CPHD 3

A NODE INTERFACE

FOR

PARALLEL PROCESSING

M.R.HAMMES BSc.

Ph.D Thesis

This thesis is submitted to the Council for National Academic Awards in partial fulfilment of the requirements for degree of Doctor of Philosophy.

Trent Polytechnic Nottingham,

Department of Computing in collaboration with

the Department of Electrical and Electronic Engineering.

July 1989

ACKNOWLEDGMENTS

The Author would like to thank Dr. D. Al-Dabass for his guidance and support through the duration of the research work. I would also like to thank the Department of Computing at Trent Polytechnic for its support of the project and the Department of Electrical and Electronic Engineering for the use of its facilities.

Finally, I would like to dedicate this work to my wife, Janet, who has been very patient with me through the course of my research.

A NODE INTERFACE

FOR

PARALLEL PROCESSING

M.R.HAMMES BSc.

ABSTRACT:

A multiprocessor communication scheme for large parallel systems is devised to offer total interconnectivity which is software programmable. The system combines associated addressing with data broadcasting along a global bus to virtually eliminate communications overheads in iterative, or cyclic, applications.

An overall introduction to parallel processing machines is given in chapter 1 and leads to an extended treatment in chapter 2 of the limitations inherent in a range of communication techniques currently employed in multiprocessor systems for parallel processing. A new communication scheme can only be devised successfully after fully considering the concepts that underlie the range of target applications. Performance models are then formulated to assess the effectiveness of the new scheme compared to a conventional common memory system. Only then will the detailed specification of the design be made. The result is a MUlti Processor Interface (MUPI) to be integrated on a single semi-custom silicon device to perform all interprocessor communications for a Multi Interfaced-Node Net for Iterative Environments (MINNIE). Chapter 3 deals with this process of devising the new scheme. Implementation details of turning the design into working hardware is carried out in two stages: a 4-node prototype system using the XILINX programmable device, followed by a full 64 node system using an MCE gate array. Verification of the effectiveness of the proposed system is dealt with in chapter 5 where, based upon the extensive use of the system, a new performance model is formulated to include basic hardware parameters. Numerical examples are given to explore the performance as the speed of the bus and allocator are increased. Summing up and suggestions for future work are given in chapter 6.

CONTENTS

Figures and Tables

Tables Page

CHAPTER 1

INTRODUCTION

CHAPTER 1

INTRODUCTION

The need to solve very large problems at moderate speed on the one hand, and relatively smaller problems at very high speed on the other, has lead to the development of special purpose parallel processing computers.

Parallelism in computing systems is simply executing more than one task at a time, and clearly there is no limit in principle to the number of concurrent actions. So, potentially, parallelism offers an arbitrary degree of improvement in computing speed.

There are two distinct phases of operation within a parallel machine, Al-Dabass [1977], that need to be considered when analysing various architectures. There is an overhead phase and a computation phase. The overhead phase includes the initialisation of the machine, the cross communication of data among processors as well as the data collection at the end of the computation phase. When evaluating the effectiveness of an architecture both of these phases need to be considered in order to obtain a full picture of the system under investigation.

When a program is divided into parallel parts and run on a number of machines, it is clear that if only the computation phase of each machine were compared then the advantage of one architecture over another would not be completely determined. The overhead phase is the more important of the two as it is this phase which ultimately determines the minimum achievable parallel run time, Al-Dabass [1976a]. Hence in the design of parallel architectures not only must we consider the number of possible processors but also the communications structure among them.

It is important to recognise that many levels exits where parallelism may be employed, Zakharov [1984]. The main categories are:

(1) *Within Functional Units*:

Arithmetic, logical, and other operations can be implemented in bit-serial mode, parallel by bit groups, or as whole operands concurrently. This category does not generally affect the way a task/problem is formulated, although it does determine the speed of execution.

Concurrent access to several interleaved memory units also fall into this category, Hellerman [1967].

(2) *Within Processing Elements*:

The most obvious form of concurrency at this level is between different operands being executed in parallel by different operations.

Another kind of concurrency here is where there is only a single instruction at one time, but the functional unit to which the instruction refers may be able to process a stream of operands overlapped in a pipelined fashion, Hockney and Jessope [1981].

(3) *Within Uniprocessing Computers*:

With only a single processor there are many activities which can proceed concurrently. Obvious examples are memory access and I/O.

(4) *Many Processor Systems*:

An obvious category of concurrency is where a computer system contains several processors, either sharing main memory or not, and intercommunicating by some means.

1.1 Parallel Computer Structures

Having shown that there are many levels within a system where parallelism can be utilised this is an appropriate point to introduce parallel architectures which utilise these various levels.

As their name implies, parallel computers are those systems that emphasis parallel processing, and can be divided into the following three architectural configurations, Hwang and Briggs [1985]:

(a) Pipeline Computers
(b) Array Processors
(c) Multiprocessor Systems

A pipeline computer performs overlapped computations to exploit temporal parallelism. An array processor uses multiple synchronous arithmetic logic units to achieve spatial parallelism. A multiprocessor system achieves asynchronous parallelism through a set of interactive processors with shared resources. These three approaches are not mutually exclusive, most existing computers are now pipelined, and some also assume an "array" or a "multiprocessor" structure. The fundamental difference between an array processor and a multiprocessor system is that the processing elements in an array processor operate synchronously but processors in a multiprocessor system operate asynchronously.

Other computing concepts to be introduced in this chapter include data flow computers and some VLSI algorithmic processors. All these new approaches demand extensive hardware to achieve parallelism. The rapid progress in the VLSI technology has made these new approaches possible.

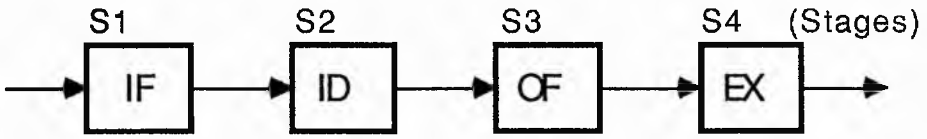
1.1.1 Pipeline computers

The basic idea in a pipeline processor is to introduce some simultaneity of processing by breaking a complex, time-consuming function into a series of simpler, faster operations. For example, floating-point operations involve exponent handling and shifts of the fractions as well as an arithmetic operation.

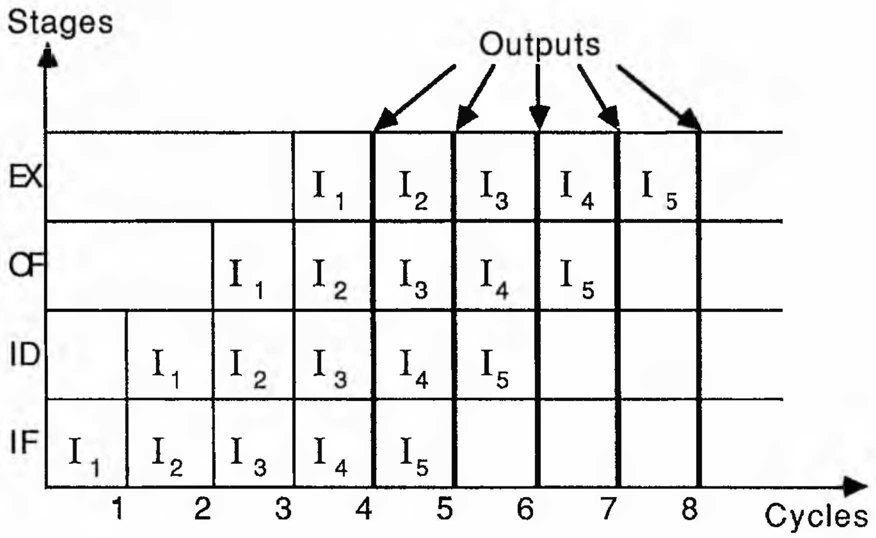
The pipeline concept involves the division of an instruction into a number of distinct stages, each of which can be allocated to a separate processing unit. Some of the stages into which an instruction can be divided are:

(a) instruction fetch (IF);
(b) instruction decoding (ID);
(c) identifying the operation to be performed;
(d) operand fetch (OF);
(e) execution (EX); and
(f) returning the result to store.

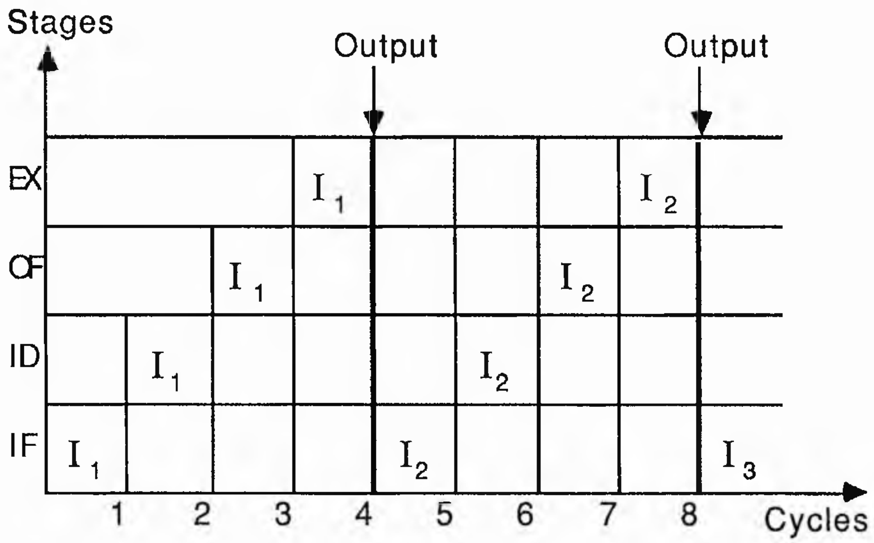
In a nonpipelined processor, these operations must be completed before the next instruction can be issued. In a pipelined computer, successive instructions are executed in an overlapped fashion, as in figure 1.1. Four pipelined stages, IF, ID, OF, and EX, are arranged into a linear cascade. The two space-time diagrams show the difference between overlapped instruction execution and sequentially non overlapped execution.



(a) A pipelined processor.



(b) Space-Time diagram for a pipelined processor.



(c) Space-time diagram for a nonpipelined processor

Figure 1.1 Basic concepts of a pipelined processor

1.1.2 Array computers

An array processor is a synchronous parallel computer with multiple arithmetic logic units (ALUs), called processing elements (PEs), that can operate in parallel in a lock step fashion. By replication of ALUs, one can achieve spatial parallelism. The PEs are synchronised to perform the same function at the same time.

The PEs are usually organised as an array of processors connected in a nearest neighbour mesh network, i.e. each PE has a direct link with its nearest neighbours.

A typical configuration contains a set of n processors, n memory modules, an interconnection network and a control unit. The control unit sends the instructions to the PEs. There is a facility to restrict the number of PEs that would execute an instruction, hence PEs can be masked at any time. Each enabled PE will execute the instruction using data obtained from its private memory module. Interprocess communication takes place via the interconnection network.

The motivation for using a processor array may be illustrated by considering the addition of two n*n matrices. Here, the n^2 additions could be performed completely in parallel in a single addition time, were there available n^2 adders. Another example might be in pattern recognition systems were each picture element (pixel) of an image can be represented by an element in an array of processors then a transformation of the picture can occur in one step rather then a step for each individual pixel.

1.1.3 Multiprocessor systems

Multiprocessor systems categorised as parallel processing systems are collections of independent processors that work together. They execute different but related programmes.

A basic multiprocessor system would contain two or more processors of approximately comparable capabilities. All processors would have access to a selection of common/global memory modules, I/O channels and peripheral devices. Most importantly. the entire system must be controlled by a single integrated operating system providing interactions between processors and their programmes at various levels. Besides the shared memories and I/O devices, each processor has its own local memory and private devices. Interprocessor communications can be done through the shared memories or through an interrupt network.

Multiprocessor architectures are determined primarily by the interconnection structure to be used between the memories and the processors to facilitate interprocessor communications.

1.1.4 Data Flow computers

Conventional von Neumann machines are termed *control flow computers* because instructions are executed sequentially as controlled by a program counter. Sequential program execution is inherently slow. To exploit maximal parallelism in a program, *data flow computers* were suggested in recent years, Dennis [1980]. The basic concept is to enable the execution of an instruction whenever its required operands become available. Thus no program counters are needed in data-driven computations.

Programmes for data-driven computations can be represented by data flow graphs. An example data dependence graph is given in figure 1.2 for the calculation of the following expression: z = (x + y) * 2

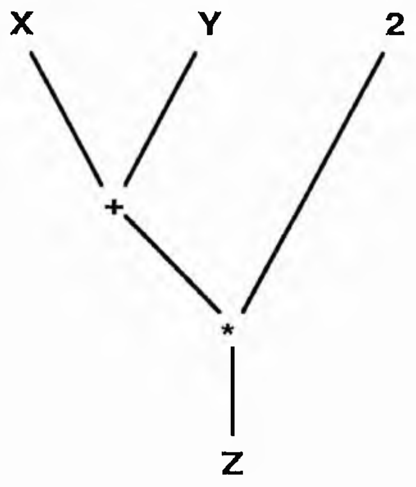


Figure 1.2 Data Flow Graph

A machine-level data flow program can be generated by constructing a data flow graph for a conventional high- level program, Allen and Cocke [1976], Kuck et al. [1981]. Note that the data flow graph shows how instructions are dependent on data. It is meaningless to execute an instruction before all its required data is available.

Conversely, once an instruction has finished executing, all other instructions that are waiting for its output can be safely activated. Execution of the program graph is thus data driven.

1.1.5 VLSI computing structures

The rapid advent of very-large-scale integration (VLSI) technology has created a new architectural horizon in implementing parallel algorithms in hardware Fairbairn [1982].

Highly parallel computing structures promise to be a major application area for the million-transistor chips that will be possible in just a few years. Such computing systems have structural properties that are suitable for VLSI implementation. Almost by definition, parallel structures imply a basic computational element repeated perhaps hundreds or thousands of times. This architectural style immediately reduces the design problem by similar orders of magnitude.

Cost effectiveness has always been a major concern in designing special purpose VLSI systems; their cost must be low enough to justify their limited applicability. Special-purpose design costs can be reduced by the use of appropriate architectures which have a repeating structure.

The use of VLSI technology in designing high- performance multiprocessors and pipelined computing devices is currently under intensive investigation in both industrial and educational environments.

1.2 FLYNN'S Classifications

In 1966 Flynn classified very high-speed computers into four broad categories which have been widely accepted. Digital computers are classified according to the multiplicity of instruction and data streams Flynn [1966]. The essential computing process is the execution of a sequence of instructions on a set of data. An *Instruction Stream* is a sequence of instructions as performed by a machine; a *Data Stream* is the sequence of data called for by the instruction stream.

Computer organisations are classified by the multiplicity of the hardware provided to service the instruction and data streams. Flynn's four machine classifications are:

SISD	:	Single instruction-single data streams
SIMD	:	Single instruction-multiple data streams
MISD	:	Multiple instruction-single data streams
MIMD	:	Multiple instruction-multiple data streams

This classification is elegant in its simplicity and symmetry, and naturally encompasses any machine organisation which can be described as executing instructions which operate on data.

STRUCTURE	CLASSIFICATION
Pipeline Computers	MISD
Array Processors	SIMD
Multiprocessor Systems	MIMD

Table 1.1 Classifications of parallel structures

With these classifications the three main structures introduced in section 1.1 are classified as shown in table 1.1. The pipeline structure is a series of instructions which act on a single piece of data but, these operations can be overlapped which gives the impression of an MIMD classification. The category of array processors are classified as SIMD as the structure is such that an array of data is manipulated by single instructions. Finally, the multiprocessor systems introduced in section1.1.3 can be classified as MIMD as they are formed by a collection of independent processing elements which work together.

CHAPTER 2

CURRENT INTERPROCESSOR COMMUNICATION SYSTEMS

CHAPTER 2

CURRENT INTERPROCESSOR COMMUNICATION SYSTEMS

In this chapter particular attention is focused on two of Flynn's classifications; SIMD and MIMD, as most of the systems to date fall into these categories. By comparison, pipelining falls into the MISD category, and data flow machines, not being of a von Neumann architecture, do not fall into any of Flynn's classifications. There is a section concerned with the Transputer, which is seen to be a major development in the popularisation of parallel processing concepts. Interconnection strategies and methods of performance analysis are introduced as they form vital areas of research in determining the appropriate architecture.

2.1 SIMD - Array Processors

The idea of regular arrays of processors appears to go back to Unger in 1958, Unger [1958]. The primary reason for this proposal was the problem of pattern recognition.

The first machine actually designed with an array type architecture was the Solomon computer, along the lines proposed by Slotnick in 1962, Slotnick et al. [1962]. The machine was to be made up from a 32x32 array of intercoupled processors, each operating in a bit serial mode with 4096 words of local storage. All of the PEs could communicate with their four nearest neighbours leaving the PE's on the edge of the array for I/O.

The main use of the Solomon's design was to be in the solution of linear systems of equations and in the numerical solution of partial differential equations, principally by finite differences. However, due to the choice of bit-serial operations and the technology available a complete machine was never built. A much more powerful

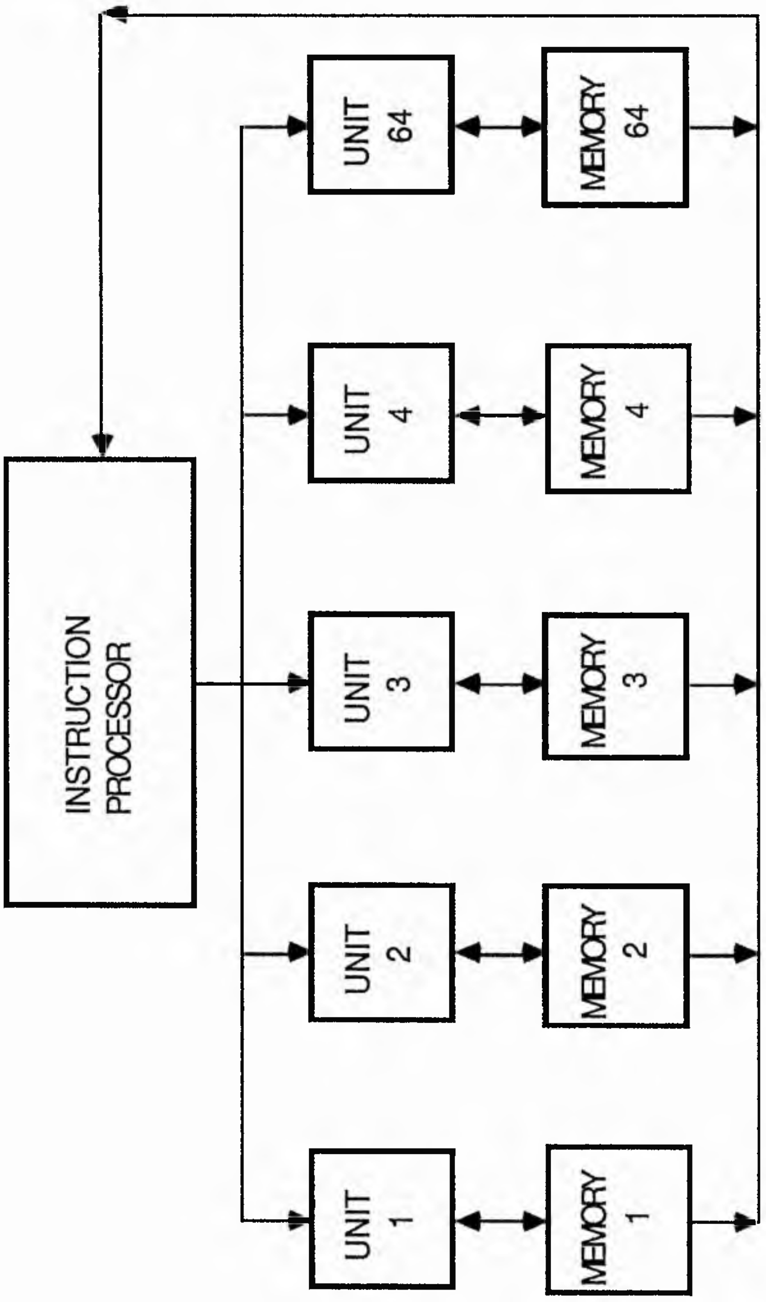


Figure 2.1 Block Diagram of ILLIAC IV.

machine based on the Solomon idea, the Illiac IV, Barnes et al. [1968], was proposed, however, and this overcame many of the principle difficulties. Nevertheless, the Illiac IV did not appear until about ten years later!

The Illiac IV system, figure 2.1, was developed at the University of Illinois in the 1960s and fabricated by the Burroughs Corporation in 1972. The system was to employ 256 PEs under the supervision of four CUs (control units). Due to cost escalation and schedule delays, the system was ultimately limited to one quadrant with 64 PEs and one CU.

The Illiac IV had its drawbacks and these were mainly due to the basic structure of memory in relation to the processors, with each PE having direct access only to its corresponding 2048-word memory. To solve this problem another machine was built by Burroughs, the BSP (Burroughs Scientific Processor), Jenson [1978], which would be an outcome of the Illiac IV experience but free from many of the difficulties, see figure 2.2. The main principle difference was simply that all the parallel PE's are coupled to main memory not in a set of corresponding banks, but through an alignment network.

The alignment network is such that all the PE's, in concurrent operation, can access any part of central memory for input operands or placement of results. Because alignment networks get rapidly more complex as the number of connections increases, the number of PE's was limited to 16.

The BSP is not a stand-alone computer. It is a back end processor attached to a host machine, a system manager. The motivation for attaching the BSP to a system manager is to free it from routine management and I/O functions in order to concentrate on arithmetic computations.

Two other machines need to be mentioned here, however, since they too are based on the original Unger idea, and both have been built. They are the DAP ,Reddaway [1973], and the MPP, Batcher [1980].

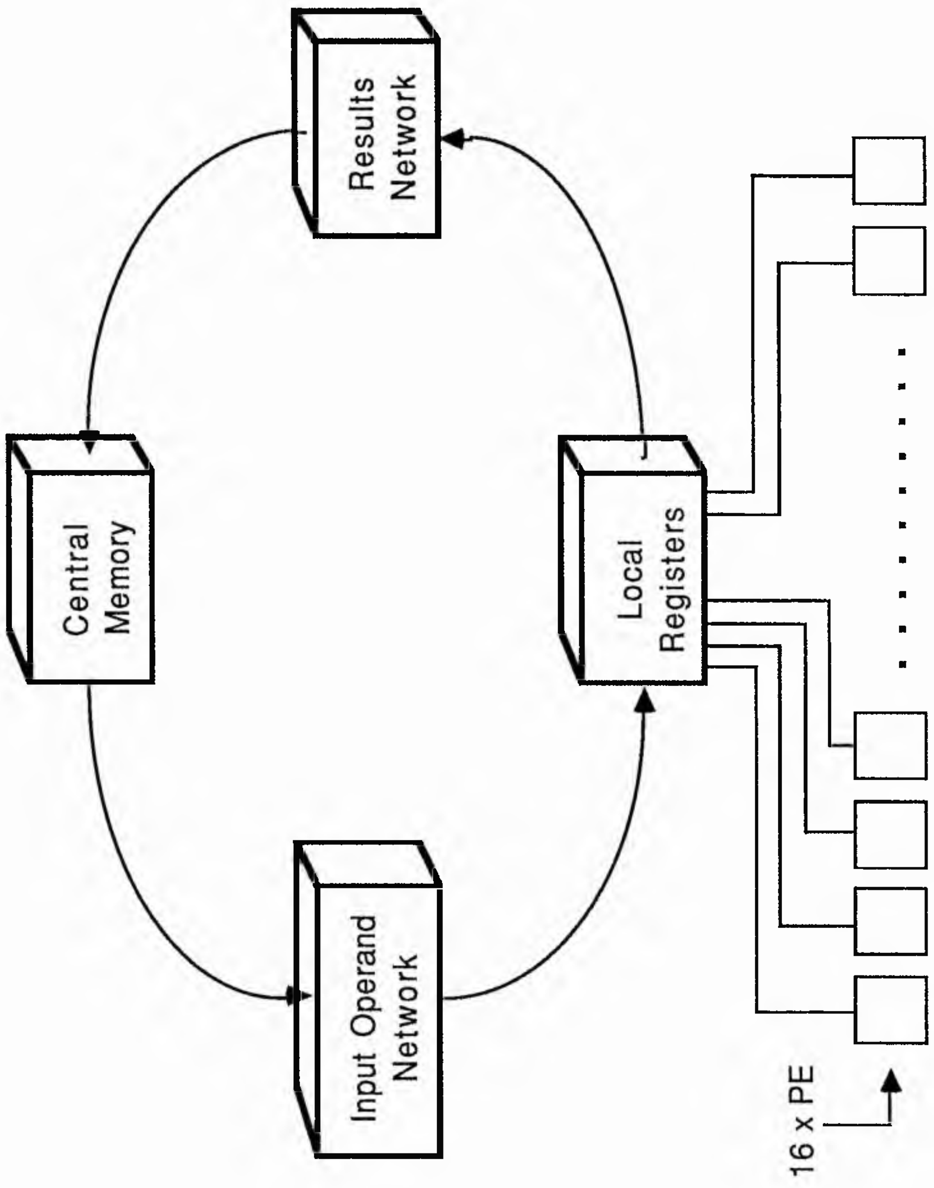


Figure 2.2 The Central Architecture of the BSP

The ICL DAP was first delivered in 1980 and consists of a 64x64 array of single-bit PE's of relatively modest speed. Each PE can communicate with its nearest four neighbours and with 4096 single-bit elements in DAP memory. There is thus, a total of 2 Mbytes of DAP memory, which is part of a larger memory in a host machine of the ICL 2900 series. The most powerful feature of the DAP is that the problem of coupling the processor array to memory is largely solved, since data can be structured both horizontally and vertically, or the structure can be transformed in some intermediate way by the attached host processor.

The other array computer mentioned, the Goodyear massively parallel processor (MPP), was due to be delivered about mid 1982. It consists of very large number (128 x 128) of single-bit PE's, designed all to operate in parallel on concurrent data streams of the same kind. Indeed the principle application of MPP is to concurrently process the pixels from satellite images.

