (j-1)
  0   0   0   0     {(rk/p+s)th row of vals of [A]
  0   0   0   0     Prepare to load matrices
120  20   0   0     Allocate space for 20 values
110   0   3   2     Read [A]
111   0   3   2    Echo
110   6   2   2     Read [B]
111   6   2   2     Echo

  0   0   0   0     Partition matrices
100  12   0   0     Enable 1,2
 15   0   0   0     A[1,1]
100   3   0   0    Enable 3,4
 15   2   0   0    A[2,1]
100  15   0   0    Enable ALL
 11   1   0   0     Set MA to zero
 12   1   4   0     Allocate space for 4 values in MA
 20   1   2   0     Read 1st row
100  12   0   0    Activate only 1,2
 16   4   0   0     Increment MAR
 10   1   2   0     Increment LM
 20   1   2   0     Read 2nd row
100  15   0   0     Enable All
 15   6   0   0     B[1,1]
 12   2   4   0     Reserve space for 4 words
 20   2   4   0     Read B

  0   0   0   0    Prepare to multiply
 12   3   4   0     Allocate values in MR
 11   1   0   0
 11   2   0   0     Set LM to 0
 11   3   0   0
 65   0   2   0     Multiply to find INPRD
 11   1   0   0     Reset A
 11   2   2   0     Set MB to second column
 65   1   2   0     Multiply for 2nd INPRD
 11   1   2   0
 11   2   0   0
 65   2   2   0
 11   1   2   0
 11   2   2   0
 65   3   2   0
  0   0   0   0     Prepare to PRINT
 11   3   0   0
```

```
100   8   0   0
 15  10   0   0      C[1,1]
100   2   0   0
 15  12   0   0      C[2,1]
100   4   0   0
 11   3   2   0
 15  14   0   0
100  15   0   0
 21   2   0   0      Send 2 values over
  0   0   0   0      PRINT
111  10   3   2     Print Result
255   0   0   0      CPHalt
```

END
DATE
FILMD
3-88
DTIC