Each PE in the MPP is considerably more powerful and faster than those in the DAP, and the level of integration much higher. Nevertheless the PE's are coupled in the classical way only to the nearest neighbours, since any form of alignment network to memory was considered (and indeed is) far too difficult to implement.

For improved maintainability, the array has four redundant columns of PEs. The physical structure of the PE array is 132 columns by 128 rows. Hardware faults are masked out with circuitry to bypass a faulty column and leave a logical array structure of 128 x 128.

The machines so far mentioned are all array processors. Array processing occurs when a group of processors perform the same instruction simultaneously on an array of data. Sometimes the processors themselves are arranged in an array, but sometimes they are pipeline processors, Krajewski [1985].

2.2 MIMD - Multi-Processors

In the last several decades many parallel variations of the von Neumann architecture have been developed. The idea behind them has been to take several processing units and memory modules and connect them in some network configuration. One prominent example of such systems is C.mmp, a multi-mini-processor system developed at Carnegie- Mellon University, Wulf and Bell [1972]. C.mmp consists of 16 processing elements, modified PDP-11/40E processors, connected to 16 memory modules via a crossbar switch, Hockney and Jessope [1981]. It can be shown that, as in any crossbar architecture, there are two kinds of buses: processor buses and memory buses. These buses are arranged in rows and columns and the connections, called Switch, are placed at each cross-point.

Each processor has an 8K-byte local memory store that is used for operating system functions. The shared memory provides a physical address space of 32 megabytes. There is also an interprocessor bus which connects the entire set of processors which is used for general interprocessor communications.

The crossbar switch permits communication between any memory modules and any processor. The existence of common memory permits close coupling between processors and thus reduces communication costs. But the complexity of the crossbar switch grows quite rapidly with the number of processors and memory modules involved, making it difficult to build these systems for more than 20 or 30 processors.

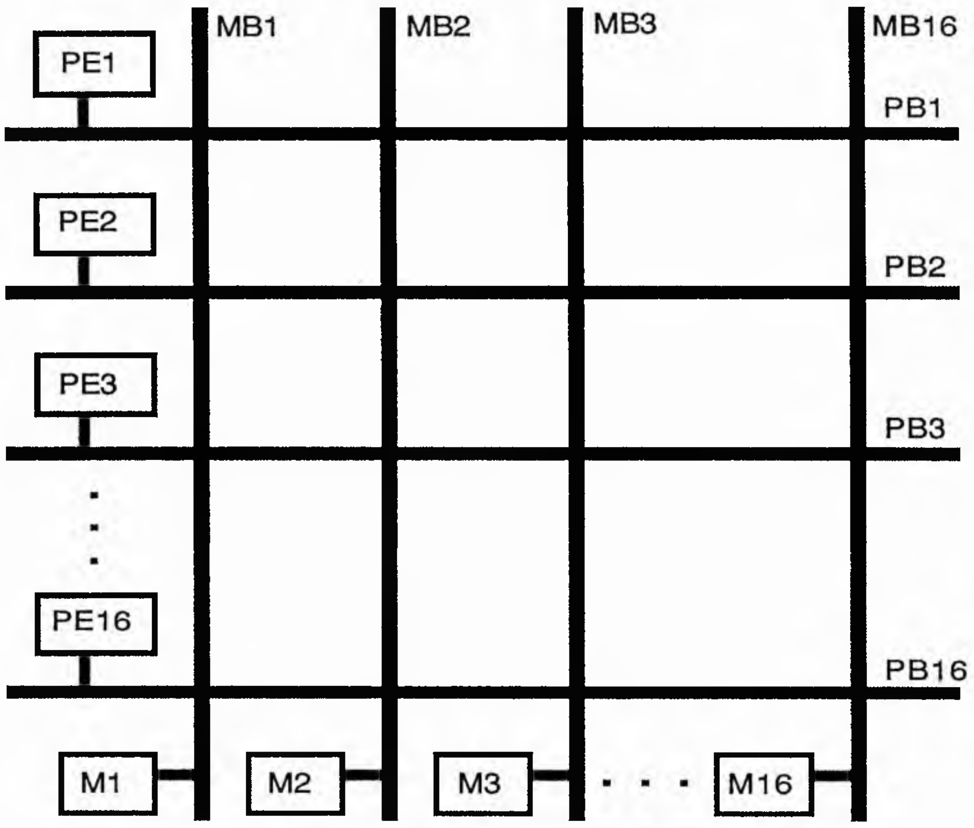


Figure 2.3 Description of the C.mmp System.

In some systems each processor is allowed to have private/local memory as mentioned for the C.mmp architecture, and these PE's are connected to each other by a common bus. These systems are easy to build for hundreds of processors. But communication occurs over a common bus, which makes interprocessor communication very expensive. Hence these systems cannot exploit fine-grain parallelism in an application.

TRAC, the Texas Reconfigurable Array Computer, Browne [1984], developed at the University of Texas at Austin, provides a middle ground. TRAC connects a number of processors to a number of memory elements via a Banyan network, Kruskal and Snir [1985], which is far less complex than the crossbar switch but provides reduced connectivity between the processors and memory elements.

Clearly the problem with the various architectures lies within the communication paths. The availability of communication paths that would enable the exchange of information between processors is the major drawback. The architecture of the Homogeneous multiprocessor, Dimopoulos [1985], is such that it ignores the need for every processor to be able to communicate directly to every other processor, as is the case with crossbar switching. Instead it is claimed that virtually every application can be formulated in such a way so that each computational subtask would require information from only its neighbouring subtasks to complete the computation. The Homogeneous multiprocessor is composed of k ($k>=3$) processing elements, k memory modules, a network of k switches isolating the processing elements from each other, and k I/O processors tied in a local network.

Each processing element owns its own memory module and accesses it via its local bus. It also has the exclusive use of its associated I/O processor. The local buses in the system are separated by intervening switches which provide the processors with the ability to communicate with either one of its two immediate neighbours by requesting the appropriate switch to close. Also, for I/O or data transfers to/from distant processing elements, each processing element may utilise the local network.

Such a system as the Homogeneous multiprocessor cannot be thought of as a general parallel processing machine because of its specific structure which requires that the applications be written specially for it due to the nature of its interconnection scheme. The critical part of the system lies within the network linked by the I/O processors, this controls the long distance communication which under some applications may be crucial and frequent.

2.3 The Transputer

A transputer is a microcomputer with its own local memory and several links for connecting one transputer to another transputer.

A typical member of the transputer product family is a single chip containing processor, memory, and serial communication links.

The transputer architecture simplifies system design by using point to point communication links. Every member of the transputer family has one or more standard links, each of which can be connected to a link of some other component. This allows transputer links of arbitrary size and topology to be constructed.

Each link consists of a serial input and a serial output, both of which are used to carry data and link control information. The communications are synchronised by a common clock link for the channel.

There are a number of research projects currently in process which are trying to utilise the transputer to its fullest capacity. One such project is the Esprit Reconfigurable Transputer Project which aims to construct a machine containing 64 nodes where one node is typically made up from 18 transputers.

The reconfiguration of the system is achieved by routing the transputer links through two software-driven VLSI switching circuits.

The ideas behind the transputer are faultless even though limited; major criticisms being the fixed topology and speed of communications through the links. The mode of communications is serial and hence the throughput is restricted because of this. If the same ideas where adapted to a parallel communications scheme rather than the serial mode then the system developed would have a larger capacity for data throughput. The ideas put forward in this thesis regarding reconfigurable virtual topologies may be adapted to form a new communications scheme for such a system as well as others.

2.4 Interconnection Networks

An investigation into present multiprocessor systems is more practically an investigation of the various connection strategies employed to make up the systems. In order to obtain a critical analysis of the various strategies employed the most appropriate approach would be to define what an Interconnection Network (IN) is to be, Bhuyan [1987]. Any processor/processing element should be able to access every other within the system. A shared bus system would be the least complex but limits the number of accesses to only one at a time, whereas with a crossbar switching network all possible distinct connections can be made simultaneously. Both systems have their drawbacks, the crossbar becomes too complex for a large number of elements and the shared bus limits access to one processor. It would seem that the shared bus option doesn't offer any great solutions, but if it is utilised in the right way in order to reduce the amount of communication then it may be the most feasible way of connecting a very large number of elements together in a system. Obviously whatever scheme is employed the benefits that it offers will only be for a limited number of applications so the aim in designing an IN would be to increase the number of applications as much as possible giving a general parallel system.

2.5 Analysis of Systems

There have been various attempts to analyse parallel systems through modelling. One of the more popular methods is that of Markovian models, Markenscoff [1985] and Marsan and Gregoretti [1981], used to evaluate configurations of multiprocessors, both multiple and shared bus systems. Because of the complexity of some configurations approximate models are applied which have shown to be very accurate for a wide range of configurations and loads.

Recent proposals and implementations indicate that bus structured interconnection networks are best suited to multi-microprocessor systems, Levy [1978], Thurber et al. [1972] and Kaiser [1980]. With this approach many different solutions for the interconnection network are possible, depending on the location of the shared memory modules and on the structure of the processing units, but little is known about the efficiency of each alternative. Hoener and Roeder [1978], presented a simple probalistic analysis of bus contention in a single bus multiprocessor system where processors are organised in a priority hierarchy. Willis [1978], considered a simplified model of multiple bus systems, assuming no queuing for busy resources. Fung and Torng [1979], developed a deterministic tool for the analysis of memory contention and bus conflicts in multiple bus multiprocessor systems. Marsan and Gregoretti [1981], used an asynchronous model to analyse the performance of a single bus multiprocessor system with a single common memory module. This analysis was extended to multiple bus and multiple common memory systems by Marsan and Gerla [1982].

It is difficult in general to prove that any given particular architecture is better than any other, but relatively easy to show its superiority for a specific application,-

CHAPTER 3

A NEW NODE INTERFACE UNIT

FOR PARALLEL PROCESSING

most machines to date have been designed with a particular application in mind. A natural corollary is that in order to obtain an interconnection structure that best meets the communications overheads of a maximum number of applications its architecture needs to be flexible in its reconfigurability to offer totally programmable interconnections among the elements.

CHAPTER 3

A NEW NODE INTERFACE UNIT FOR PARALLEL PROCESSING

When considering the construction of large parallel systems consisting of 1000's of processing nodes communicating through a global bus, the need is recognised for an interface unit (preferably a single silicon chip) to handle all data and control protocols between each node and the global bus. The design of the unit is seen to progress through three stages: conceptional definition of its function and operation, performance modelling to determine the superiority of the proposed system, and lastly a detailed design specification of the unit.

3.1 Concepts

3.1.1 Problem Formalism

Consider a vector function F to be executed on a multiprocessor system. It is divided into N sub-functions F_1, F_2 ... F_N which are executed in parallel. Sub-function F_i will compute new values for the vector variable X_i such that

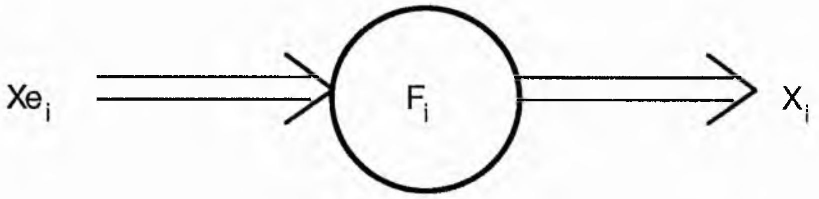
$$X_i(T+1) = F_i(X_i(T) , Xe_i(T))$$

where X_i is an n_i-vector of variables generated by node i;

Xe_i is an m_i-vector of external variables generated by other nodes in the system and needed in the i-th node to compute F_i;

and, F_i is an n_i-vector function.

The time index T relates the previous value of variables to the newly calculated values indexed by T+1.



$$Xe_i = \begin{pmatrix} xe_{i,1} \\ xe_{i,2} \\ \\ \\ xe_{i,m_i} \end{pmatrix} , \quad F_i = \begin{pmatrix} f_{i,1} \\ f_{i,2} \\ \\ \\ f_{i,n_i} \end{pmatrix} , \quad X_i = \begin{pmatrix} x_{i,1} \\ x_{i,2} \\ \\ \\ x_{i,n_i} \end{pmatrix}$$

Example: node 5 uses 3 external variables to generate 2 output variables. These variables and corresponding algorithms are represented as shown below:

$$Xe_5 = \begin{pmatrix} xe_{5,1} \\ xe_{5,2} \\ xe_{5,3} \end{pmatrix} , \quad F_5 = \begin{pmatrix} f_{5,1} \\ f_{5,2} \end{pmatrix} , \quad X_5 = \begin{pmatrix} x_{5,1} \\ x_{5,2} \end{pmatrix}$$

3.1.2 Common Variables

The collection of variables generated by all N nodes form a pool of common variables to be accessed for calculating new values. Thus two nodes i and j generate unique subsets of common variables by accessing overlapped subsets of their own and other variables as shown below in figure 3.1.

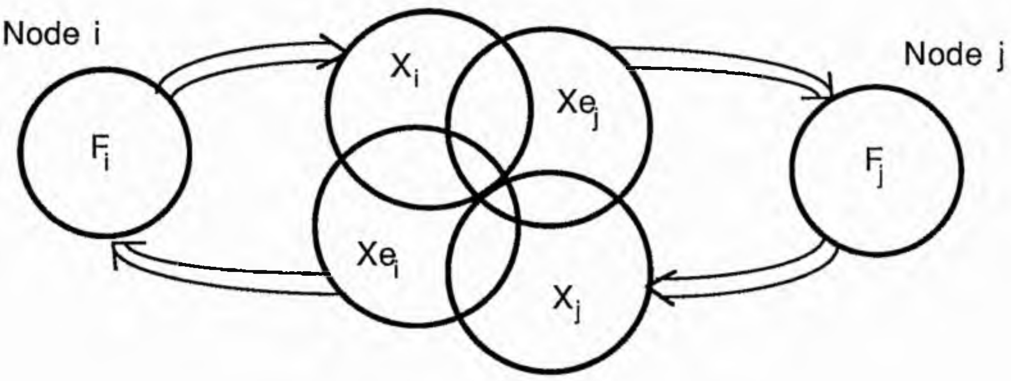
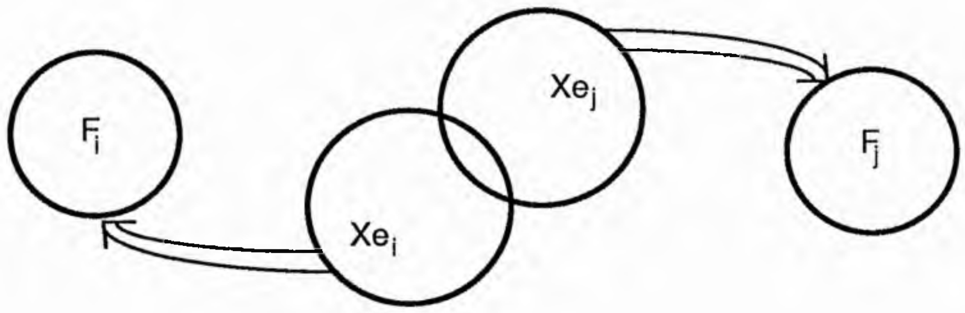


Figure 3.1 Unique and overlapped subsets of variables.

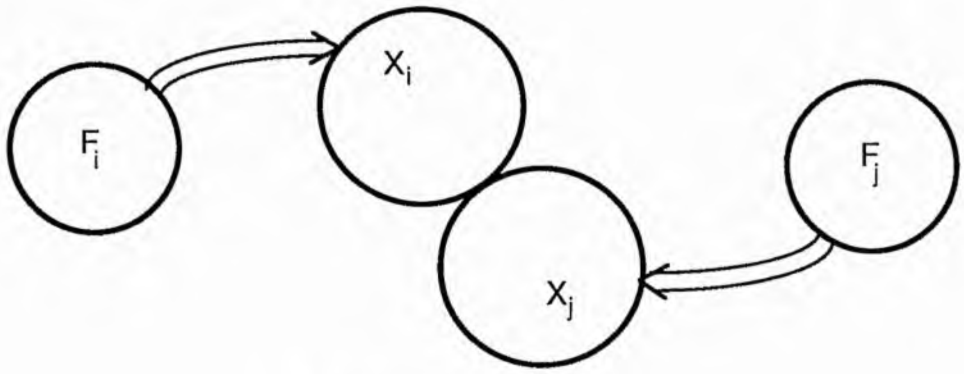
As nodes generate their variables at random, new values will over-write old ones, sometimes before the old values have had the chance of being used by corresponding nodes, thus leading to loss of synchrony and erroneous results.

3.1.3 Separate Computation and Update

To overcome the problem of over-writing values before they are used, the update of common variables is held back until all nodes have completed their computation, and thus no longer require the old values. Therefore each time cycle will consist of 2 phases: computation, followed by update, as shown in figure 3.2.



i. Computation phase.



ii. Update phase.

Figure 3.2 Separate computation and update phases.

3.1.4 Double Copy of Common Variables

The time delay that separates computation from updates may not be tolerated in practice. An alternative method of avoiding loss of synchrony is to provide a double copy of common variables: old and new, where nodes use the old copy (T) to access external variables but feed the newly generated values (T+1) into a second copy. Access to the two copies is then reversed before the begining of the next cycle, i.e. new becomes old, and the old is over-written with the new.

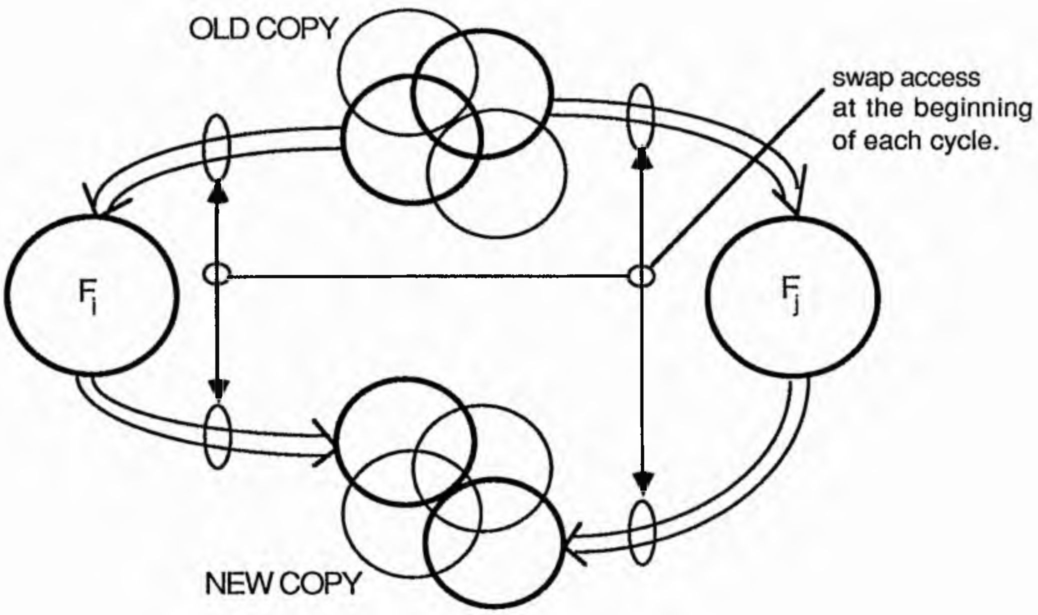


Figure 3.3 Double copy of common variables with cycle swapping.

3.1.5 Distributed Memory

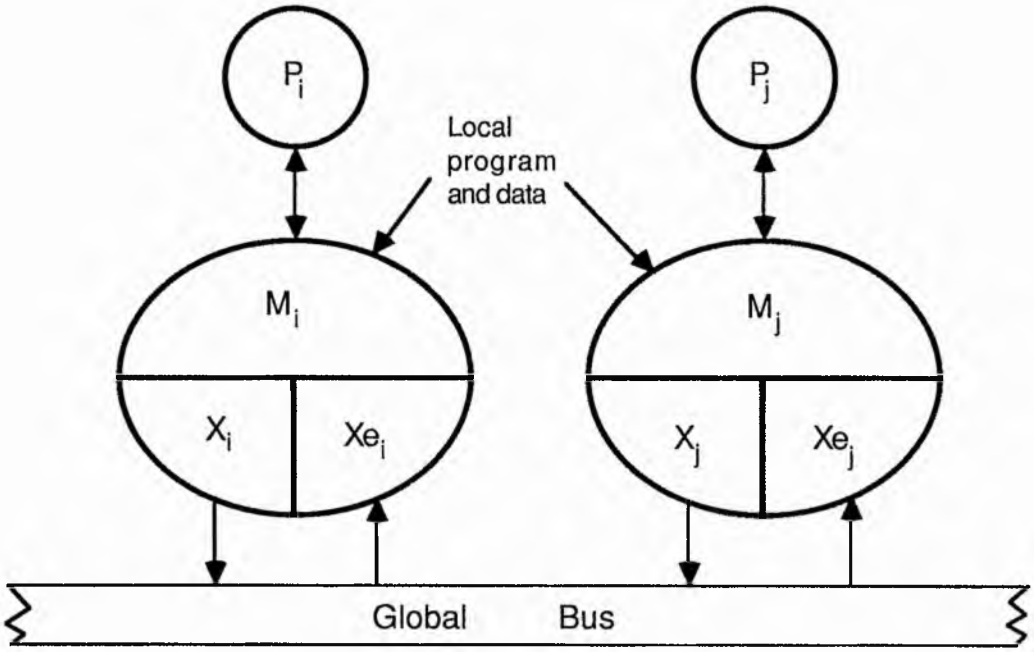


Figure 3.4 Distributed memory view of nodes.

To prevent bottlenecks caused by heavy access to global memory let each node have its own local memory to store its program, data and its own copy of the subset of common variables, the latter consisting of X_i and Xe_i, as shown in figure 3.4.

3.1.6 Input and Output Registers

The subset of local memory that holds common variables will need special interface facilities to maintain variable integrity and update synchrony. Such a communications interface would initiate the distribution of new values of variables X_i generated by each node into the external variable registers Xe's of destination nodes. Therefore the common variable interface for each node would consist of : output registers to hold X_i, input registers to hold Xe_i, and a communication unit C_i.

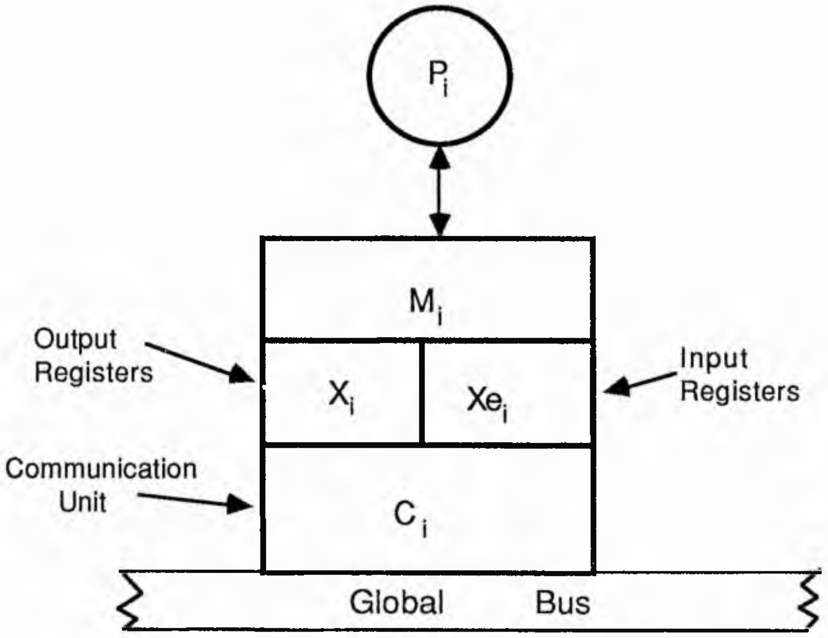


Figure 3.5 Communication unit with input and output registers.

3.1.7 Associative Addressing

Updating the multiple copies of external variables spread out over the whole system is carried out in response to a new value being written into an output register $x_{i,k}$. A broadcast protocol is the most efficient way of performing this function when complemented by associative addressing within the communication unit for the input registers.

A source node broadcasts the new value of $x_{i,k}$ together with its own address on the global bus. Nodes that find a match with this address copy the associated data into their input register. As all destination nodes perform this function at the same time, the saving in global bus cycles is clearly evident. The incorporation of address registers

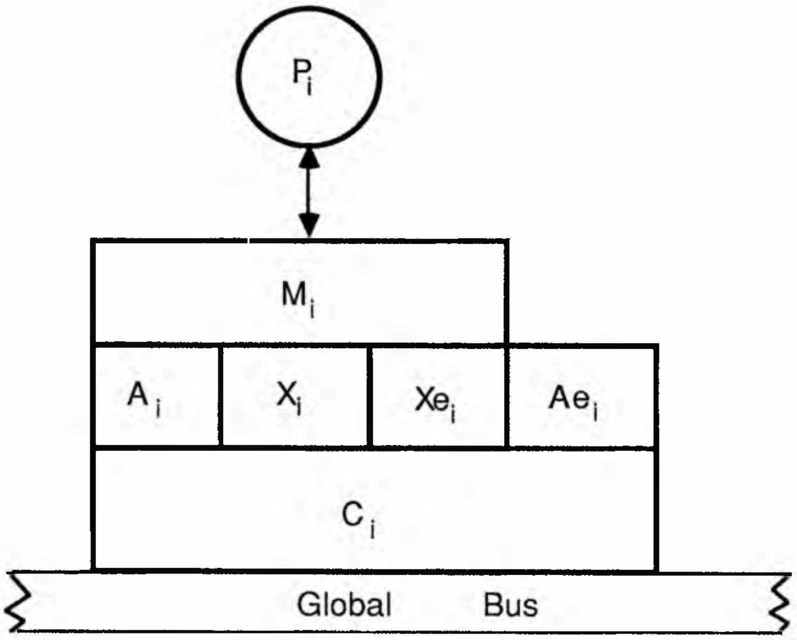


Figure 3.6 Incorporation of address registers in a node.

with input registers completes the fundamental structure of the interface unit. The address vector Ae_i is an m_i-vector which defines the source register addresses for the external variables vector Xe_i, i.e.

$$Ae_i = \begin{pmatrix} ae_{i,1} \\ ae_{i,2} \\ \\ \\ ae_{i,m_i} \end{pmatrix}$$

Moreover, A_i is an n_i-vector which simply defines the addresses of the variables generated by node i, i.e.

$$A_i = \begin{pmatrix} a_{i,1} \\ a_{i,2} \\ \\ \\ a_{i,n_i} \end{pmatrix}$$

In practice all nodes are expected to have the same number of output registers; however, not all registers within a given node are used in a given application.

3.1.8 Asynchronous and Data Flow Operations

The implied operation of the system considered so far assumes the availability of all data at the beginning of the cycle which is taken to be synchronised to a regular external event such as a clock or a master processor start signal. The updating of data is assumed to have taken place before the start of the new cycle.

The system also needs to operate asynchronously, where the start of execution of individual node workloads is dependent on the arrival of data from other nodes. Special features need to be incorporated within the interface unit to cater for this mode of operation. Apart from a facility to enable the master processor to switch the system between synchronous and asynchronous modes, the main requirement centres on the suspension of local processing until data arrives into the relevant input register. An interrupt line is triggered by the arrival of data and causes resumption of processing.

This completes the conceptual definition of the proposed interface unit. For ease of reference the unit will be termed MUPI (MUlti Processor Interface) and the resulting parallel processing system MINNIE (Multi Interfaced-Node Net for Iterative Environments), where 'iterative' refers to aapplications ranging from real-time embedded systems to numerical integration for dynamical systems which formed some of the original work in parallel processing, Al-Dabass [1976b].

3.2 Performance Models

The performance of the new system introduced above can be determined by comparing it with a system based on a global memory accessed through a single global bus employing conventional addressing techniques for data transfer, i.e. normal read/write operations, or a one to one channel between processor and global memory at any one time.

Consider that both systems contain V global variables which are available for inter-processor data transfer, i.e. the global memory system would contain V locations and the MUPI/MINNIE system would contain V output variable locations which are divided equally among the nodes. Both systems contain the same number of processing elements, i.e. there are N nodes. In terms of the analysis of the last section, V is the sum of the number of the X_i variables generated by the N nodes, i.e.

$$V = \sum_{i=1}^{N} n_i$$

3.2.1 Constraints

To ease the derivation of a performance model, but without loss of generality, a number of constraints are imposed. Let the number m_i of variables Xe_i used by each node be the same and equal to I (Input variables). Let the number n_i of variables X_i generated by each node to be the same and equal to O (Output variables). Therefore with the global memory system each node is able to read from I (input) locations and write to O (output) locations. With the MUPI/MINNIE system each node contains I input registers and O output registers. In both cases the total number of global variables V are distributed evenly among the nodes in the system i.e. V = N x O.

The model is derived from considering the worst case situation of both systems. The worst case situation would be that when all of the global variable locations in both systems are to be utilised, i.e. all O locations would be written to at least once and all I locations would be read from at least once. For convenience and again without loss of generality, it shall be assumed that all locations are accessed only once.

A list of all variable names and their meanings which are used in the performance models are shown in table 3.1.

I	Number of Input Registers
O	Number of Output Registers
N	Number of Nodes in the System
F	Function to be executed
t_1	Execution time on a single processor system
η_g	Number of accesses to Global memory system bus
η_m	Number of accesses to MINNIE System Bus
t_g	Execution time on a Global memory system
t_m	Execution time on MINNIE system
ϕ	Overheads fraction
τ	Time taken for single access to system bus

Table 3.1 Variables used in the Performance Models.

3.2.2 Global Memory System Model

In this case the total number of global memory accesses i.e. the total number of global bus accesses is given by:

$$\eta_g = N(I+O) \tag{3.1}$$

Given that a function F can be executed in t_1 time units on a single processor system, the time it takes to execute on the global memory system can be estimated by dividing the overall solution into two phases: a computation phase and a communications phase. The time for the computation phase will be assumed to be the quotient of the single processor time t_1 and the number of processors N. Some absorbtion of communications overheads within computation would normally take place to leave only a fraction of the total data transfer time to represent the second phase, i.e.

$$t_g = t_1/N + N(I+O)\phi\tau \tag{3.2}$$

where τ = time taken for a single bus access and ϕ is a dimensionless overheads fraction such that ϕ of the bus accesses occur after time t_1/N, i.e. after the computation phase of function execution.

3.2.3 MUPI/MINNIE System Model

In the case of the MUPI/MINNIE system the total number of global bus accesses is reduced because of the employment of associative addressing, i.e. there are no read operations, just write operations. The actual number of global bus accesses is given by:

$$\eta_m = NO \tag{3.3}$$

Otherwise, an identical argument to the one above is employed to determine the execution time for the new system, which will clearly be,

$$t_m = t_1/N + NO\phi\tau \tag{3.4}$$

3.2.4 Comparison

To compare the two models given by equations (3.2) and (3.4), consider the worst case situation where there is no absorption of communications within the computation phase, i.e. all communications occur after computations, giving $\phi = 1.0$. Furthermore, the bus access time τ can be eliminated from both equations by expressing the execution times t_1, t_g and t_m in bus access time units. This gives the following pair of equations for the overall execution time on the two systems:

Global Memory System: $$t_g = t_1/N + N(I+O) \tag{3.5}$$

MUPI/MINNIE System: $$t_m = t_1/N + NO \tag{3.6}$$

3.2.5 Illustrative Examples

Consider three seperate examples where the function F takes $t_1=100$, $t_1=1000$ and, $t_1=10000$ time units on a single processor. Let the number of input and output variables for each node in the systems be I=4 and O=1, for all three examples. Equations (3.5) and (3.6) reduce to :

Global Memory System: $$t_g = t_1/N + 5N \qquad (3.7)$$

MUPI/MINNIE System: $$t_m = t_1/N + N \qquad (3.8)$$

Figures 3.7, 3.8 and 3.9 show the decrease in execution time as the number of nodes in the system are increased. It must be stressed that when a particular function is divided into N separate parts these are not usually of equal execution durations; however, the assumption that they are is valid enough for the comparisons of the two separate systems. It should also be pointed out that as the function is further divided into a greater number of sub-functions that these sub-functions may require more or less communications but again the assumption that all variables are accessed only once is acceptable for comparison purposes.

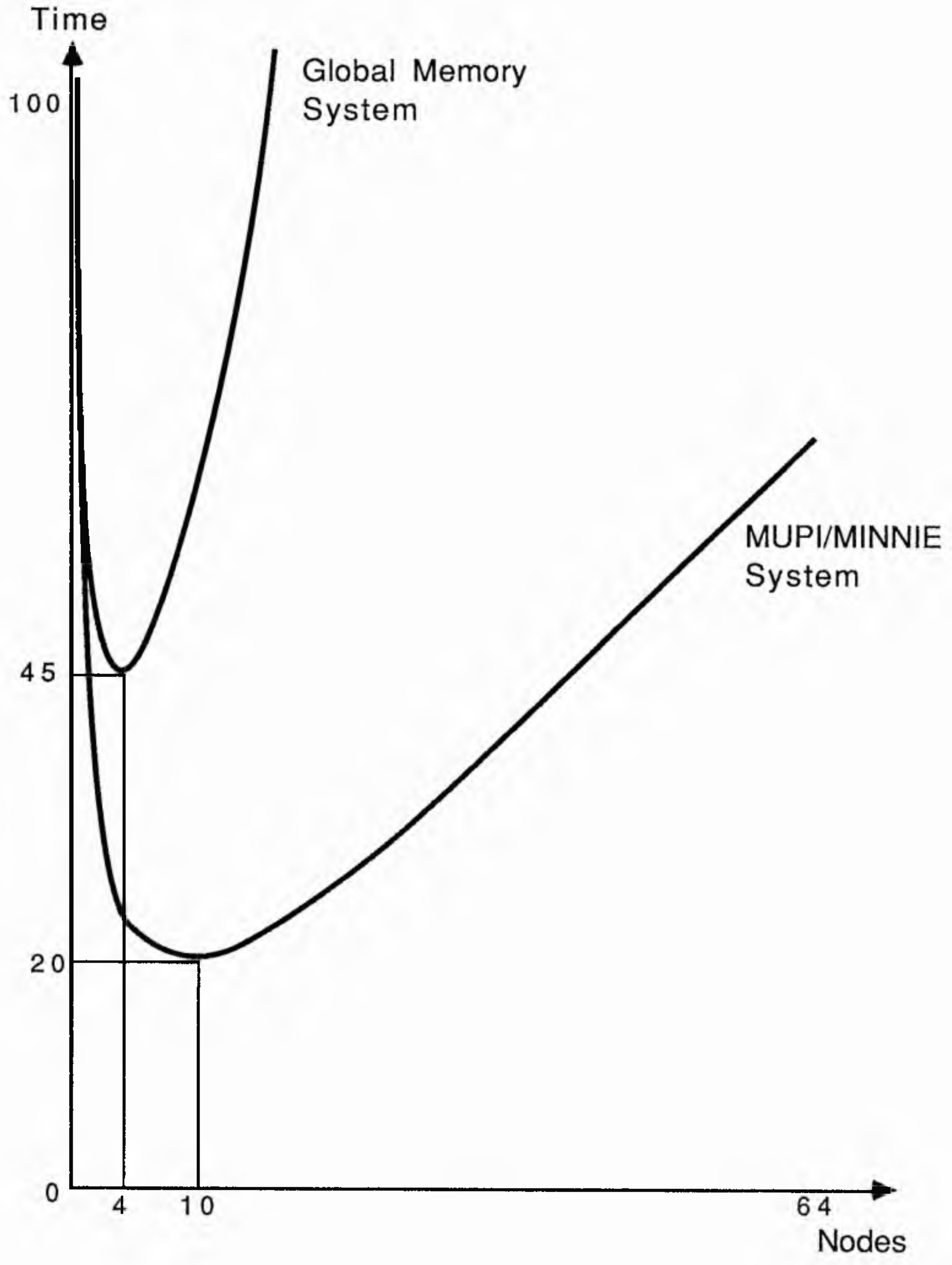


Figure 3.7 Performance models, t_1=100 and ϕ=1.0.

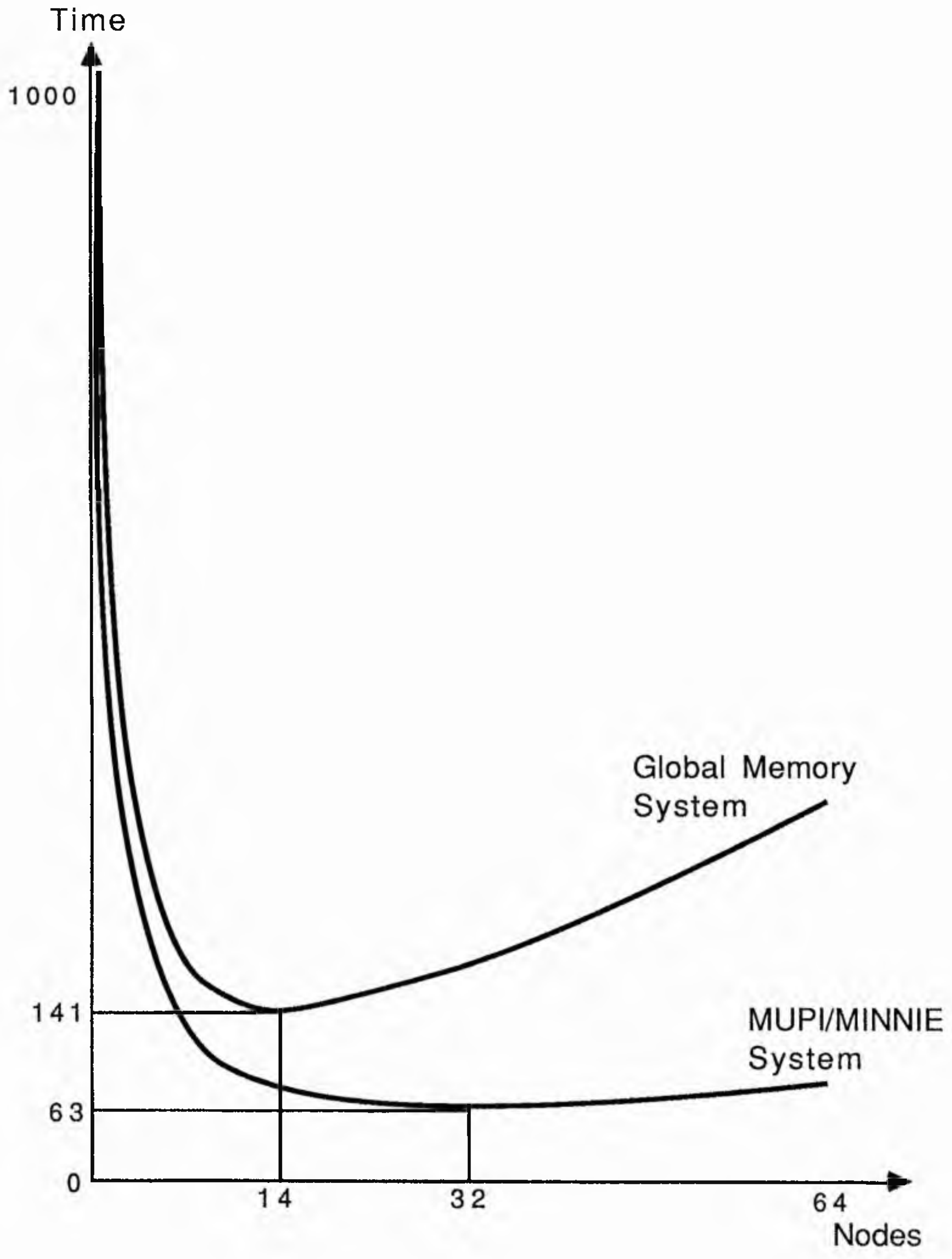


Figure 3.8 Performance models, t_1=1000 and ϕ=1.0.

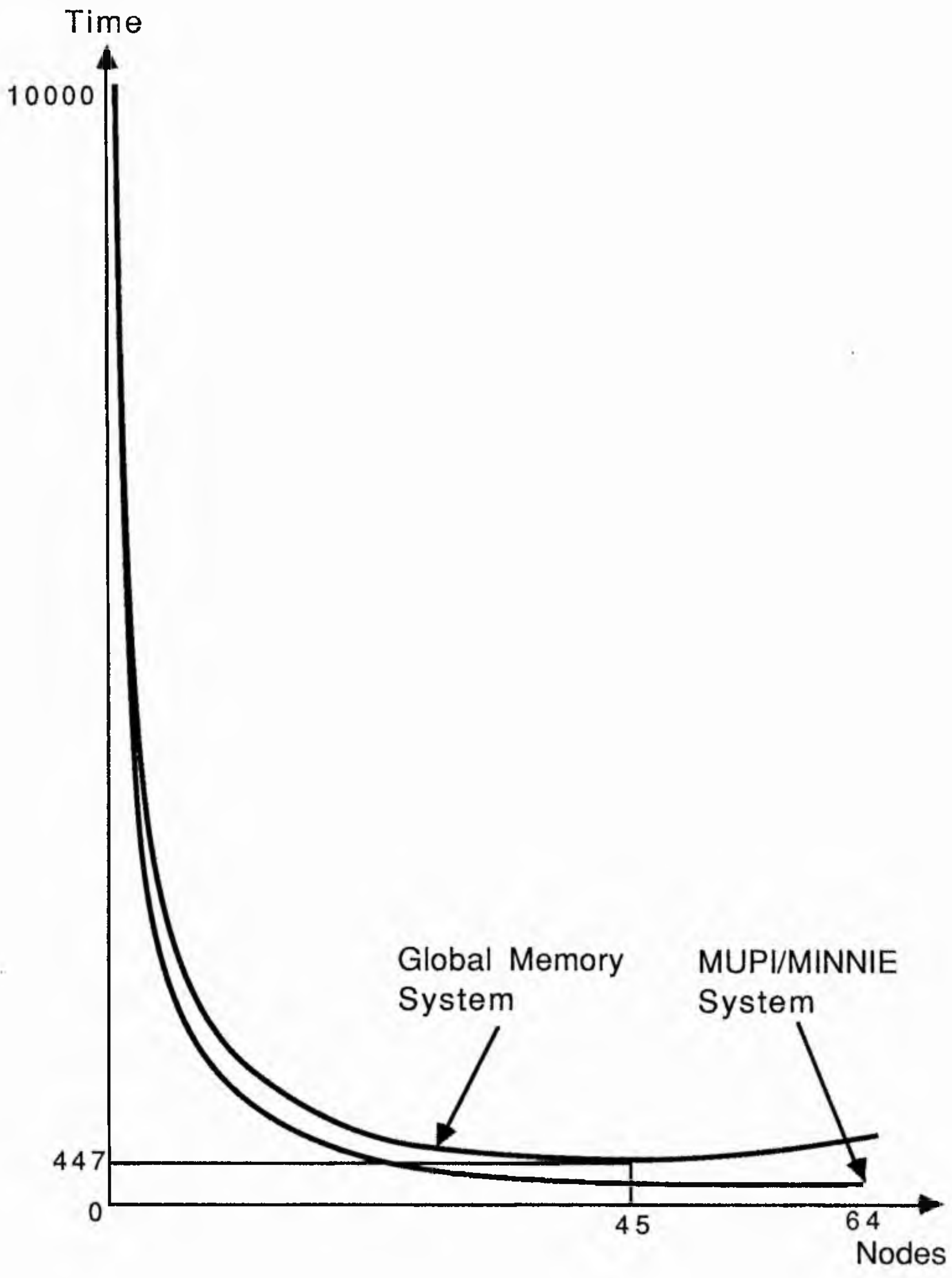


Figure 3.9 Performance models, t_1=10000 and ϕ=1.0.

As is evident from figures 3.7, 3.8 and 3.9 a function F would take longer to execute on the global memory system than it would on the MUPI/MINNIE system. Furthermore, an optimum point is soon reached beyond which any further increase in the number of processors would only increase the total execution time rather than decrease it. Note that the corresponding minimum point is lower for the MUPI/MINNIE system than for a global memory system, even if it is achieved using more processors. The minimum point occurs because at some point in dividing a function the communications phase will start to have a greater influence on the overall problem solution time than the computation phase. This is to be expected from the position of the variable N in the two terms for computation and overheads in equations 3.7 and 3.8.

3.2.6 Minimum time and optimum number of processors

Equations (3.2) and (3.4) are similar in structure and can be reduced to the following form:

$$t = t_1/N + NC \tag{3.9}$$

where C is taken to be a constant and is system and function dependent. The forms of C for the two systems are given in equations (3.2) and (3.4) as

$C=(I+O)\phi\tau$ for the global memory system, and

$C=O\phi\tau$ for the MUPI/MINNIE system.

Differentiating equation 3.9 with respect to N gives,

$$dt/dN = -t_1/N^2 + C \tag{3.10}$$

equating to zero and rearranging gives

$$N_{opt} = \sqrt{(t_1/C)} \qquad (3.11)$$

which for the two separate systems yields,

Global Memory System: $$N_{opt} = [t_1/((I+O)\phi\tau)]^{1/2} \qquad (3.12)$$

MUPI/MINNIE System: $$N_{opt} = [t_1/(O\phi\tau)]^{1/2} \qquad (3.13)$$

Substituting N_{opt} from equations (3.12) and (3.13) into equations (3.2) and (3.4) and manipulating gives:

Global Memory System: $$t_{min} = 2[\phi\tau\,(I+O)t_1]^{1/2} \qquad (3.14)$$

MUPI/MINNIE System: $$t_{min} = 2[\phi\tau\,Ot_1]^{1/2} \qquad (3.15)$$

The forms of the equations for the minimum achievable time are such that when 3.14 is divided by 3.15 they give the following ratio for the minimum execution times,

$$R_{min} = [(I+O)/O]^{1/2} = [1 + I/O]^{1/2} \qquad (3.16)$$

A particularly significant result as it is independent of the three variables of single processor time t_1, bus time τ and overheads fraction ϕ.

Table 3.2 below shows the increase in the minimum time ratio as a function of the ratio of the number of input to output registers I/O. The presence of this ratio is to be expected as the difference between the two systems is due to the elimination of input accesses I from the new system model. It is clear that the greater the number of data input registers required in a system, assuming the same number of output registers, the more powerful does the new system become as compared with an ordinary global memory system.

I/O	1	2	4	8	16
R_{min}	1.4	1.7	2.2	3	4.1

Table 3.2 Values of R_{min} for various I/O.

3.3 Design Specification

The design of the new interface is divided into the following modules:

(1) Read Unit;

(2) Write Unit;

(3) queuing Unit;

(4) Node Address Unit;

(5) Bus-Switch Unit;

(6) Allocation System.

Each unit performs a specific function within the interprocessor communications process and the global processor control functions.

3.3.1 Read Unit

A specification for the design of the unit is developed as follows:

(i) The read unit performs two functions; (a) reading in data from the global bus; and (b) responding to read commands from the local processor. To allow for anticipated hardware limitations, the read unit is specified to have four data registers (D1-D4) to hold data read-in from the global bus, figure 3.10 There is no conceptional limitations to the number of registers or word lengths.

(i i) When performing the first function, reading data from the global bus, the read unit is constantly comparing the address on the global address bus with a store of addresses, input variable addresses (IVAs). The IVAs correspond to the memory elements that the processor requires to read from in the original global memory system, and refered to as Ae_i (Addresses of external variables for node i) in figure 3.6 earlier.

If a match with one of the IVAs is found then the read unit recognises that its local processor requires the data currently on the global bus. This data is copied into its relevant data register, pointed to by the matching IVA, and a corresponding data flag is set. This flag forms an interrupt signal (enabled by the processor) which alerts the processor to the arrival of data in a given register. This is the associative addressing mode introduced earlier, i.e. all of the IVAs are compared simultaneously.

The interrupt facility enables the processor to overlap processing with input.

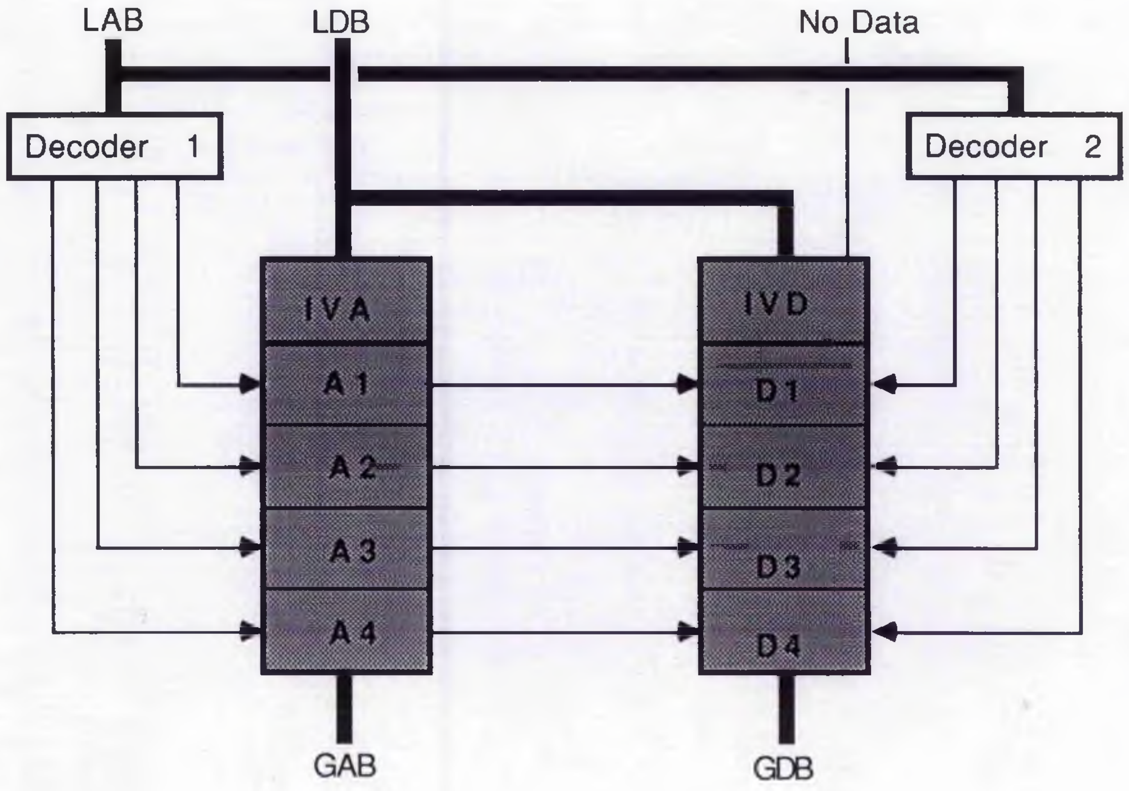


Figure 3.10 Read Unit.

(iii) IVAs contents are variable and can be altered by the local processor through decoder 1, figure 3.10. With this flexibility the local processor can vary its image of the global memory i.e. it can specify which locations it wishes to read from at any one time.

(iv) When the local processor wishes to access one of the data registers it performs a normal read operation on it through decoder 2. If the register has not yet received its data from the global bus, i.e. its corresponding data flag has not been set, then if the processor has enabled its interrupts it will receive a 'no-data' interrupt from the read unit. If the processor has not enabled its interrupts then when access of a data register is performed there will be no resulting interrupt and the processor will recieve whatever data is currently resident in the register. If the data flag is set then the read operation from the local processor will occur as normal.

If the local processor tries to read a data register while it is being updated via the global bus then it will receive a 'no-data' interrupt. The data will only be available when the update is complete. Two separate data register copies may be accessed at the same instance, one by the local processor, the old copy, and one by the global bus for updating, the new copy.

(v) If two or more MUPIs recognise the address on the global address bus, i.e. achieve a match with one of their IVAs then all of the matches would result in the data being copied simultaneously. As such, interfaces operate independently while copying data.

3.3.2 Write Unit

(i) The write unit broadcasts data that a local processor has calculated onto the global bus. Although this unit may be designed to incorporate any number of data items, this specification will only refer to a system with four write registers, without loss of generality.

(i i) When a local processor generates a data item required by other processors it writes the data to its MUPI unit. Decoder 3 (figure 3.11) within the write unit recognises this and stores the data in an output data register and sets its associated data flag. The unit, like all other units, is memory mapped so the processor will access the locations as special addresses within its own memory map.

(i i i) To write the data onto the global bus the interface unit requires control of it. The write logic sends a bus request signal to the queuing unit which in turn replies with a bus acknowledge signal when control of the global bus is granted. On receiving this signal, the write logic copies the data stored in its output registers onto the global bus. Only one full register, i.e. one of those with a set data flag, is recognised and processed at a time. The write logic signals to the active register to place its contents on the global bus while at the same time the logic generates the address associated with this register and places this on the global address bus.

(i v) The address is generated by combining the node address, obtained from the address unit, with the output data register address which is generated by the write logic.

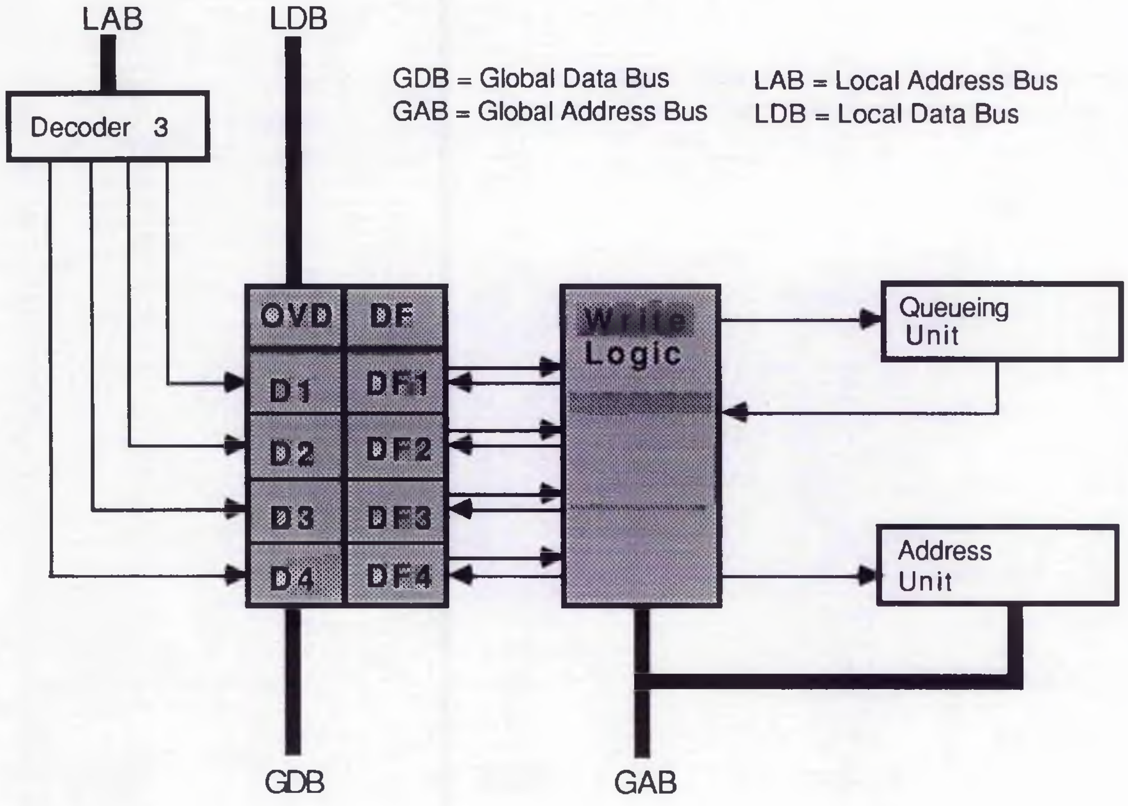


Figure 3.11 Write Unit.

While the output data register is placing its contents on the global data bus it clears its data flag; the write logic then iterates through the other flags to find the next set flag. The process is repeated on the next full output data register.

(v) The process checks all registers in such a fashion that if data is entered into register 3, say, while register 4 is active (writing to the global bus) then its contents can eventually be written onto the global bus within this bus operation. Once the logic concludes that there are no more full output data registers then it resets the bus request signal so that the queuing unit can release control of the global bus.

3.3.3 Queuing Unit

The queuing unit performs the function of gaining control of the global bus. The queuing process is activated by a bus request signal from the write unit (see section on write unit), and is specified as follows:

(i) The queuing unit simply waits for a signal from the global bus (bus queue signal) asking the unit if its write unit needs the bus, figure 3.12. If it does, the queueing unit halts the bus queue signal to prevent its propagation to the next node, and sends a bus acknowledge signal to the write unit indicating that the global bus is available for use.

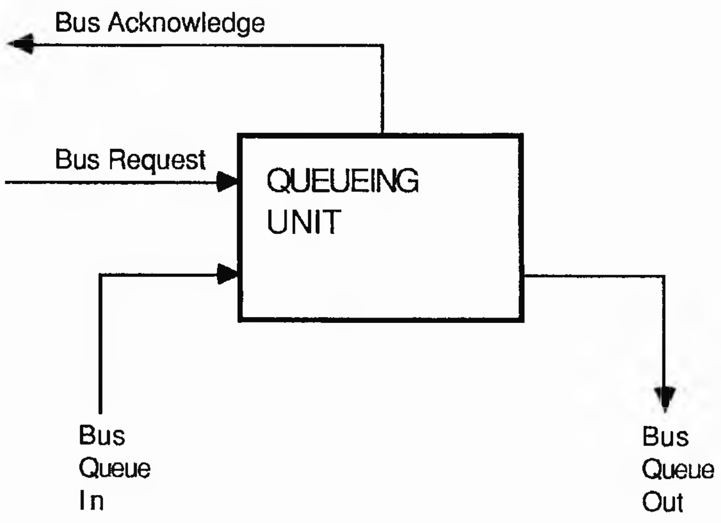


Figure 3.12 Queueing Unit.

(ii) When the write unit no longer needs the global bus the queuing unit releases the bus queue signal, i.e. it allows it to continue along the global bus. As such, the bus queue signal is a round robin signal which travels around the system from one MUPI to another, and is generated by the Master processor, figure 3.13.

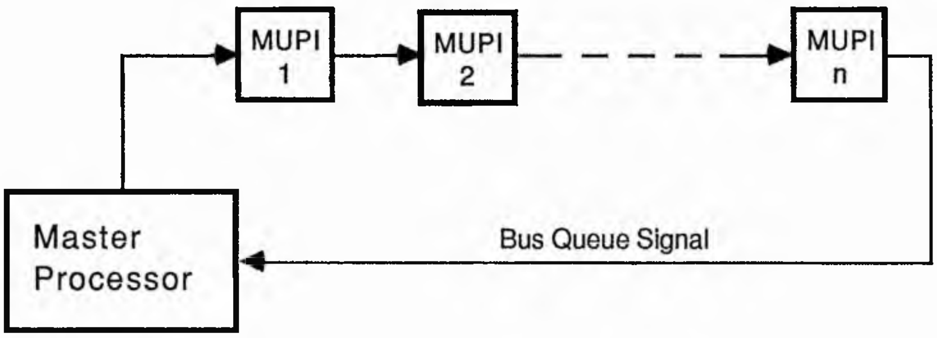
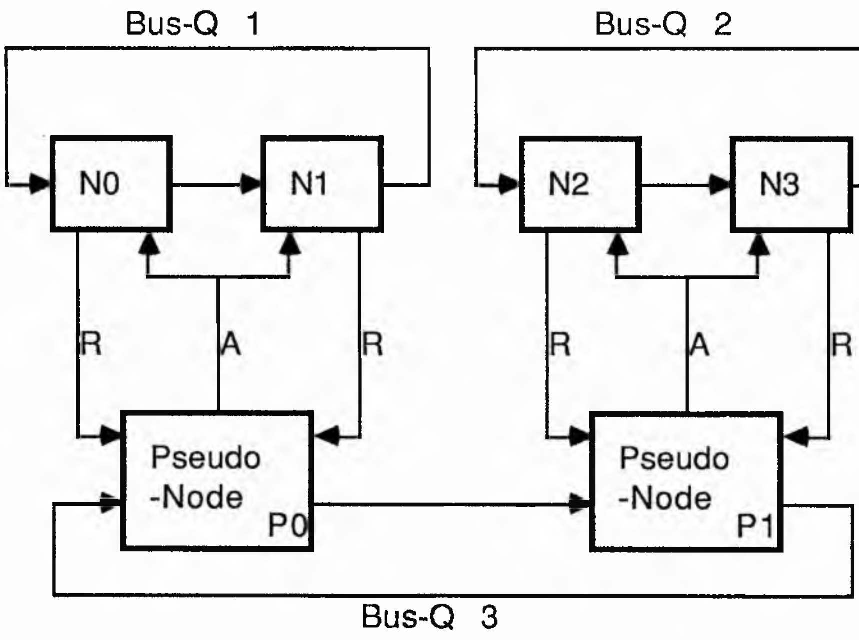


Figure 3.13 Queueing System.

(iii) This is a single tier queuing scheme suitable for a small number of nodes. The criteria for limiting the number of nodes is the time taken by the queue signal to propagate round the nodes, - this time having to be 'insignificant' by comparison to the global bus cycle time, e.g. an order of magnitude smaller. Furthermore, to enable the global bus to deal efficiently with a large number of nodes, its cycle time needs to be much smaller than the local processor/memory bus cycle time; again say an order of magnitude smaller.

Practical values would be: 1µS for local bus cycle; 100nS for global bus cycle; and 10nS for the propagation cycle across each node of the round-robin signal.

(iv) For a large number of nodes a hierarchical system of round robin tiers is introduced to speed up the detection of access requests to global bus. For instance, at the highest level there will be several groups each of which contain two devices as shown in figure 3.14. Each group of two devices would have its own bus queue signal. The groups are also clustered together to form a second series of groups for the next level and these again would have their own bus queue signal.



R = Request A = Acknowledge

Figure 3.14 Bus-Q system.

(v) If device N0 requests the bus then it would wait for its local bus queue (Bus-Q 1) to reach it. Bus-Q 1 remains in N0 and a request signal (R) is sent to the next level i.e. to Pseudo-Device P0. When P0 receives a request signal it waits for its own bus queue (Bus-Q 3) to reach it. When Bus-Q 3 reaches P0, the device will hold the bus queue signal and send an acknowledge signal (A) to the upper level. Both devices in group 1 of the upper level will receive the acknowledge signal but only the device with the local bus queue (Bus-Q 1) will act i.e. device N0 is granted the use of the global bus.

(vi) When a device no longer needs the global bus, it releases its local bus queue signal, which drops the request signal to the pseudo-device. The pseudo-device then releases its bus queue signal and drops the acknowledge signal to the higher level. When a device is using the global bus another device in another group e.g. device N3 in figure 3.14 can request the bus; this will eventually result in a request signal being sent to its pseudo-device (P1) which will wait for P0 to release Bus-Q 3.

(vii) In a multilevel bus queue system the maximum time that a device has to wait from first requesting the bus to finally being granted access to it is clearly proportional to the number of levels, which for a binary tree system with n devices (2 devices per level) gives

$$n=2^m \tag{3.30}$$

i.e. $$m=\log_2 n \tag{3.31}$$

where m is the number of wait units, and

1 unit = time for allocator to travel from one device to the next.

Whereas with the initial single level allocator system that this was developed from would result in the maximum waiting time of (n-1) units.

The system for the multi-level allocators shown is based on a binary tree i.e. two devices at each grouping and this can be extended to contain further devices then the four shown and also to contain various numbers of devices in the groups.

3.3.4 Node Address Unit

The node address unit contains a register for the storage of a node address and its associated logic for the master processor to set up the address and for providing it to other units within the MUPI. The specification is developed as follows:

(i) Node addresses for all MUPIs are set up by the master processor during the initialisation stage. A daisy chained line from one MUPI to another provides an enable signal to the node address unit giving the Master processor access to its address register. The daisy chained signal is sourced at the master processor and is clocked from one MUPI to another so that the master can in turn set up the various node addresses within the system. This signal is necessary as there is no other means to distinguish one MUPI from another before the addresses are set up.

(i i) When the master processor clocks the enable signal through to a MUPI it has complete access of the node address register of that MUPI, both read and write operations can be carried out. Once the address of a MUPI has been set then the master processor either clocks the enable signal through to another MUPI or it resets the enable to avoid any MUPI units being in an unknown state i.e. their node address being set but unknown.

(iii) The master processor in writing to a node address register within a MUPI sets a flag within the node address unit of that MUPI signifying that the node is an active node. This is important as not all the nodes in MINNIE will be required for certain applications.

(iv) On request from the write unit the node address unit will place its address onto the global address bus combining it with one of the output data register addresses to form the full data address.

(v) The node address unit also provides the node address to the bus-switch unit, see section 3.3.5.

3.3.5 Bus-Switch Unit

The bus-switch unit serves the function of switching the global bus through to one of the local buses to give the master processor access to the local memory modules, one at a time. This function is required so that the master processor can provide the local processors with their sections of the program and data for processing. Also, for the retrieval, if necessary, of data after the execution of the task being processed.

(i) The switch is straight forward in its operation: when the master processor wishes to access the local memory module of a node, it would place the node address of that node onto the global address bus with a selection of control signals that specify switch on. The address will be interrogated by all bus-switch units of active nodes in the system, i.e. only the nodes whose node addresses have been set up. When a match is met the switch unit with the match will place the local processor into a halt state, i.e. suspend its operations, and switch the local bus through to the global bus.

(i i) Once the switch is active the master processor has access to the local memory module of that node and performs read and write operations on it as if it were part of the master processors own memory.

(i i i) The bus-switch is deactivated by one of three methods. The first method, used as a last resort or as a final operation, is a global reset which will reset the whole system i.e. all node addresses are reset and all active nodes become inactive. The second method is to activate the switch of another node, i.e. a different address to the one presently activated is specified in the master processors bus-switch operation. The third method is to send a set of control signals that specify switch off, switching any active switch within the system off.

3.4 Modes of Operation

A general purpose multiprocessor system should be flexible enough to operate in a wide variety of modes. The design specification must include a large degree of flexibility to enable the user to reconfigure the system to any one of a number of modes. Two modes of operation are considered to be essential in any general purpose multiprocessor:

(a) Cyclic Mode;

(b) Data Flow/Asynchronous Mode.

3.4.1 Cyclic Mode

This mode of operation is probably the most utilised mode of the three. Its strength lies in its ability to execute a system of equations via successive iterations i.e. the values of the next state are dependent upon the values of the previous state. The specification for its operation is developed as follows:

(i) A typical set of equations would be as follows:

$$x(T+1) = x(T) + Ay(T) \tag{3.19}$$

$$y(T+1) = y(T) + Bx(T) \tag{3.20}$$

Equations (3.19) and (3.20) are coupled and would be computed in parallel; they are linked through the results of the previous computation. These equations can be implemented on the system specified above with a slight modification to the original specification of the MUPI read units. The solution uses two separate nodes of the system, N1 and N2. Each node will have a store of initial values i.e. values x(0) and y(0). The first cycle begins and the nodes compute the values for i=0 i.e. N1 computes x(1) and N2 computes y(1). The values for i=0 are calculated and then placed onto the global bus to be transferred to the other node as they will be needed for the next computation cycle, i.e. x(1) will be sent to N2 and y(1) will be sent to N1. When the computation and data transfers have been completed this is made known to the master processor which restarts the system for the next cycle.

(ii) The processors execute the equations in the second cycle in the same manner as the first except that the previous values are now stored within the read units of the interfaces. If one of the processors were to compute its new value e.g. N1 computes x(2) and transfer it to N2s read unit, before N2 accessed its read unit for x(1) then when N2 does eventually access x(1) it results in an error as x(2) will be in its place.

(iii) The solution would be to provide a separate register within the read units for each value of x in the case of N1 and each value of y in the case of N2. This would require that the read units contain a large bank of input registers when computing a large number of values for the system of equations, which is usually the case.

(iv) A more effective solution is to double-up the input registers so that when the system is placed in the cyclic mode by the master processor one set of input registers is accessible by the local processor for reading and the other set is accessible by the read unit for updating with data from the global bus, see figure

3.15. At the end of each cycle the register sets are swapped so that the local processor can access the data that was updated in the previous cycle. If it is required that all the values from all the cycles be kept then this can be achieved in local memory which can be accessed by the master processor when a specific number of cycles have been completed.

(v) The read unit would still need to behave in the same manner as described previously, i.e. data is copied into the input registers if the address is matched. When the local processor reads a data register and there is no data the processor is not interrupted as this would be pointless i.e. the data will have arrived in the previous cycle if it was to arrive at all. When the system has completed its execution in the cyclic mode the master processor will switch this mode off returning the interfaces to a single bank of input registers which are accessible by both the local processor and the global bus.

(v i) The derivation of a performance model for this mode utilises the fact that a great deal of the communications occur towards the end of each cycle i.e. after the computation phase has finished on a given node. Thus on a well balanced distribution of the workload, there is little chance for the communications to be absorbed within computations. The model presented in section 3.2 will apply here, where ϕ will assume a value near to unity; although a more practical figure would be $\phi = 0.9$ due to the presence of some absorption as not all nodes in the system will complete their computations at the same instance.

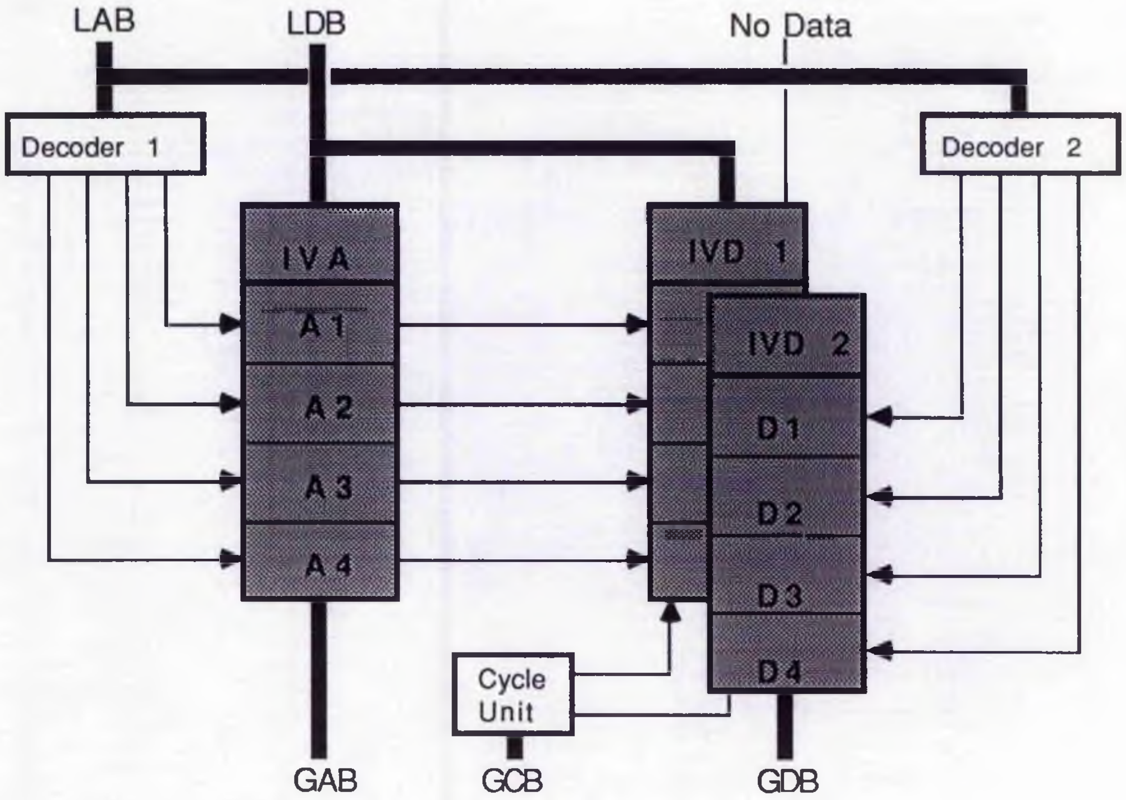


Figure 3.15 Read Unit with Double Registers.

(vii) With an overheads factor of $\phi = 0.9$ the performance of the proposed system (MUPI/MINNIE) shows a larger difference to that of the global memory system given in section 3.2. Hence for the cyclic mode of operation the performance models for the two systems are as follows:

Global Memory System: $t_g = t_1/N + 0.9N(I + O)$ (3.21)

MUPI/MINNIE System: $t_m = t_1/N + 0.9NO$ (3.22)

A plot of these two equations with $t_1 = 100$, $I = 4$ and $O = 1$ can be seen in figure

3.16. It is clear that the gap between the two curves is smaller than that in figure 3.7. The 10% decrease in the overheads factor has eroded the advantages of the new system by a small ammount.

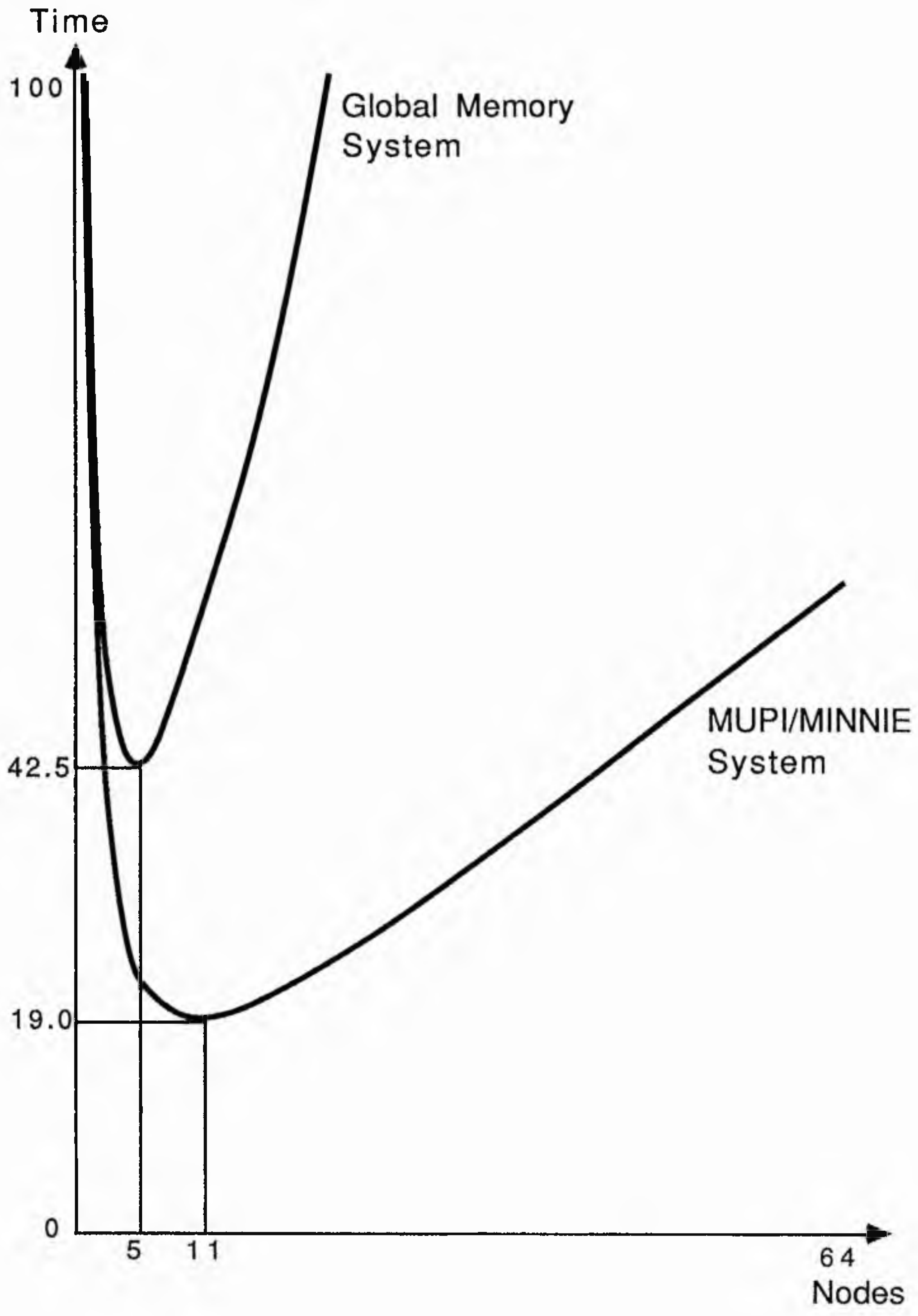


Figure 3.16 Performance models, $\phi = 0.9$.

3.4.2 Data Flow/Asynchronous Mode

Von Neumann processors execute a program by fetching instructions sequentially through the program counter. The instructions then call the data they need from memory. Data flow architecture, Dennis [1980], on the other hand, has the data calling for instructions when the data become available as a result of previous calculations.

A limited version of the data flow is easily implemented on the MINNIE system. Local processors enable interrupts which will signal the arrival of data at their input registers. The interface (MUPI) sends an interrupt signal to its local processor to initiate a data driven sequence of events. Though careful consideration of programming a task the advantages of such a mode of operation can be derived.

(i) The interfaces are placed in a non-cyclic mode and interrupts are enabled. Some overheads are necessary to facilitate the use of processors of a von Neumann architecture whose actions are triggered by data driven interrupts. Ideally a separate interrupt line needs to be provided for each input register; however, this may be unpractical for a large number of input registers. A compromise interface design is to enable the processor to interrogate the read unit to determine the interrupt source, i.e. the input register number.

(i i) The interrogation of the input registers of the read unit would occur within the interrupt routine of the local processor. There are various operations that can be carried out if data is present at one or more of the registers. One such operation would be to transfer the data into a table in local memory, - otherwise it would be overwritten by other data coming in through the same input register.

The local processor would clear the data flags so that other data may arrive through the same registers if necessary. The processor would use data in the table when all required data elements for a specific operation or instruction have arrived.

(iii) All synchronisation of events within such a structure occurs with the structure itself, i.e. only operations or instructions that have a complete set of operands will be executed so there is no danger of the task over-running itself. As mentioned earlier MINNIE is not a purely data flow machine hence there will be unavoidable overheads which may degrade the performance of certain tasks which are implemented in this manner.

(iv) The estimated performance for this mode of operation will depend on the portion of the data communications overheads which is assumed to take place outside the computation time, i.e. those that cannot be absorbed or overlapped within computation. This effect is modelled as before through the parameter ϕ. Consider the case where almost total absorption can take place, say $\phi = 0.1$. The performance models are:

Global Memory System: $t_g = t_1/N + 0.1N(I+O)$ (3.26)

MUPI/MINNIE System: $t_m = t_1/N + 0.1NO$ (3.27)

A plot of these two equations with $t_1 = 100$, $I = 4$ and $O = 1$ is shown in figure 3.17. Although the gap between the two curves is smaller than in figure 3.7 (setting $\phi = 0.1$ implies almost total absorption of communications within computations) the superiority of the new system is very evident in achieving a minimum execution which is less than half that of an ordinary global memory system.

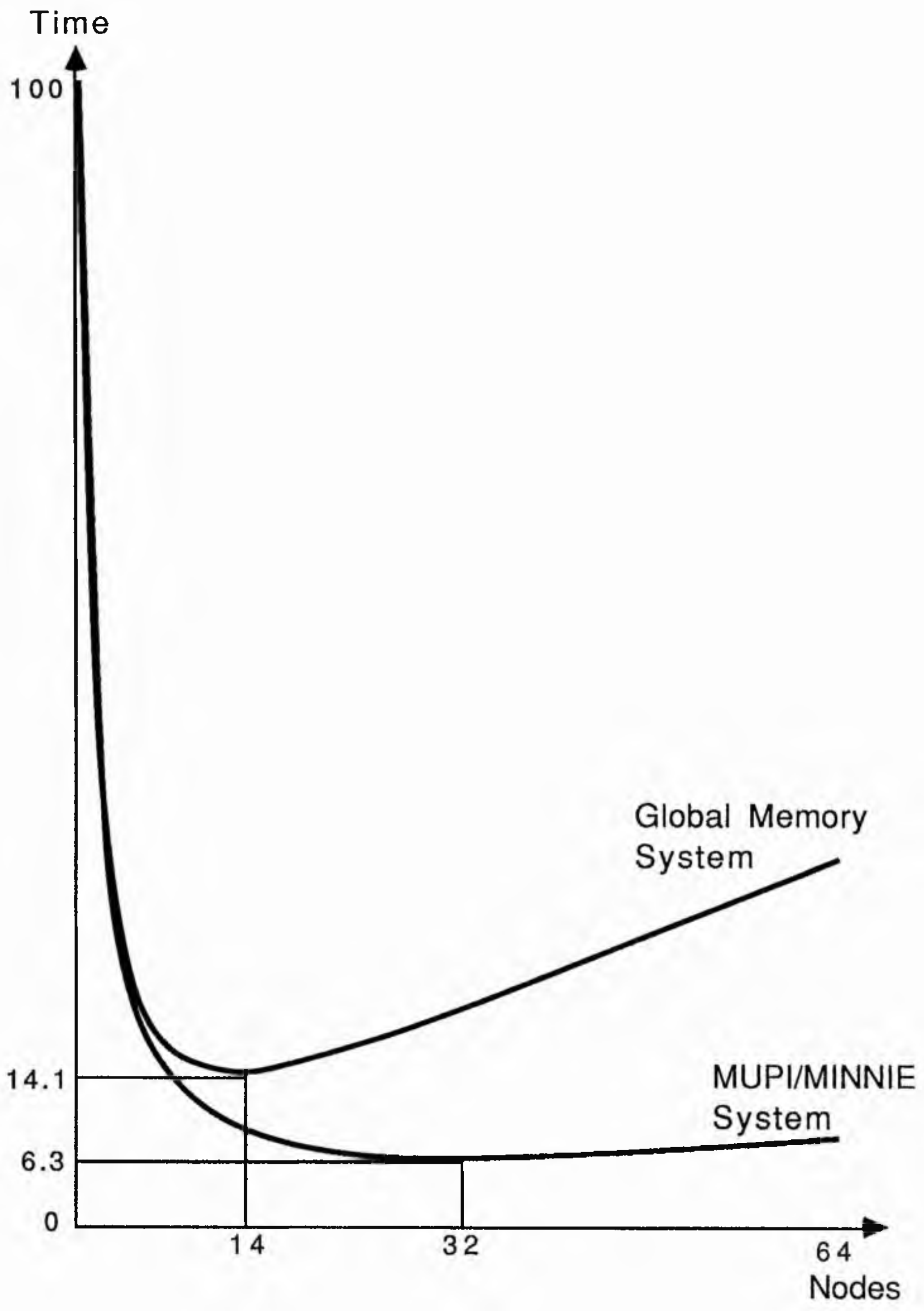


Figure 3.17 Performance models, $\phi = 0.1$.

3.5 Classification of the Interconnection Strategy

Flynns classification scheme introduced in section 1.2 does not fully encompase the interconnection strategy being employed in the newly devised system. The closest classification of Flynns that would help to classify the scheme would be the MIMD classification. Flynns classifications only specify the multitude of instruction and data streams and not the concurrency of these streams nor the interconnection strategies employed.

Artym and Mason [1988], have introduced a new scheme for interpreting the coupling techniques for processor systems, the XPXM/C taxonomy. With this taxonomy the method of processor coupling techniques can be classified with a greater degree of flexibility. The XPXM taxonomy is based on the multiplicity of simultaneously-active access paths between processors and memory modules, while XPXC is used to describe multiprocessor systems which do not employ shared memory.

XPXM is broken down into four categories:

SPSM:	single port, single memory,
SPMM:	single port, multimemory,
MPSM:	multiport, single memory,
MPMM:	multiport, multimemory.

It must be stressed that single and multi within the taxonomy do not refer to the number of coupled ports or memories but rather to the number able to support simultaneous activity.

Likewise XPXC can be broken down into its four component groups which are used for systems with no global memory, the groups are:

SPSC:	single port, single channel,
SPMC:	single port, multichannel,
MPSC:	multiport, single channel,
MPMC:	multiport, multichannel.

From these eight separate groups, the system specified earlier in this chapter does not entirely fit into one group only. The method of communications being employed is such that the number of channels used for communication purposes (simultaneously active channels) is dependent upon the layout of the input address registers within the read units of the interfaces. This is such that the specification of the system can change from task to task or within a single task if necessary.

Of the eight different groups of the XPXM/C taxonomy, MPSM and SPSC combined would represent the new system, i.e. the system contains a single multiport global memory but the updates occur over a single ported channel. It must be stressed that this classification does not effectively take account of the mode of communications associated with the updates over the single channel, i.e. the associative mode of addressing which reduces the amount of channel requests.

The XPXM/C taxonomy is such that it can be used as a good basis for comparison of multiprocessors, discussion of their relative parallelism, and help to identify common structures of the systems.

3.6 Block View of System

From the above system specification and the various modes of operation to be employed a series of block diagrams of the system to be developed can be derived.

The block diagrams shown in figures 3.18, 3.19 and 3.20 show the three main functional levels of the system specified above. The various levels are the MINNIE system as a whole, a single processing element and a block diagram of the MUPI.

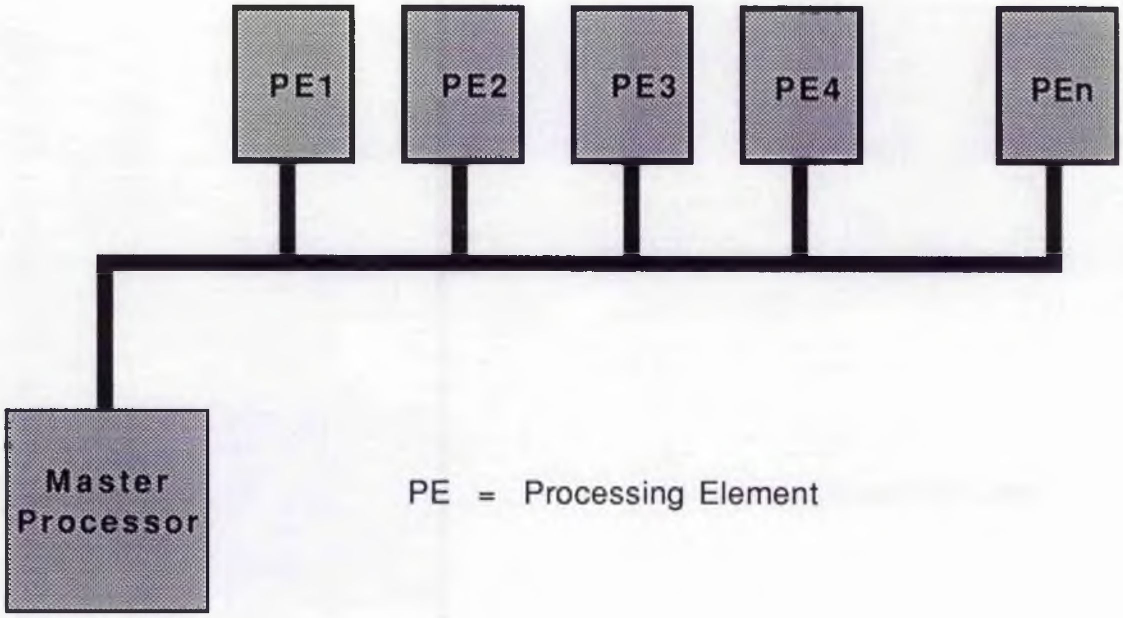


Figure 3.18 Block view of MINNIE.

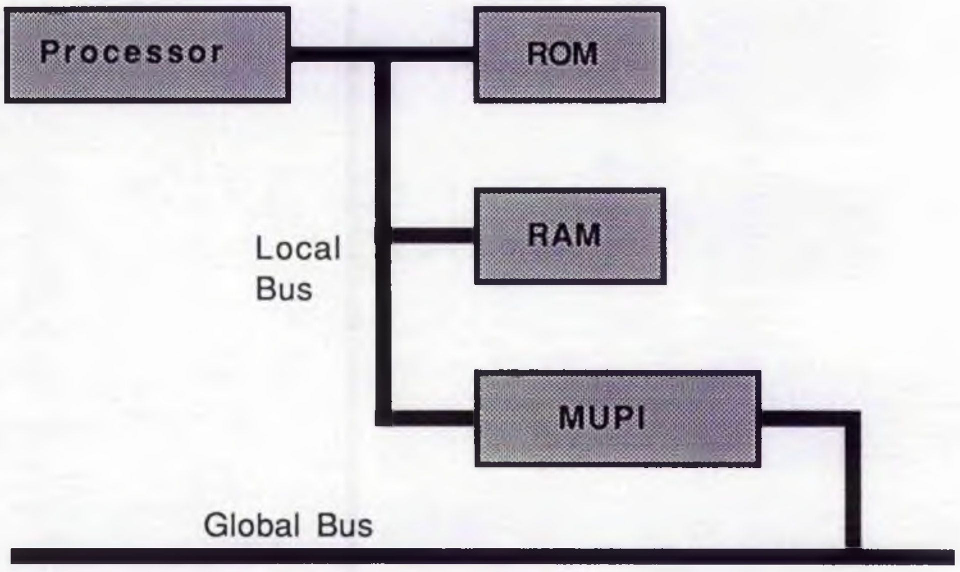
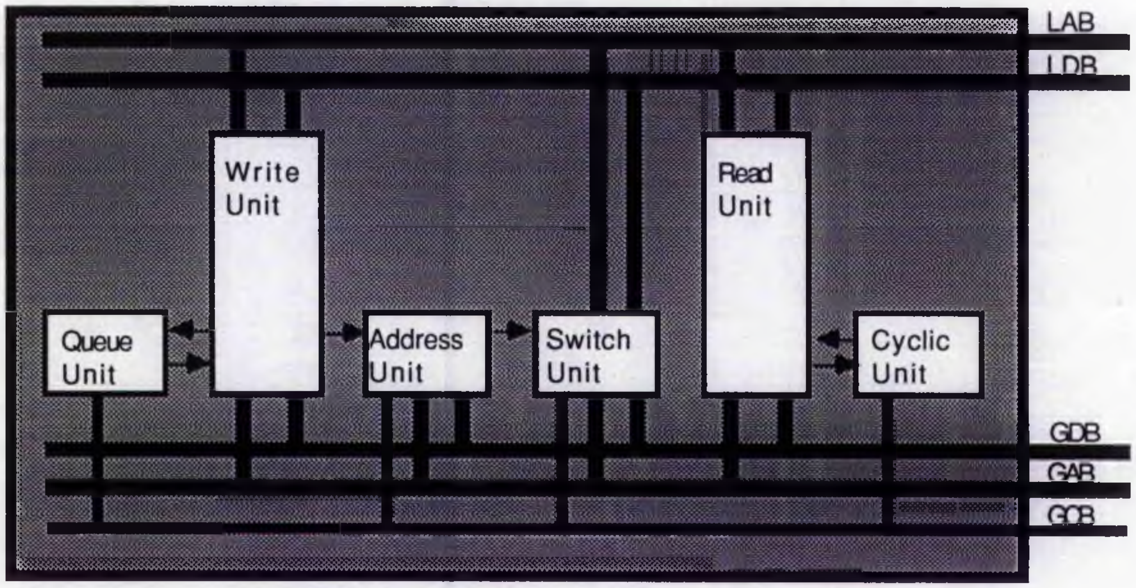


Figure 3.19 Block view of a PE.



GDB = Global Data Bus
GAB = Global Address Bus
GCB = Global Control Bus

LAB = Local Address Bus
LDB = Local Data Bus

Figure 3.20 Block view of MUPI.

CHAPTER 4

DEVELOPMENT AND IMPLEMENTATION OF THE MUPI/MINNIE SYSTEM

CHAPTER 4

DEVELOPMENT AND IMPLEMENTATION OF THE MUPI/MINNIE SYSTEM

The MUPI/MINNIE system is developed through two prototypes, the first of which is constructed to facilitate the testing of the associative addressing scheme of data transfers. The first prototype has various limitations imposed upon it as a full system at this stage would be difficult to realise and would result In a far more intensive testing scheme.

From the block diagrams in section 3.6 it can be seen that a Master processor is required to facilitate various control functions such as initialisation and program transfer. The role of the Master processor is taken by a development system, Windrush Model 8000 Design Centre, Windrush [1985]. Plugged within the rack of the Design Centre is a communications board (developed by the author specifically for this project) which provides the necessary connections to the MUPI/MINNIE system, this can be seen in figure 4.1 which shows a block diagram of the overall system.

The Design Centre executes a control program to perform the necessary steps in setting up the various processor nodes within the system and monitoring the execution of the task. An outline of the control program is given in appendix 1; it shows the various levels of the menu driven program, a brief description of the program is given in section 4.3.2.

The communications logic between the Master processor and the MUPI/MINNIE system is straight forward as it only employs a few buffers and registers for the bus and various control signals respectively. The transfer of blocks of program code is achieved by employing a memory paging system within the Master processor, i.e. the local memory of one nodal processor at a time is accessed within the memory map of the Master processor. The access is achieved by switching through to the local bus via the Master processor bus, the global bus of MINNIE and the MUPI switch logic.

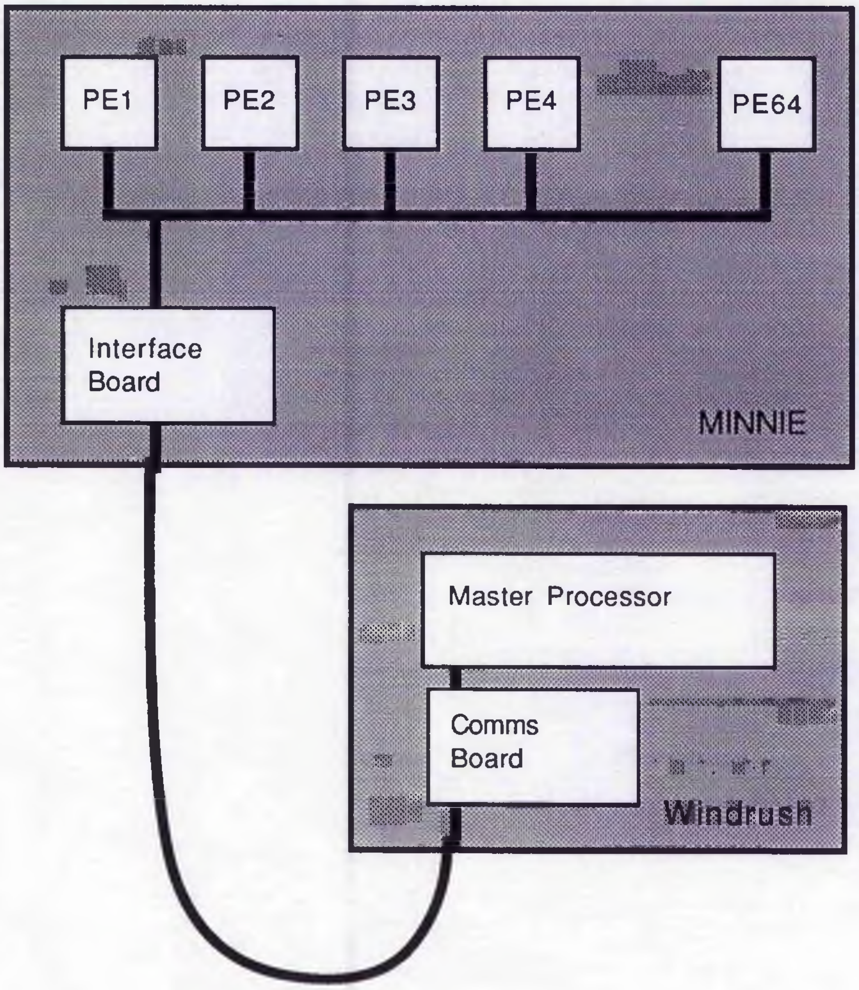


Figure 4.1 Block diagram of overall system.

Each node contains a monitor program in EPROM. The monitor provides the node with the necessary reset procedures and various sub-routines that may be required by the executing program. A brief description of the monitor is contained within section 4.3.3.

4.1 Four Node Prototype MUPI/MINNIE System

The four node system is developed with the following limitations imposed upon it so that testing and evaluation of the associative addressing scheme could be carried out;

- (i) only 4 nodes in MINNIE;
- (i i) a small number of Input and Output registers for each MUPI;
- (i i i) Cyclic mode of operation is not implemented;
- (i v) only 4K of local memory for each node, 2K ROM and 2K RAM; and
- (v) there is only a single level allocator signal for the bus allocation scheme.

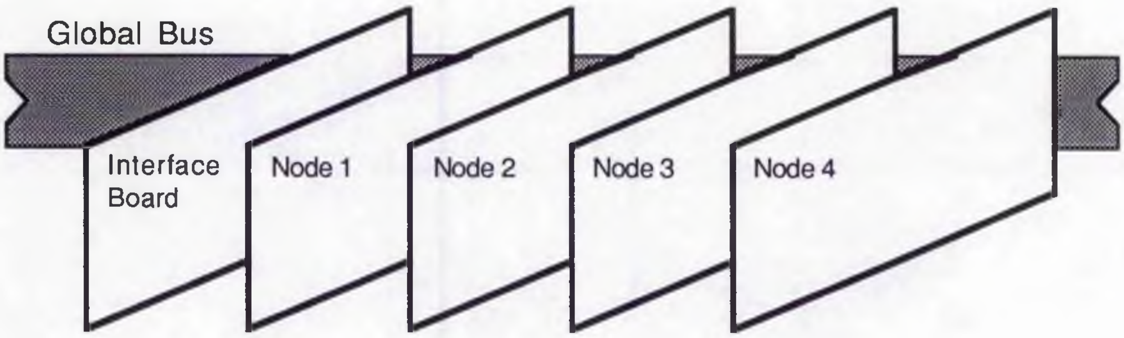


Figure 4.2 Four node prototype system.

The system is housed in a single eurocard height rack which contains four processor/node boards, one interface board and its own power supply, which can be seen in figure 4.2. The interface board provides buffering for the connections to the Master processor. Each processor board contains the following components which comprise a single node, see figure 4.3:

(a) 6809 processor,

(b) Asynchronous communications adaptor,

(c) 2K of static RAM,

(d) 2K of ROM,

(e) Prototype MUPI.

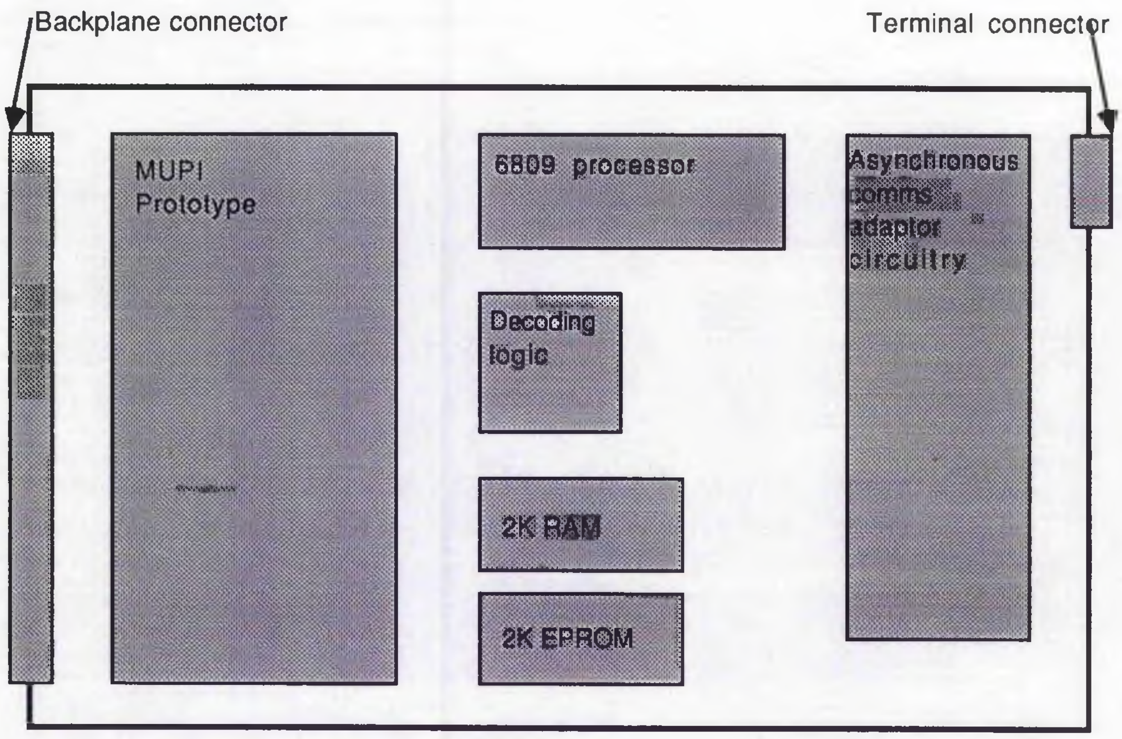


Figure 4.3 Single node components of four node system.

The prototype MUPI is a collection of a few components which are as follows, see figure 4.4:

(a) One Xilinx LCA chip,

(b) One 2K EPROM chip,

(c) Four 4x4 register file chips,

(d) five bus transceivers.

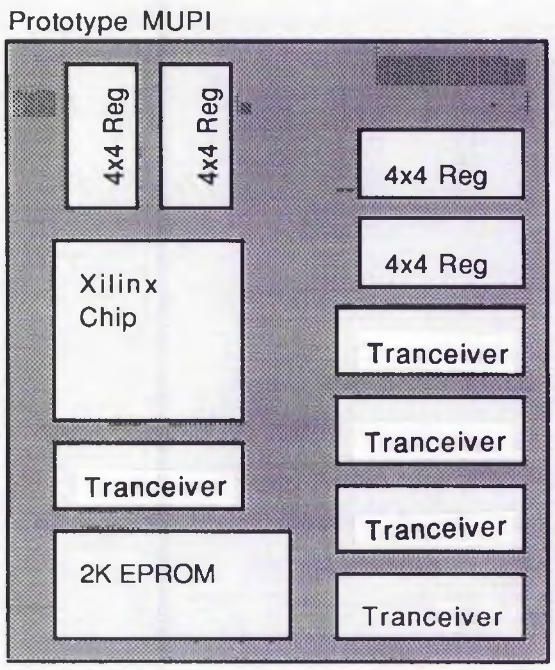


Figure 4.4 Components of prototype MUPI for four node system.

The Xilinx LCA (Logic Cell Array) device is in effect a programmable gate array as it provides the user with the ability to program the various gates and routes using a CAD package running on an IBM PC. The configuration of the gates and routes chosen is contained within the Eprom which downloads this data at power-up of the system. One of the advantages of the LCA is that it is very flexible in that a circuit can be changed in a matter of minutes. For more information concerned with the LCA refer to appendix 7.

As the LCA used has a limited number of pins that could be used for the circuit itself, a bus transceiver is added which isolates some of the address lines of the Eprom once configuration is completed giving rise to a few more pins which are used for various signals. Also, since there is a limit to the number of gates within the LCA a few register chips are added to act as both Input and Output registers of the MUPI.

All of the circuits were constructed on single height eurocards using wire wrapping techniques. Although the prototype is limited it can still be used to investigate the transfer of data by means of associative addressing techniques so that further modifications can be made to the design before full implementation.

<u>The NADE Signal:</u> Through various stages of testing using a logic analyser and a terminal connected to the asynchronous communications adaptor it was necessary to introduce a new control signal NADE (Node Address Data Enabled). This signal is needed as continuous errors will occur with associative addressing as the address comparisons are running continuously and a match may occur when an address is changing rather then when it is active.

The NADE signal works in much the same way as a chip select line on a memory chip, i.e. when an interface which has control of the global bus needs to send address/data it also sends a low NADE signal which tells the other nodes in MINNIE that the address/data are valid, see figure 4.5 for clarification.

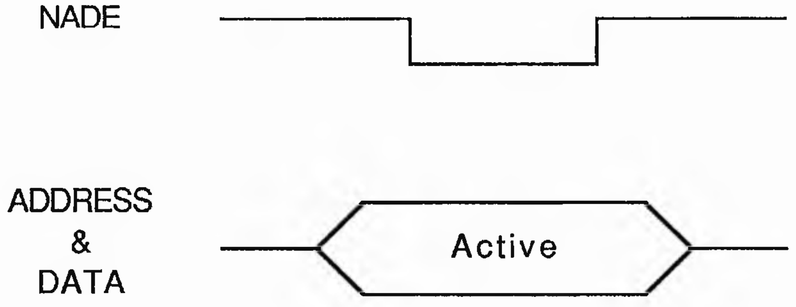


Figure 4.5 NADE signal.

The inclusion of the NADE signal was the only result of the testing stages of the first prototype, once the signal was introduced the transfer of data through the associative addressing scheme was proven.

4.2 The 64 Node MUPI/MINNIE System

With the associative addressing scheme proven through the first prototype it was decided to develop a substantial system which could form the basis of a computational machine to be used for computing the solutions of various parallel algorithms.

The structure of the 64 node system is not unlike that of the prototype 4 node system in that the Master processor behaves in the same manner and there is one single global bus with the same control lines. The topology chosen for the allocation system is deduced from the physical structure of the system. The system is constructed on 8 separate printed circuit boards each containing 8 nodes, a subsidiary circuit for allocation purposes and buffering to support the nodes. All 8 circuit boards are housed in a large rack with a communications interface board and a power supply, see figure 4.6.

The nodes are grouped in sets of eights giving 8 nodes on 8 PCBs. The PCBs were designed in sections using a PCB CAD package (Smartwork) such that only one node was layed out and copied. The layout of the board buffering and subsidiary circuits were added and the various connections between each node and these additions were made.

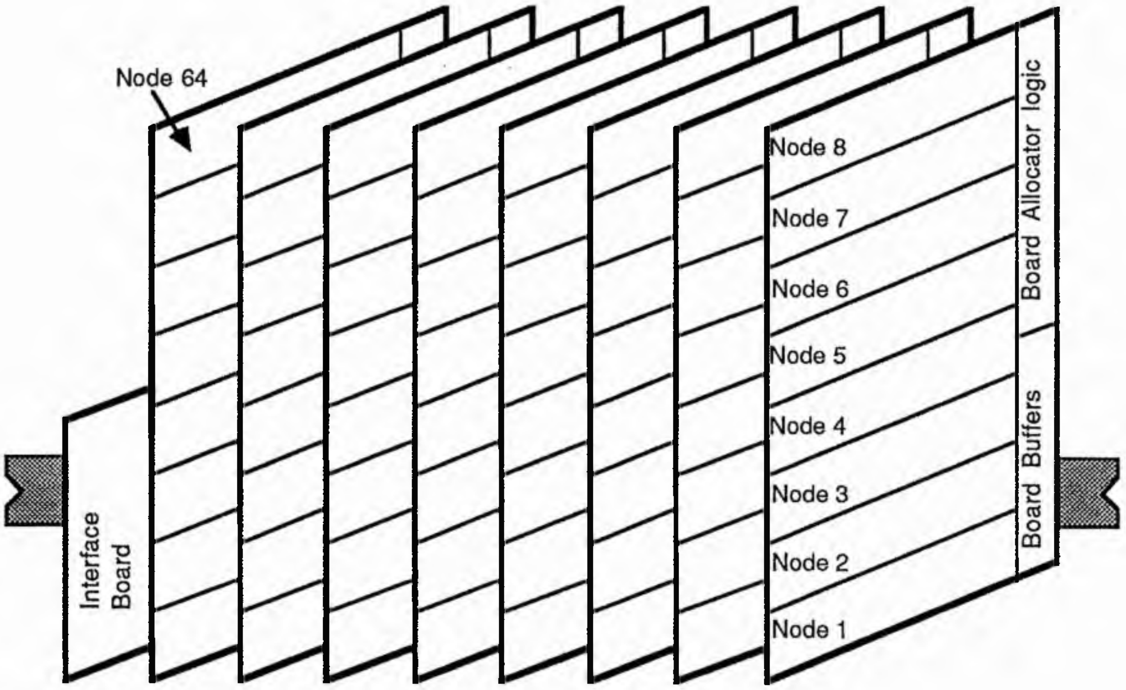


Figure 4.6 MINNIE rack.

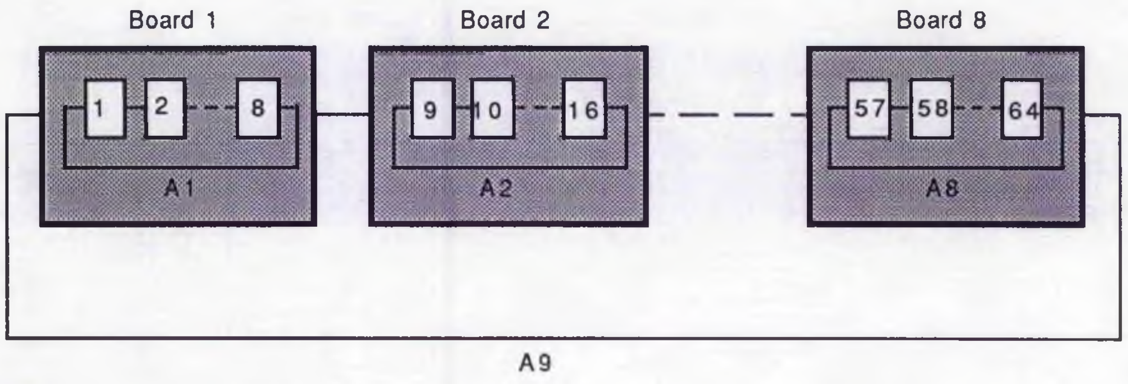


Fig. 4.7 Allocation system for 64 node MINNIE system.

As the nodes are physically split into groups of eight it seemed a logical solution to use this as a basis for the multi-level allocator system. The allocator is split into two levels, one level is for the rack itself and the other level is for each individual board, i.e. there are nine allocators in total as can be seen in figure 4.7. A further detailed description of the allocator systems signals is given in section 4.2.4.

The nodes as previously mentioned do not differ substantially from those in the 4 node system. Each node contains the following components:

(a) 6809 processor,

(b) 8K of static RAM,

(c) 2K of ROM,

(d) MUPI,

(e) Expansion port.

The expansion port is simply a connector which provides the necessary lines in order to connect further devices such as an asynchronous communications adaptor or memory expansion if necessary. The memory of the node is expanded to 8K of static RAM to facilitate larger portions of program code and data although only half of this memory will be accessible by the Master processor, refer to section 4.2.5.

In order to contain 8 nodes on a single PCB the MUPI has to be physically as small as possible, ideally a single integrated circuit. With this in mind the MUPI for the 64 node system was designed to fit within a single gate array device manufactured by MCE (Micro Circuit Engineering), designed using their CAD packages on an IBM XT, MCE [1985a, b, c].

The gate array available contains 1440 gates and 64 pins and provided the following constraints:

(a) Only 59 I/O pins available for use.

(b) Only 1050 gates usable.

Because of these restrictions several obstacles has to be overcome in order to fit the design within the framework provided. A brief design description of each unit in MUPI together with associated development problems will now follow.

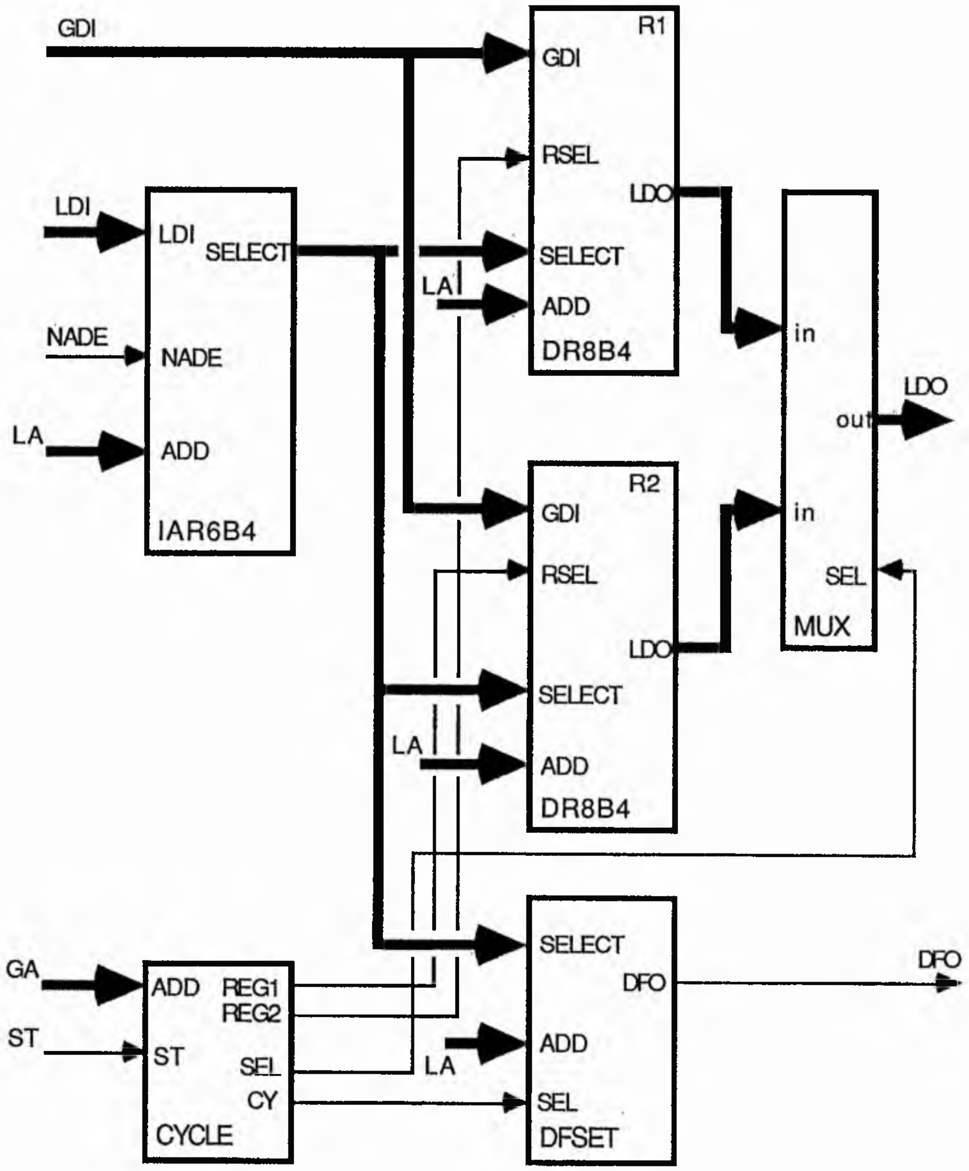
4.2.1 Read Unit

As with all other units within MUPI, this unit is developed with its block diagram in mind, (refer to section 3.3) as follows:

(i) The requirement is for a system with the ability to access two separate sets of registers under the control of a cyclic unit. There is also the need for a bank of input addresses which could be set up and verified by the local processor and used to interrogate the global address bus for matching addresses. The data associated with the matching addresses is stored in one of the data registers of one of the register banks, dependent on the state of the cyclic unit. Also, the local processor needs to gain access to one of the data register banks so as to read the data that has arrived from the global bus. The circuit was developed using the gate array development system (MCE BX software suite); the structure is shown in figure 4.8.

(i i) The data register banks, R1 and R2 are shown as DR8B4 (Data Registers 8 Bit x 4) i.e. there are only 4 input data registers in each bank. The restriction to 4 registers is due to the number of usable gates for the array.

The input address registers are shown as IAR6B4 (Input Address Registers 6 Bit x 4), i.e. four registers of length 6 bits. Only 6 bits are required to facilitate 64 unique addresses as each MUPI can only send 1 piece of data through its write unit; the write unit is fully described in section 4.2.2.



LA = Local Address, GDI = Global Data In, LDI = Local Data In,
LDO = Local Data Out, DFO = Data Flag Out.

Figure 4.8 Read unit circuit.

(iii) As specified in section 3.3.1 the read unit requires data flags which are associated with each input data register. These data flags are provided in the small DFSET (Data Flag SET) module of the read unit. There are in effect only 4 data flags which are associated with both banks of data registers as when the data flags are in operation only one data register bank will be active for both reading and writing.

The read unit operates in such a manner that if the cyclic unit is set for non-cyclic mode register bank R1 is active for inputting data from the global bus and also active for being read from by the local processor. This is done by the signals from the cyclic unit denoted by REG1, REG2, and SEL which enables R1, disables R2 and selects R1 for reading through the multiplexer. Also, the CY signal enables the data flags to be read and to provide interrupts when necessary.

(iv) In the cyclic mode of operation the register banks are alternatively selected and made active and the data flags are no longer active. Initially R1 is active for inputting data from the global bus and R2 is selected for being read by the local processor through the multiplexer. When the present cycle is completed the next cycle is initiated by an ST (STart) signal from the Master processor which restarts the system. This ST signal performs a swap within the cyclic unit such that R1 is no longer active to accept data from the global bus but rather R2 is and like wise R1 is now selected for reading through the multiplexer and R2 is not. The continuation of the cycles of the system will perform this operation within the cyclic unit until the Master processor resets the unit back to non-cyclic.

(v) The inputting of data from the global bus is performed mainly by the IAR6B4 such that this module recognises the various addresses or does not as the case may be. When the module does match an address (when NADE is true, see section 4.1) with one within its register bank it sends the necessary SELECT signal to the data banks and the data flag module. The data banks and the data flag module will input the data from the global bus and set the necessary flag depending on which of the modules are actively set by the cyclic unit as mentioned above.

(vi) All the modules within the read unit can directly or indirectly be accessed externally i.e. either by the local processor or the Master processor. These accesses are necessary to facilitate the various operations contained within the read unit. The accesses occur through the local and global address buses which are decoded to give internal address lines denoted by LA (Local Address) and GA (Global Address). The Input Address register for example is accessed by the local processor in order to set up the set of input addresses for the read unit through LA.

4.2.2 Write Unit

The Write unit requires three separate modules:

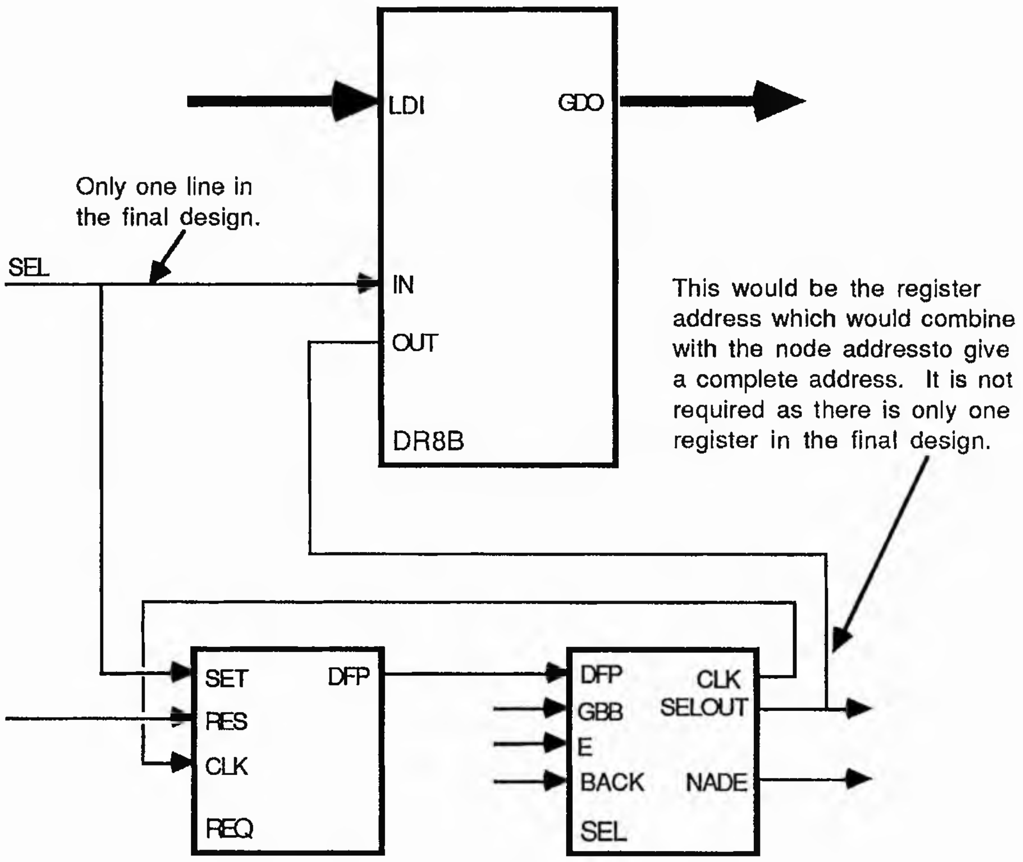
(a) a bank of output data registers which are used to store the data to be broadcast over the global bus;

(b) a request module which on receipt of a command from the local processor will send a bus request signal to the queuing unit;

(c) a selection module which receives a bus acknowledge signal from the queuing unit and performs the function of selecting the data registers in turn for outputting their data to the global bus with the relevant address.

In figure 4.9 the bank of data registers are depicted by DR8Bn (n x Data Registers of 8 Bits) and the other modules are shown as REQ (REQuest) and SEL (SELection).

(i) DR8Bn allows the local processor to write to the registers to store the relevant data for outputting to the global bus and for the selection module to control the outputting of the data to the global bus. The inputting of data to the module, through LDI (Local Data In), is achieved by the local processor performing a normal write operation to one of the registers which will be selected via the decoding logic through the Sel In lines of the module and verified as a valid address by the IN signal. Outputting of data to the global bus, through GDO (Global Data Out), is performed in much the same way as inputting, i.e. the relevant register is selected by the selection unit through the Sel Out lines of the data register module and confirmed by the CLK signal which performs the confirmation of a valid selection.



LDI
GDO
Only one line in the final design.
SEL
IN
OUT
DR8B
This would be the register address which would combine with the node addressto give a complete address. It is not required as there is only one register in the final design.
SET
RES
CLK
REQ
DFP
DFP
GBB
E
BACK
SEL
CLK
SELOUT
NADE

LDI = Local Data In, GDO = Global Data Out,
RES = Reset, CLK = Clock for Outputing, SEL = Select,
DFP = Data Flag Present, GBB = Global Bus Busy,
E = Local Processor Clk, BACK = Bus Acknowledge,
NADE = Node Address Data Enable, SELOUT = Output Selection.

Figure 4.9 Write unit circuit.

(ii) The REQ module is activated by the local processor during a write operation to the DR8Bn module as explained above. The REQ module receives the same inputs as the DR8Bn module so that a data flag associated with the active register can be set and a DFP (Data Flag Present) signal can be sent to both the selection module and the queuing unit to initiate the queuing process within that unit, refer to the queuing unit in section 4.2.4. The REQ module also provides the SEL module with all the data flags so that the SEL module can select one when necessary. The data flags within the REQ module are selected by the SEL module as mentioned and hence one of the data registers in DR8Bn is chosen as described above. The lines used to select one of the data registers are also used to reset the chosen registers data flag within the REQ module as shown in figure 4.9. Hence, the REQ module provides the SEL module with the necessary signals so that a choice can be made of the relevant registers. If necessary the data flags within the REQ module can be reset via a global reset signal, RES.

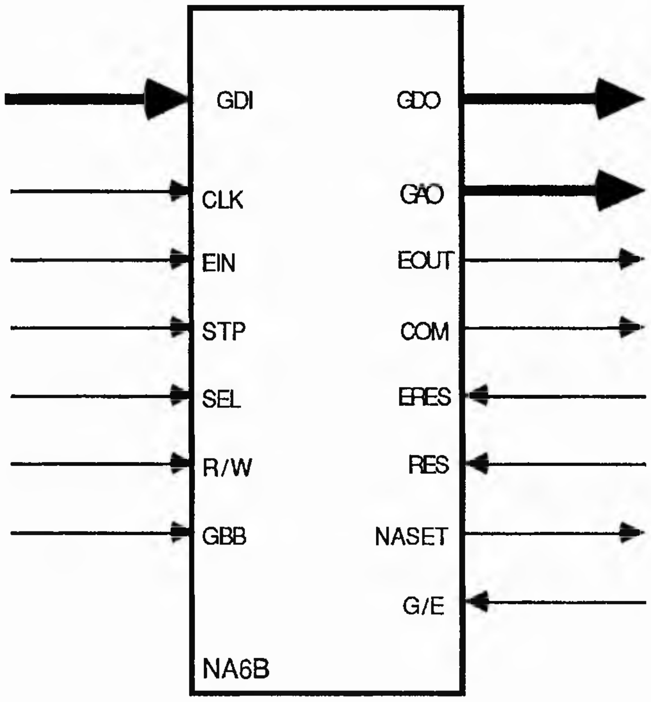
(iii) The SEL module is probably the most crucial of the three modules. It provides the necessary signals for choosing one of the registers for outputting its data, it outputs the address, and finally provides the important NADE signal which is necessary for the global data broadcast operation. The SEL module as mentioned earlier receives all the data flags as well as the DFP signal which initiates the whole process. Once the SEL module receives a BUS ACKnowledge signal from the queuing unit (see section 4.2.4) it performs the operation of selecting one of the data registers from its set data flag and in the process outputs the address associated with the register to the global address bus. The address outputted is a combination of the selection address sent to the DR8Bn module and the node address which is activated by the node address unit receiving the CLK signal from the SEL module. Once the data and addresses have been sorted by the CLK signal the SEL module sends out an active NADE signal which completes the process.

Since the data flag associated with an active register will be reset once the process of data transfer is commenced, i.e. when the clk signal is active, the SEL module will have the ability to select another register, if there are any remaining set data flags available as signified by the DFP signal. If any of the data flags are set then the process of outputting data is repeated. If there are no remaining data flags selected then the SEL module remains idle and the DFP signal is no longer active, which signals to the queuing unit that the global bus is no longer required.

(iv) In the final designs of the write unit for the logic array the number of output data registers are reduced to only one because of the shortage of available gates in the array. The operation and design of the unit is as explained above except for the removal of several selection lines as there is only one register to be selected. The lines that are no longer necessary are the SEL IN lines and the SEL OUT lines. The address which is outputted onto the global bus is now only the node address and not a combination as mentioned above, hence only 6 bits are required for the global address bus for data transfer purposes.

4.2.3 Address Unit

The address unit is closely linked with the write unit. The write unit requires the ability to tell the address unit when to output the contents of its address register onto the global address bus. Moreover, the Master processor needs access to set up the address, and the switch unit will require a signal from the address unit to confirm the switch address.



GDI = Global Data In, GDO = Global Data Out, GAO = Global Address Out, EIN = Enable In, EOUT = Enable Out, STP = Step for enable, SEL = Select, R/W = Global Read not Write, GBB = Global Bus Busy, COM = Node Address Comparison, ERES = Enable Reset, RES = Reset, NASET = Node Address Set, G/E = Global Clock Signal, CLK = Clock signal from Write Unit.

Figure 4.10 Address unit circuit.

The address unit is comprised of only a single module, NA6B (Node Address 6 Bits), see figure 4.10. A number of signals are grouped together to form the control signals for setting up the address. To enable the Master processor to set up the node address registers in each MUPI, it needs to distinguish each MUPI uniquely. The simplest and most apt method is to route a daisy chained signal from one MUPI to another. This is known as the Enable signal and is seen in figure 4.10 as Ein and Eout i.e. it enters the unit and exits it to facilitate the signal for the next MUPI.

(i) When the Master processor wishes to set up a series of addresses in various MUPIs it starts by sending a GBB (Global Bus Busy) signal which suspends all operations within the MUPIs in the system, if there is any activity at all. The Master then sets the Enable line and sends a STP (Step) signal to all the MUPIs which in effect clocks through the Enable signal by one unit which enables the first MUPI in the system for the Master to read and write to its address register. The Master now has the ability to read and write to the address register in a normal memory access fashion using SEL (Select) R/W and G/E (Global /E clock), also the global data buses GDI and GDO. When the Master has completed the task of setting up the address in the first MUPI it can then enable the second MUPI, if necessary.

(i i) The next MUPI in line would be enabled if the Master processor sends another STP signal which again clocks the Enable signal through by one unit. Hence, another address can be set up by the Master at this point. When the Master does not wish to set up any more addresses, i.e. the number of nodes needed for an application is reached, the Master would reset the Enable signal by sending a NERES (Node Enable Reset).

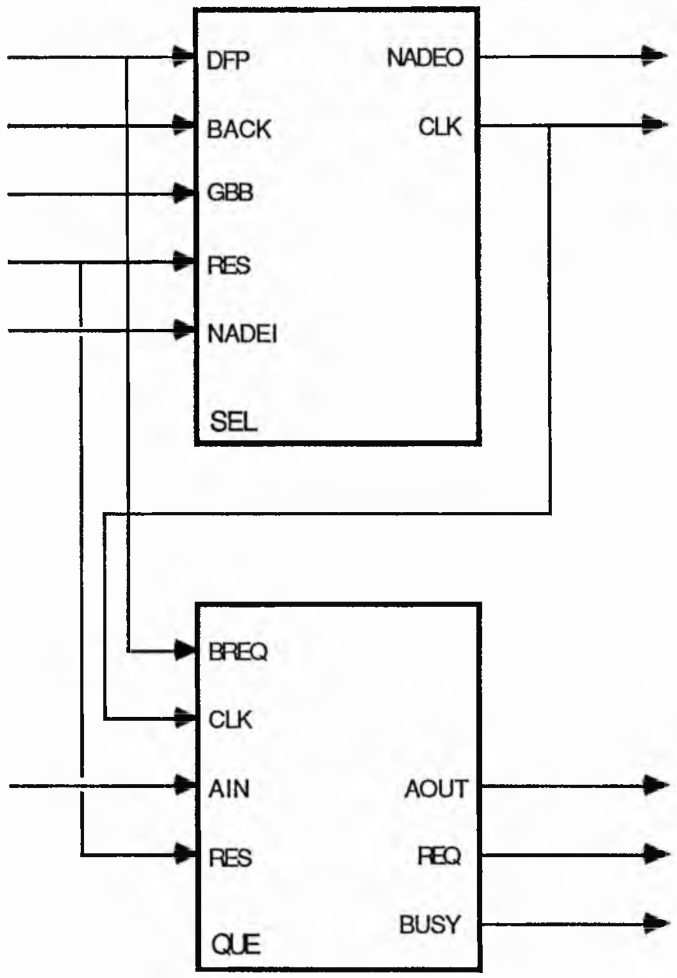
(i i i) When an address has been set up by the Master processor then the node that has been accessed now becomes an active node within the MINNIE system, and is indicated via a signal from the address unit, NASET (Node Address Set). The NASET signal is provided for the start/done module (STADON) which is responsible for providing the local processor with the necessary signal to start the computations, refer to section 4.2.6 for the peripheral modules of MUPI.

(iv) The address unit also provides the switch module with the node address comparison confirmation so that the global bus can be switched through to the local bus to enable the Master to send the code and data to the memory of the local processor. The address unit recognises the address in on the GDI lines and sends a comparison confirm signal if there is a match with the data within the address register. Again as with the read unit this comparison is continuous with a mask such that if the address has not been set up then there will be no positive confirmation.

4.2.4 Queuing Unit

The unit is developed as a single module which receives signals from the write unit to initialise its operation and also receives signals from the global bus to grant and confirm access of the global bus.

The unit receives two signals from the write unit, DFP and CLK which initialise and sustain the bus request procedure in turn, see figure 4.11. The DFP signal starts the queuing process and once the write unit is in control of the global bus when the DFP signal is no longer present the CLK signal holds the global bus for the remainder of the present broadcast cycle.



DFP = Data Flag Present, BACK = Bus Acknowledge,
GBB = Global Bus Busy, RES = Reset, NADEI = NADE In,
NADEO = NADE Out, CLK = Clock for outputting ,
BREQ = Bus Request, AIN = Allocator In, AOUT = Allocator Out,
REQ = External Request for Bus, BUSY = MUPI Busy .

Figure 4.11 Queueing unit circuit.

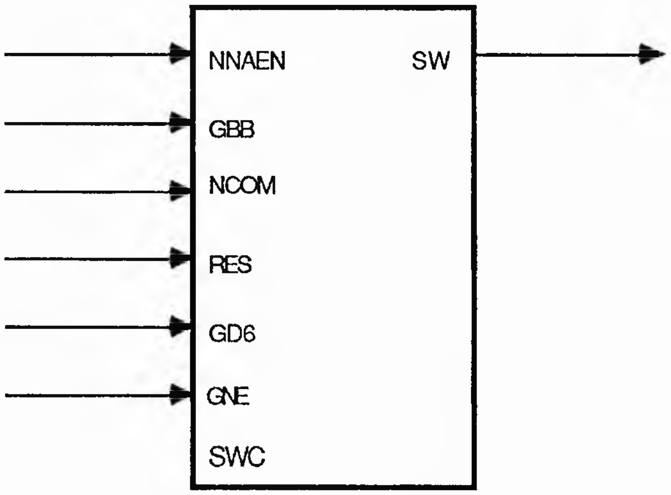
(i) When the queuing unit receives a DFP signal the unit waits for an allocator signal to arrive through AIN (Allocator In). Once AIN arrives the unit holds it, i.e. block its transfer through to AOUT unchanged as is the case if there were no DFP signal. The queuing unit will on receipt of AIN send an external bus request signal REQ to the queuing logic contained outside of MUPI.

(ii) The queuing logic contained outside of MUPI is required to facilitate the multi-level allocator system introduced in section 3.5. This external logic behaves in much the same way as the queuing unit itself. The external logic will receive only one request at a time depending on which node has control of the group allocator. The logic then waits for the system allocator and as with the queuing unit will contain this signal once it arrives. On the arrival of the system allocator the logic sends a common acknowledge signal to all of the MUPIs within the group. Only one of the MUPIs within the group will act upon the Acknowledge as the acknowledge is received by the queuing unit which internally gates this signal with the request signal sent out. This signal, which is the result of the gateing, is the BACK signal which is fed to the write unit to provide the bus grant.

(iii) When the write unit has completed its use of the global bus: (a) the DFP signal will no longer be active, (b) the CLK signal will become inactive, (c) which results in the queuing unit releasing the group allocator, i.e. letting the allocator continue through to AOUT and to the next MUPI in the group. The releasing of the allocator gives rise to the external bus request signal being inactive which results in the external queuing logic releasing the system allocator and dropping the acknowledge signal sent to the group. The allocators in the system, i.e. the group and system allocators, are now free to be selected by other nodes and groups respectively.

4.2.5 Bus-Switch Unit

The bus switch unit, figure 4.12, is required to allow the Master processor to switch the global bus to the local bus of only one node at a time to facilitate access to local memories. The operation of the unit is initiated by the Master setting the GBB signal so that it may obtain control of the global bus. Once the Master has control of the global bus it will write to the MUPIs with the relevant node address as data and the address such that it will be addressing all of the switch units at once.



NNAEN = Not Node Address Enabled, GBB = Global Bus Busy,
NCOM = Node Address Compare, RES = Reset,
GD6 = Global Data 6, GNE = Global clock, SW = Switch signal.

Figure 4.12 Bus-Switch unit circuit.

(i) The switch units in the MUPIs perform the required switch operation if and only if they receive a node address comparison confirmation from their respective address units and the 6th global data line is set high (1). When these conditions are true the unit will send a SWI signal to put the local processor into a halt mode, i.e. data and address buses put into high impedance state at the end of the present instruction. Once this has happened the unit will set internal switch signal (SW) to route the global address and data bus through to the local address and data bus with the direction of the data being dependent upon the state of the global read/write signal.

(ii) A number of control signals are required to interface the Master processor to local memory. These control signals are the R/W, /E, and /CS which are all sourced at the Master processor. They can be routed through MUPI but because of the limited number of I/O pins this option is not implemented. Instead they were routed with the identical control lines from the local processor through a multiplexor outside MUPI. The local processor provides a couple of signals to indicate its halt state, these lines are used to switch the multiplexor from either selecting the local or the global control lines through to the memory.

This method of connecting the control lines to the memory is also necessary as the array would otherwise add considerable delay onto the crucial control signals. The delays would cause conflicts with memory access.

(iii) When the Master processor is switched to the memory of a chosen node it performs normal read and write operations on it to set up program code and data. On completing this transfer it can either: (a) turn the switch off using the 6th data line, or (b) turn another MUPIs switch on by specifying another address. When switching another unit on, the active unit will be closed automatically as it will not receive a node address comparison confirmation.

(iv) When the switch unit of thė MUPI is turned off its local processor is returned to its normal state with an active reset. This reset signal is needed as conflict tends to occur because the switch unit does not recognise when the end of a present instruction has occurred and will hence interrupt it by switching the buses during the execution of the present instruction. With the reset introduced this will place the processor into a known safe state.

4.2.6 Peripheral Modules Within MUPI

Four peripheral modules have been developed to facilitate the operation of MUPI.

(i) Two of the modules, the local decoder and the global decoder, provide the necessary internal decoding of both the local and global addresses and set various selection lines to allow access to the main modules within the MUPI.

(ii) The third module is STADON (Start/Done) which provides the necessary data for the local and Master processors to interrogate. After initialising the various nodes in the system, i.e. address set up and code/data transfer, the Master processor sends a start command to the MUPIs within the system. The start command can either be in the form of a single line, i.e. a fast start, or an addressed location, i.e. a slow start.

The STADON modules within the MUPIs act upon this start command only if their respective node addresses have been set up. The STADON modules which are contained within active MUPIs set the start flag within their modules. A local processor interrogates its STADON module and upon recognising the start signal will commence execution of the portion of program code within memory. On completing the execution of its subtask a local processor will reset the start flag within its STADON module. When all of the start flags in the system have been reset, remembering that the inactive nodes contain reset start flags initially, the Master processor will receive an interrupt to signify the completion of the task.

(iii) The fourthmodule is actually a collection of modules which together facilitate the multiplexing of the various buses throughout the MUPI. The multiplexor modules provide the means for switching the buses to various locations when necessary. Multiplexors were preferred to internal tristate gates for connecting internal buses together as tristates would need more gates and be more complicated to simulate and test. The decoding necessary for selecting the appropriate paths through the multiplexors is carried out by the local and global decoders mentioned above.

4.2.7 General Array Development Features

The array was developed in such a fashion as to facilitate simulations at all levels. Each module was designed with a specific function in mind and simulated on its own. Because each module was simulated separately the reliability of each module could be confirmed and problems could be pin pointed accurately.

SIMULATION: With the simulations of the individual modules completed, the units which MUPI is comprised of were pieced together and simulated as a complete unit to confirm the interconnections between the various modules. Various problems were brought to light at this stage which mostly concerned the polarity of the various control signals which interconnected the various modules. The control signal which is sourced at one module may be active-high when the receiving module expects it to be active-low and vise versa. Once these minor problems were ironed out the units were assembled together to form the complete MUPI which was simulated as a single functional unit as it would be operating in the MINNIE system.

GATE COUNT: Simulations at this stage were successful in that they resulted in all of the functions operating as designed. A gate count at this stage showed that the total number of gates exceeded the number available for the array. Several modules within the system were redesigned to reduce their gate count by paying particular attention to the number of equivalent gates each functional gate represented. Module functionality remained the same but some output circuits were different in that they were designed using negative or mixed logic rather than positive logic as in the first design stage.

INITIALISATION: Various essential points had to be followed in order for the array to function properly. The most important that was noted throughout the design was that of initialisation; i.e. all of the modules would need to be initialised, especially those with memory elements. The initialisation was achieved through the global reset line which can be seen on the majority of the modules in the diagrams above.

LOADING and DELAY: The effects of signal loading were also noted in that the loading on a particular gate resulted in an increased propagation delay through that gate. The increase in propagation delay may result in the function of the gate not behaving as designed if the timing is a critical factor. During simulations the simulator used (MCE BXSIM) provided the option of simulating with various delay factors imposed on the array which would take in account the fluctuations in operating supply voltage, temperature and varying batches of silicon used for the production of the array.

PRODUCTION: Once the array was fully simulated the data concerned with the array and a full list of the test patterns used for simulation was sent to MCE for production and post production testing purposes. The layout of the array was completed at MCE with no post layout simulation. The technology used for the array (5 micron CMOS) is such that the capacitive loading of the routing does not impose a substantial delay on the gates which would alter the functionality of the array.

When the final production parts were completed at MCE the arrays were subjected to intensive testing using a functional tester which fed the array with the test patterns used in the simulations. The tests at this stage resulted in a quality of prototype devices which were implemented in a test board. Details concerning the characteristics of the array and a full set of circuit plots of the array can be found in the Customer Procurement Specification Document CPS [1988].

The test board that was developed was a two node system which would be able to test the complete functionality of the array. The test board was such that it slotted into the rack of the Master processor and all of the buffering between the two was contained on the test board.

The test board resulted in the array being passed as a functional device which could be used in the construction of the proposed 64 node system.

4.3 Development Support Facilities of MUPI/MINNIE

With the development of the parallel systems mentioned above there needed to be some means to support them. The means of support were comprised of two interface circuits which interfaced the main system with the Master processor and two sets of software which helped to control the system. The software is divided into two separate sections concerned with the main control program residing in the Master processor and a node controller in the form of a monitor.

4.3.1 Interface Circuits For MINNIE

The two interface circuits were designed separately as one would sit in the rack of the Master processor while the other would reside in the rack of the MINNIE system. The interface circuit which was contained within the Master processor provided the means for the Master to address the various control lines of the MINNIE system and also provided the buffering for the address and data buses of the Master to both buffer and isolate them from the global buses of the MINNIE system when necessary. The board which contained the circuitry is a half height Eurocard which slotted into the backplane of the Master rack and provided a link to the MINNIE interface board via a connector.

The MINNIE interface design is such that there is no intervening circuitry between the incoming 34 way D connector and the MINNIE backplane. The boards main function is to provide two RS232 connections to facilitate connections of terminals to any two nodes within the system. The RS232 circuits are such that a connection could be made between one of these circuits and any chosen node through the expansion ports provided on the main circuit boards.

A set of circuit diagrams of the interface circuitry can be found in appendix 4.

4.3.2 MINNIE Control Program

The Control program is a menu driven program which resides on the Master processor and gives the user full control of all aspects of MINNIE. The program was developed to provide a support facility in the development of the parallel system.

The program allows the user to reset, initialise the node addresses, access the local memories, set cyclic mode, and start the system. All of these operations are possible with the use of single key commands except where specific node addresses, file names and memory locations have to be specified. Adetailed description and listing of the program can be found in appendix 1.

4.3.3 Node Monitor Program

The node monitor program was developed to facilitate the connection of a terminal to any individual node in the system as well as providing various subroutines that a subtask running on a node may require.

The monitor was written along the lines of a basic monitor providing various commands such as memory dump, memory examine and change and jump to user program. There is also the option to return control to the monitor rather then through the keyboard to allow the node to function normally. A description of the program together with its listing can be found in appendix 2.

4.4 Concluding Remarks Concerning Development

The development of the 64 node system concerned a large number of different aspects. The array, the most important component of the system, was developed with the operational facilities of the system in mind, while the remaining components were added to the array to produce the complete system.

The circuits developed for the nodes can be found in appendix 3 which show the details of a single node of the system and the associated control logic for the printed circuit boards such as buffering and system allocator circuitry. It should be noted that the circuits have not been explained in detail as it is assumed that the reader is familiar with basic microprocessor systems. A number of points deserve a mention, however: The global reset signal which is sourced at the Master processor resets each MUPI in the system as mentioned previously; but, the reset signal also resets each processor in the system at the same time which ensures that all processors are in a known state. Also included in the node processor circuit is the fast start signal which is fed to all of the processors as well as all of the MUPIs to allow the system to be restarted quickly in the cyclic mode.

The various support circuits and software were necessary in the initial stages as they provided the means to test and evaluate the various aspects of MUPI and MINNIE; while in the latter stages of the development the support components tended to become part of the system thus simplifying its use greatly.

CHAPTER 5

EVALUATION, RESULTS AND DISCUSSIONS

CHAPTER 5

EVALUATION, RESULTS AND DISCUSSIONS

In evaluating the MUPI/MINNIE system several timing measurements are taken which are crucial to various aspects of the system. The timings taken are generally concerned with the passing of data over the global bus and the allocation of the global bus.

As well as timing measurements, several applications are implemented which emphasise the Cyclic mode of operation.

5.1 Measurements of Timing Parameters

The parameters that are critical in the operation of the MUPI/MINNIE system are those concerned with the transfer of data between nodes, the delays associated with allocation of the global bus, various initialisation processes and online support via the Master processor.

Initialisation: The initialisation of the system is a once only overhead i.e. the system is only initialised once for each separate application. The initialisation procedure includes the resetting of the system, the setting up of node addresses and the passing of sections of program code and data to the nodes. A detailed description of a typical master program and listing used for initialisation is shown in appendix 1.

The time taken for the Master processor to initialise a full 64 node system is calculated from the node address setup time and the time taken to pass a 2K block of code/data to each node.

Node-Address: Using a timer connected to the Master which is started when the process of setting up the node addresses begins and stopped when this process is finished a time of 12mS was measured for 8 nodes having their node addresses setup with a verification step included. This gives an average time of 1.5mS for each node address setup.

Program and Data: Similarily the time taken for the Master to pass a 2K block of code/data to any one node was measured in much the same manner i.e. a timer being activated/deactivated by the Master processor. The time measured for passing the code/data to one node was found to be 2.6 seconds,- a long time period as the operations involved include a hard-disk access by the Master processor to retrieve the data to be passed.

Using the above timing results the total time taken to initialise a full 64 node system is 2 minutes 46.5 seconds.

Start/Re-start: As well as engaging the system initially the Master processor also provides online support i.e. the Master will start the system and restart it when and if necessary. A restart is needed if the system is operating in the cyclic mode to execute the next cycle. The overheads associated with the start function are negligible but those associated with the restart function are of particular importance. Consider a task running in the cyclic mode of operation, when a cycle is complete the Master is told so and will restart the system. If the number of cycles is large and the time taken to restart the system is comparable with a single cycle execution time then the overheads grow rapidly with the number of cycles. For a full description and detailed listing of the restart procedure refer to appendix 1.

The time taken by the Master processor to restart the system from first recieving the done signal was measured to be 52.5mS with the Master using the normal start operation of gaining control of the global bus and writing to the start registers. There is a second start/restart option which was included in the design: it is a single signal common to all the MUPIs in the system and only requires the Master to pulse it without having to gain control of the global bus, this is known as Fast Start. The time taken to achieve a Fast Start was measured and found to be 52.5mS. It can be seen that the Fast start option is of no greater advantage at the moment as the actual procedure of restarting the system is achieved by an interrupt routine within the Master. What is needed is a circuit which can be programmed to facilitate the interrupt routine.

Global Bus Access: One of the most critical aspects of system operation during program execution is the time taken to gain access of the global bus. A selection of timings have been measured which give the best and worst case time durations for a node wishing to gain control of the global bus. The signals that have been measured are the two separate allocator signals and the time taken for a node to use the bus, i.e. the time the allocator is held by a node which is using the bus. The times are given as follows:

(i) Let τ_s be the time taken for the system allocator to propogate through one pseudo-node. This was measured to be $\tau_s = 1uS$.

(i i) Let τ'_b be the time taken for a board allocator to propogate through one idle node (not requiring the global bus). This was measured to be $\tau'_b = 0.6uS$.

(i i i) Let τ_b be the time taken for a board allocator to propogate through a busy node (one which wishes to use the global bus). This was measured to be $\tau'_b = 2.15uS$ (includes bus use time).

(i v) Let τ_w be the time that a node has to wait before it gains access to the global bus. The minimum value for this time is when the node is granted the use of the bus as soon as it is requested, i.e.

$$\tau_{w,min} = 0 \tag{5.1}$$

(v) The worst case for gaining access is when the node has to wait for all other 63 nodes in the system to complete their use of the global bus, i.e. the system allocator will propogate from one pseudo node to another each time a node is finished with the global bus, giving:

$$\tau_{w,max} = 63(\tau_s + \tau_b) \tag{5.2}$$

or, $$\tau_{w,max} = 198.45\text{uS}$$

However, in some sense the worst case for a node to gain access is not completely negative, as it means that the global bus is being fully utilised. But, as it is unlikely that all of the nodes in the system will want access at the same time, the worst case situation can be regarded as that when only one node requires access and the allocators in the system have to travel their maximum signal length before this request is granted. With this situation the worst case timing can then be given by,

$$\tau'_{w,max} = 7(\tau_s + \tau'_b) \tag{5.3}$$

or, $$\tau'_{w,max} = 22.05\text{uS}$$

Equation 5.3 is arrived at by considering the board allocator and the system allocators separatly; i.e. the board allocator has to pass through a maximum of 7 nodes before it can service the request; similarily the system allocator has to pass through a maximum of 7 pseudo nodes before the allocator can service the request. This worst case situation can in some sense be seen to be more undesirable as it indicates that the global bus is not being utilised when it is needed. Suggested future developments of the allocation system are given in chapter 6.

Discussion: As mentioned above the time taken for a node to complete its use of the global bus was measured to be 2.15uS. To increase the throughput of the global bus this time will have to be reduced, by reducing both the global bus cycle time and the allocator propogation time.

At the moment the timings taken are to broadcast of one byte of data, while the design of the interface is such that future developments can incorporate increased data transfer capacity, i.e. the 2.15uS used to broadcast one byte will not double as the number of bytes doubles, - it will be less than that as the allocator overhead is only present once.

In broadcasting multiple byte data, say 3 separate bytes, the time taken to do so would be a function of two separate factors: the time taken for the MUPI to broadcast one byte over the global bus, and the time taken for the MUPI to select a new register for data transfer purposes. These two factors may be overlapped to some degree to reduce the time taken, i.e. while a piece of data is being broadcast the MUPI can be selecting the next piece of data. These factors must be considered in future develpments of the system.

5.2 A First Order Vector Non-linear System

An application has been chosen which emphasises the benefits of MINNIE when operated in the cyclic mode. The application was manually coded and is not the result of parallel compilation of a serial program as envisaged for future applications. The development of a parallel compiler forms a sister project which is nearing completion here at Trent. The compiler would automatically detects parallelism within a sequential program and produce code for the MUPI/MINNIE system. A detailed description and listings of the various node programs are shown in appendix 8.

This application has been devised to demonstrate the advantages gained from employing the associative addressing scheme. The application is not a specific one but such a system of equations may occur in the real world, Whiting et al.[1975].

$$X_1(T+1) = X_1(T) + 10 - X_3(T)\,\mathrm{Cos}[0.07\,X_2(T)] + X_4(T) \tag{5.4}$$

$$X_2(T+1) = X_2(T) + 22 - X_3(T)\,X_4(T)\,\mathrm{Sin}[0.03\,X_1(T)] + X_1(T) \tag{5.5}$$

$$X_3(T+1) = X_3(T) + X_1(T) + \mathrm{Sin}[0.07\,X_2(T)] \tag{5.6}$$

$$X_4(T+1) = X_4(T) + 17 - \mathrm{Sin}[0.07\,X_1(T)] \tag{5.7}$$

$$X_5(T+1) = X_5(T) + \mathrm{Sin}[X_3(T)] + \mathrm{Cos}[X_4(T)] \tag{5.8}$$

$$X_6(T+1) = X_6(T) + X_2(T) + \mathrm{Sin}[0.05\,X_2(T)] \tag{5.9}$$

$$X_7(T+1) = X_7(T) + 0.02\mathrm{Cos}[\,X_3(T)] \tag{5.10}$$

$$X_8(T+1) = X_8(T) + X_1(T)\,\mathrm{Sin}[\mathrm{Cos}[0.09\,X_1(T)]] \tag{5.11}$$

Each equation is a function of its own previous value as well as the previous values of other equations (refer to problem formalism section in chapter 3).

The system of equations was programmed for three different system configurations to be computed over 256 cycles.

(a) 1 node Asynchronous.

(b) 4 nodes Cyclic.

(c) 8 nodes Cyclic.

These three configurations were chosen to illustrate the advantages gained when the number of processors are increased in the cyclic mode.

The single node configuration was also executed in the cyclic mode so as to determine the overheads associated with this mode. Although the overheads of the cyclic mode can be evaluated from the measured cyclic restart time, confirmation of the overhead figures would not be inappropriate here.

The following sections explain the programming of the system of equations as well as the results obtained.

5.3 One node evaluation

The system of equations were programmed in a normal sequential manner contained within a loop to be repeated 256 times. The results from the equations for X_5, X_6, X_7, and X_8 are stored in an array within local memory as these are the results equations. The input registers of the node's MUPI were not setup as there is no global communications. A detailed description and listing of the node programs are given in appendix 8.

This configuration was timed at 23.5 seconds for the execution of 256 values. The timing was achieved by using a timer/counter connected to the Master processor which recieves a DONE signal from the local processor being utilised.

To confirm the timing of the asynchronous operation the program was alterred so as to compute only one equation at a time, i.e. there are now nine single node programs, one for all eight equations and one for each equation alone. From the execution of the individual equations each equation was timed separatly. The results are shown in table 5.1.

EQUATION	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	TOTAL
TIME (sec.)	2.4	3.1	2.3	2.3	4.7	1.6	1.4	6.1	23.9

Table 5.1 Timing results for individual equations.

From table 5.1 it can be seen that the sum total of the individual equations is approximatley 400mS longer than all eight equations computed together. This difference is due to the overheads associated with the computation of each equation in its own separate loop of 256 and the storage of all values in local memory, not just those associated with the output equations.

To confirm the overheads associated with the cyclic restart mechanism the single node configuration was executed in the cyclic mode. This required removing the loop within the program only, as the Master processor would now control the number of cycles/calculations.

With this mode of operation the total execution time was timed at 37.9 seconds which is 14.4 seconds longer than the asynchronous configuration. From these figures the amount of time wasted during a cyclic restart was calculated as 56.25mS which is comparible to the 52.5mS measured.

It can be seen that over the 256 cycles the cyclic restart overhead of 14.4 seconds is unacceptable as compared with the cyclic execution time of 37.9 seconds, - it is over a third of the total time.

This only emphasises the fact that the cyclic restart mechanism should be totally implemented in hardware, this would reduce this overhead by at least a factor of 100 if not totally diminishing it except for a single TTL gate propogation delay which is typically 20nS.

5.4 Four node evaluation

With the four node configuration the equations were allocated to the nodes in a manner such that each node had an intermeadiate equation, either X_1, X_2, X_3, or X_4, and a results equation, either X_5, X_6, X_7, or X_8. So as to achieve as close to a balanced system as possible, i.e. the computations of each node being as small and as equal as possible, the single node timings in table 5.1 were taken into account to determine which equations should be paired together.

Node Address (HEX)	Equations	Total Time (sec.)	Input Register Addresses 00	01	10	11
00	X_1, X_6	4.0	01	02	03	- -
01	X_2, X_7	4.5	00	02	03	- -
02	X_3, X_5	8.0	00	01	03	- -
03	X_4, X_8	8.4	00	- -	- -	- -

Table 5.2 Equation mappings for a 4 node system.

The mapping of equations and Input Register Addresses can be seen in table 5.2. With the pairs of equations in the table are a set of times which are the sums of the pairs individual times from table 5.1, these times will help to give some indication as to the total execution time for the four node system. A detailed description and listing of relevant programs are given in appendix 8.

Input Register Addresses: Consider the equations for node 00. Equations X_1 and X_6 require between them data from equations X_2, X_3, and X_4 which are located at nodes 01, 02 and 03 respectively. Hence, the first three Input Address Registers for node 00 contain addresses 01, 02 and 03 while the fourth Input Address Register contains no specific address as this is not used.

The Input Address Registers are set up by each local processor. The local processor simply writes the necessary address to a particular memory location which is contained within its MUPI. In this particular design the location of the IVAs in the local processors memory map are locations $A00C through to $A00F, see appendix 5 for the complete memory map of the local nodes.

The local processor will have within its local memory a list of IVAs which correspond to the code to be executed by that node. Both the program code and the IVAs are set up in the local memories by the Master processor during initialisation. Hence, associated with each node program is a relevant data file which contains the IVAs for the node. The data file is just a simple list of addresses, usually four although there may be more, see chapter 6 for the use of more than four IVAs.

Execution: After initialisation the Master processor sends a START command to all nodes. All active nodes, i.e. those nodes whose node address has been set up, will have their START latch set. While this start procedure is occuring each local processor is constantly reading its respective START location within its MUPI. If, and When, a node recieves an active START signal from the START location the local processor will initialise its IVAs. Initialisation of IVAs by the local processor is a simple task of transfering four bytes of data from local memory to the IVA registers in its MUPI. The local processor then jumps to the portion of program code to be executed. On completion of the portion of code specified the local processor will write a DONE to its MUPI which will transfer it to the Global Bus.

Old and New Values: As introduced in section 3.4.1 the cyclic mode of operation utilises two sets of Input Data Registers the first of which holds data from the previous cycle and the second holds data from the present cycle.

The set of registers which hold data from the previous cycle can only be accessed via the local bus for reading purposes only, where-as the set of registers which hold data from the present cycle can only be accessed via the global bus for writing purposes.

At the start of a new cycle the two sets of registers are effectively swapped round so that the local processor can gain access to the previous cycles data, i.e. cycle T-1, where as the set of registers which contain data from cycle T-2 are swapped round to be accessed via the global bus to recieve data associated with the present cycle, cycle T.

Results: With the four node configuration operating as outlined above the total execution time was timed as approximately 22.9 seconds for 256 cycles.

If the wasted time associated with the cyclic restart mechanism, evaluated from the single node asynchronous and cyclic configurations, is subtracted from the total execution time for the four node configuration this would leave a time for computation and communications. Hence, (22.9-14.4) gives 8.5 seconds for the computation and communications parts of the task performed. This figure can be compared with the summed total execution time for equations X_4 and X_8, allocated to node 03, which is given as 8.4 seconds, see table 5.2. This comparison is justified as node 03 contains the largest computational part of the task and will hence complete its portion last.

The comparison of the times given above re-emphasises the fact that there are excessive overheads associated with the cyclic restart mechanism, it also shows that the total execution time on a multi-node system is dependent on the node with the largest computational part,- assuming that there are no holdups due to communications in the form of forced synchronisation which may put some processors in a wait state while waiting for data.

5.5 Eight node evaluation

For the eight node configuration each equation was allocated to a separate node. The mapping of equations and Input Register Addresses are shown in table 5.3.

Node Address (HEX)	Equation	Total Time (sec.)	Input Register Addresses 00	01	10	11
00	X_1	2.4	01	02	03	- -
01	X_2	3.1	00	02	03	- -
02	X_3	2.3	00	01	- -	- -
03	X_4	2.3	00	- -	- -	- -
04	X_5	4.7	02	03	- -	- -
05	X_6	1.6	01	- -	- -	- -
06	X_7	1.4	02	- -	- -	- -
07	X_8	6.1	00	- -	- -	- -

Table 5.3 Equation mappings for an 8 node system.

The implementation procedure for the eight node configuration is much the same as that for the four node configuration, the only difference being that eight nodes rather than four nodes need to be initialised. A detailed description and listings of all node programs are given in appendix 8.

The operation of the eight node configuration as mapped above in table 5.3 resulted in a total execution time of 20.6 seconds for the 256 cycles.

If, as for the four node configuration, the wasted time associated with the cyclic restart mechanism is removed, this would give a time of 6.2 seconds for the computation and communications parts of the task performed. From table 5.3 it can be seen that the execution time for equation X_8 is the longest at a time of 6.1 seconds which is comparible with the 6.2 seconds calculated above for the computation and communications parts together.

5.6 Discussion of results

The results from the 1,4 and 8 node configurations are shown in table 5.4 along with the execution times less the cyclic restart time.

	1 Node Asynchronous	1 Node Cyclic	4 Nodes Cyclic	8 Nodes Cyclic
Total Execution Time (sec.)	23.5	37.9	22.9	20.6
Total Execution Time less Cyclic Restart (sec.)	- - - -	23.5	8.5	6.2

Table 5.4 Results for the First Order Vector Non-linear System.

The timings shown in table 5.4 are plotted on the graph shown in figure 5.1. There are 2 separate curves depicted: curve A, shows the results of the configuration used in the Cyclic mode. Curve A is similar in appearence to those curves depicted in chapter 3; however, due to the lack of hardware restart mechanism, a large portion of the timing figures is due to the overhead of software restart. An estimate of this overhead is obtained by subtracting the figure for the asynchronous mode from that for the cyclic mode (both figures for one node), giving a figure of 14.4 sec. Taking this figure away from all three results for the cyclic mode gives the lower row of results in table 5.4. These are plotted as curve B in figure 5.1 which is similar in appearence to the curves depicted in chapter 3 with the communications overheads having a dominant role with the addition of more nodes to the system.

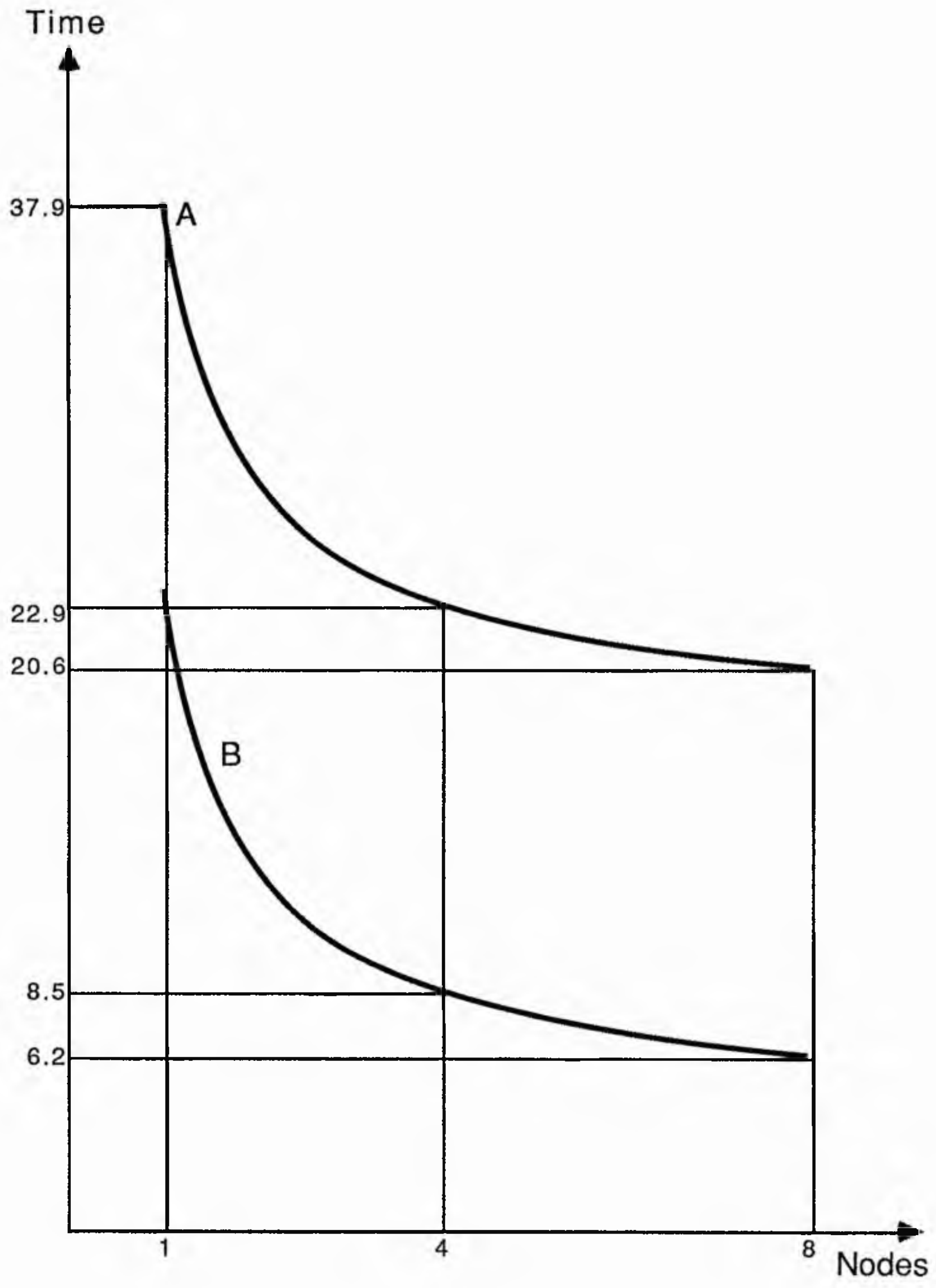


Figure 5.1 Results for the First Order Vector Non-linear System.

5.7 A Performance Model with Hardware Parameters

Experience gained from the extensive use of the system in the last section has provided insight to derive a performance model to show the effect of the three major hardware parameters: global bus cycle time, local bus cycle time and the allocator propogation period.

The total execution time on a multi-node system executing in the cyclic mode is given by ,

$$t_m = N_c\{t_c + t_o + t_r\} \qquad (5.12)$$

where t_m	=	Total Execution Time.
N_c	=	Number of Cycles.
t_c	=	Total Time due to Computations.
t_o	=	Total Time for non-absorbed Communications Overheads.
t_r	=	Time taken to perform a Cyclic Restart.

t_c : Let τ_l be the local bus cycle time. Let η_i be the number of machine cycles performed by node i in executing its workload. Let η_{max} be the number of machine cycles for the largest workload. The computation time t_c is therefore given by,

$$t_c = \eta_{max}\,\tau_l \qquad (5.13)$$

t_o : The communications overheads component of equation 5.12 is the sum of: (a) a waiting period for the allocator to propogate, and (b) the global bus cycle time.

(i) Allocator Propogation Time: The maximum and minimum time that a node has to wait before recieving the allocator are given by equations 5.3 and 5.1 respectively as,

$$\tau'_{w,max} = 7(\tau_s + \tau'_b)$$

$$\tau_{w,min} = 0$$

Hence the average waiting time can be given as:

$$\tau'_{w,av} = 0.5(\tau'_{w,max} + \tau_{w,min}) = 3.5(\tau_s + \tau'_b) \qquad (5.14)$$

To simplify symbols, let τ_a be the average (system and board) allocator time, i.e.

$$\tau_a = 0.5(\tau_s + \tau'_b)$$

which reduces equ. 5.14 to

$$\tau'_{w,av} = 7\, \tau_a$$

(ii) When a node which wishes to use the global bus recieves the allocator and acknowledge signal, it begins the process of broadcasting the data. The time to carry out this function was defined as τ_b earlier, and has been timed to take 2.15uS.

(iii) τ_b is made up of a period proportional to the global bus cycle time, and the allocator propogation time, τ'_b. Let τ_g be the global bus cycle time; and let k be a constant of proportionality. The total time spent by the allocator signal to cross a broadcasting node is therefore

$$\tau_b = \tau'_b + k\,\tau_g \tag{5.15}$$

Therefore with τ'_b = 600nS and τ_g = 1000nS and τ_b = 2150nS, k is calculated to be $2.15 = 0.6 + k$ or $k = 1.55$

(iv) Let τ_o be the time taken to broadcast one byte. This is made up of waiting time for the allocator to arrive, $\tau'_{w,av}$, and broadcast time τ_b , or,

$$\tau_o = \tau'_{w,av} + \tau_b = 3.5\,\tau_s + 3.5\,\tau'_b + \tau'_b + k\,\tau_g \tag{5.16}$$

Taking the average (system and board) allocator time τ_a to replace τ_s and τ'_b gives,

$$\tau_o = 8\,\tau_a + 1.55\,\tau_g \qquad (5.17)$$

(v) Let η_m be the total number of global bus broadcasts that cannot be absorbed with computation. The total overheads time t_o is therefore, $t_o = \eta_m \tau_o$.

t_r : The cyclic restart component of equation 5.12 is measured as a constant equal to 56mS but when implemented in hardware can be reduced to approximately 20nS.

Hence from equation 5.12, the total execution time of a multi-node system can be represented completely by its component parts as:

$$t_m = N_c\{\,\eta_{max}\,\tau_l + \eta_m[\,8\,\tau_a + 1.55\,\tau_g\,] + 20\text{x}10^{-9}\} \qquad (5.18)$$

5.8 An Example Using the New Model

Using equation 5.18 a hypothetical example is calculated and the results are plotted to show the possible execution times which can be achieved as the global bus and the allocator are speeded up, e.g. τ_g = 100nS and τ_a = 60nS.

Due to the ability of associative addressing registers to capture data in parallel, and the provision of 4 input registers in the MUPI/MINNIE system, a conventional global memory system would require 4 times the number of global bus cycles to transfer the same quantity of data. The overheads term needs to be multiplied by a factor of 4 when applying the new model to a conventional system. However, the factor of 4 represents the maximum theoretical advantage, and the more practicle factor of 3 is chosen for the comparison.

The example is assumed to contain a total of 3000 machine cycles which are divided equally between the nodes hence giving $\eta_{max} = 3000N^{-1}$, where N=number of processing nodes in the system. The example is executed over 256 cycles i.e. N_c=256. The number of non-absorbed bus cycles, η_m , is set to be 80% of the total, i.e. 0.8N, where in the simplest case each processor would broadcast one data item and 20% of which are absorbed within the computation phase. For the conventional system the communications overheads are simply 3 times this figure, otherwise an identical bus cycle time and allocator propogation delay are assumed. t_g is the multi processor time using the global memory (conventional) system.

From these considerations the two equations are given as:

MUPI/MINNIE: $$t_m = 256 \{ 3000\, \tau_l\, N^{-1} + 0.8\, [\, 8\, \tau_a + 1.55\, \tau_g\,] N \} \qquad (5.19)$$

CONVENTIONAL: $$t_g = 256 \{ 3000\, \tau_l\, N^{-1} + 2.4\, [\, 8\, \tau_a + 1.55\, \tau_g\,] N \} \qquad (5.20)$$

It is assumed that the inclusion of the cyclic restart overhead is neglible and hence can be disregarded for the comparison.

The two equations 5.19 and 5.20 have been evaluated for a range of values for τ_g, τ_l and τ_a which reflect the timings for the present system and those which can be achieved with todays technology.

Table 5.5 shows the various conditions which have been evaluated for both systems.

τ_g (nS)	τ_l (nS)	τ_a (nS)	GRAPH
1000	1000	600	Fig. 5.2
1000	1000	60	Fig. 5.2
1000	100	600	Fig. 5.3
1000	100	60	Fig. 5.3
100	1000	600	Fig. 5.4
100	1000	60	Fig. 5.4
100	100	600	Fig. 5.5
100	100	60	Fig. 5.5

Table 5.5 Table of conditions which have been evaluated.

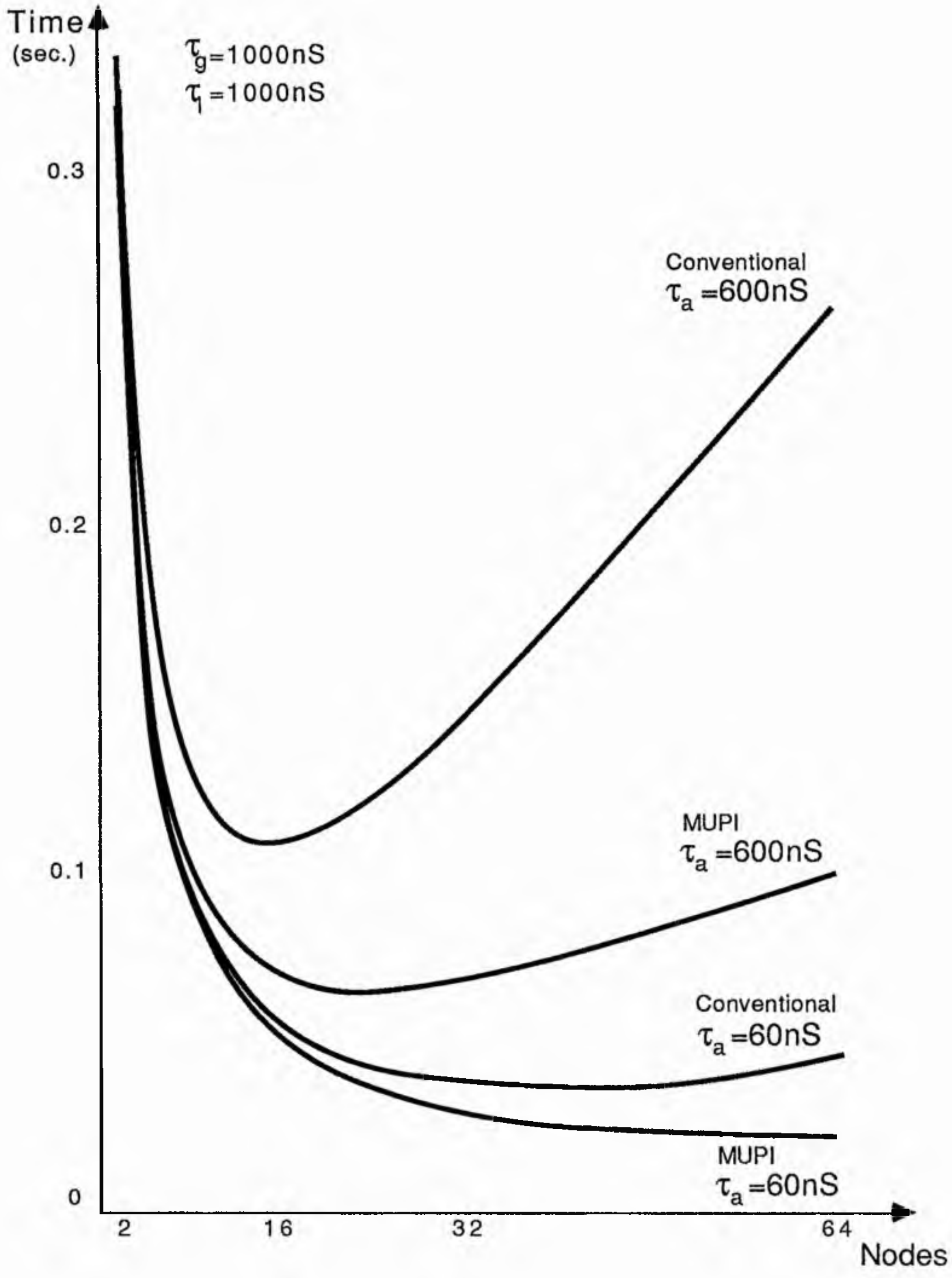


Figure 5.2 Graphs for τ_g =1000nS and τ_l =1000nS.

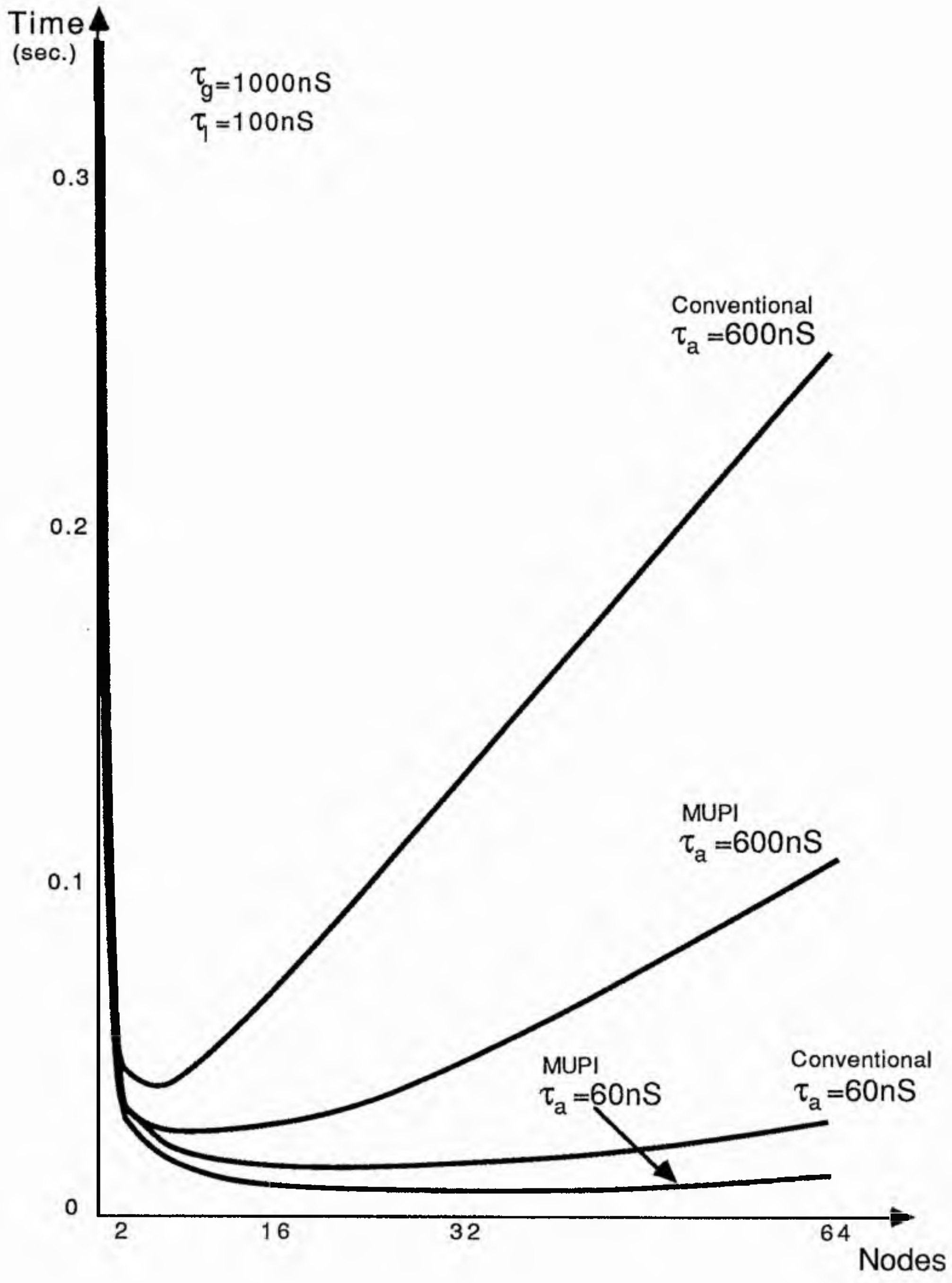


Figure 5.3 Graphs for τ_g =1000nS and τ_l =100nS.

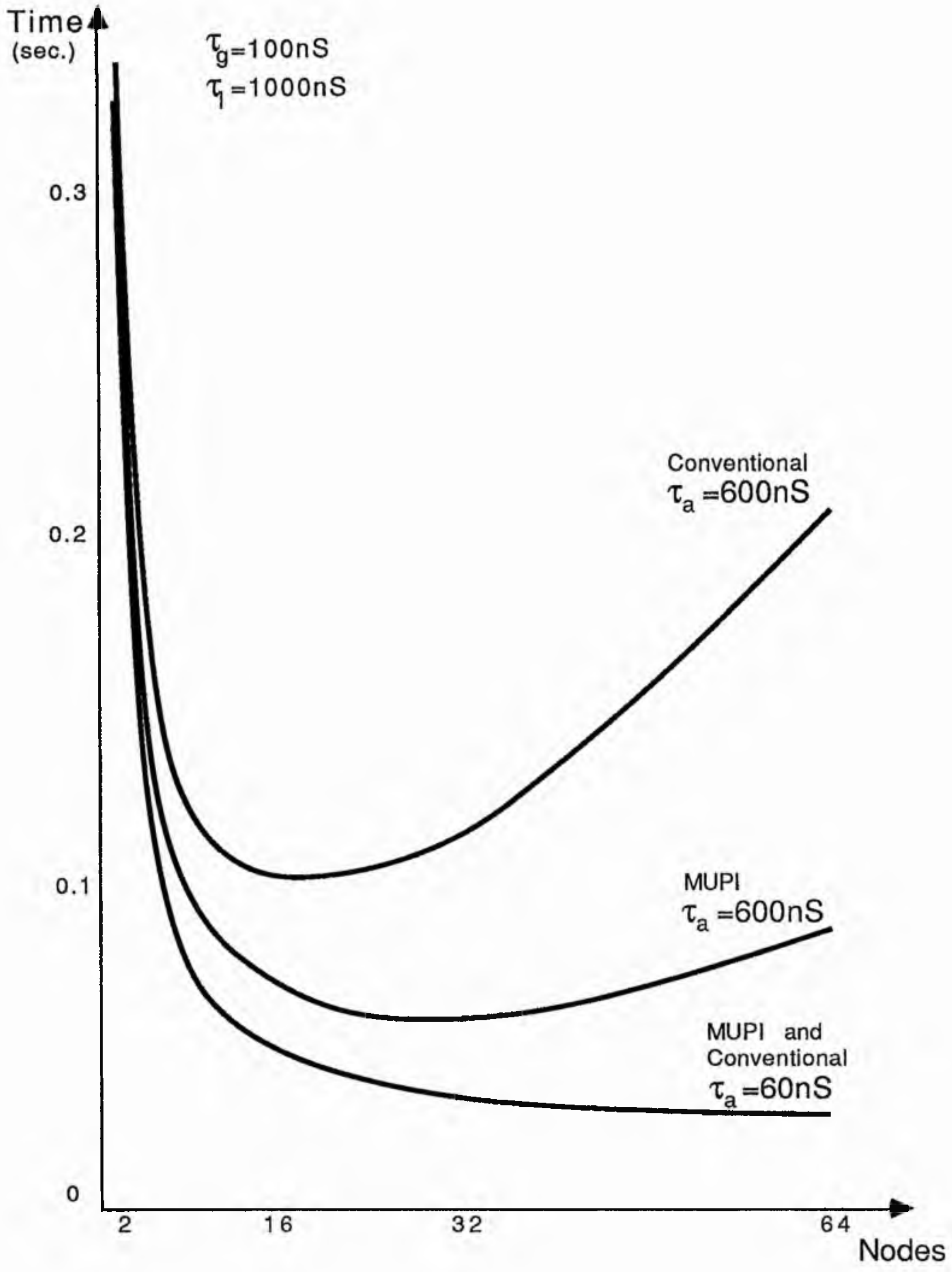


Figure 5.4 Graphs for τ_g =100nS and τ_l =1000nS.

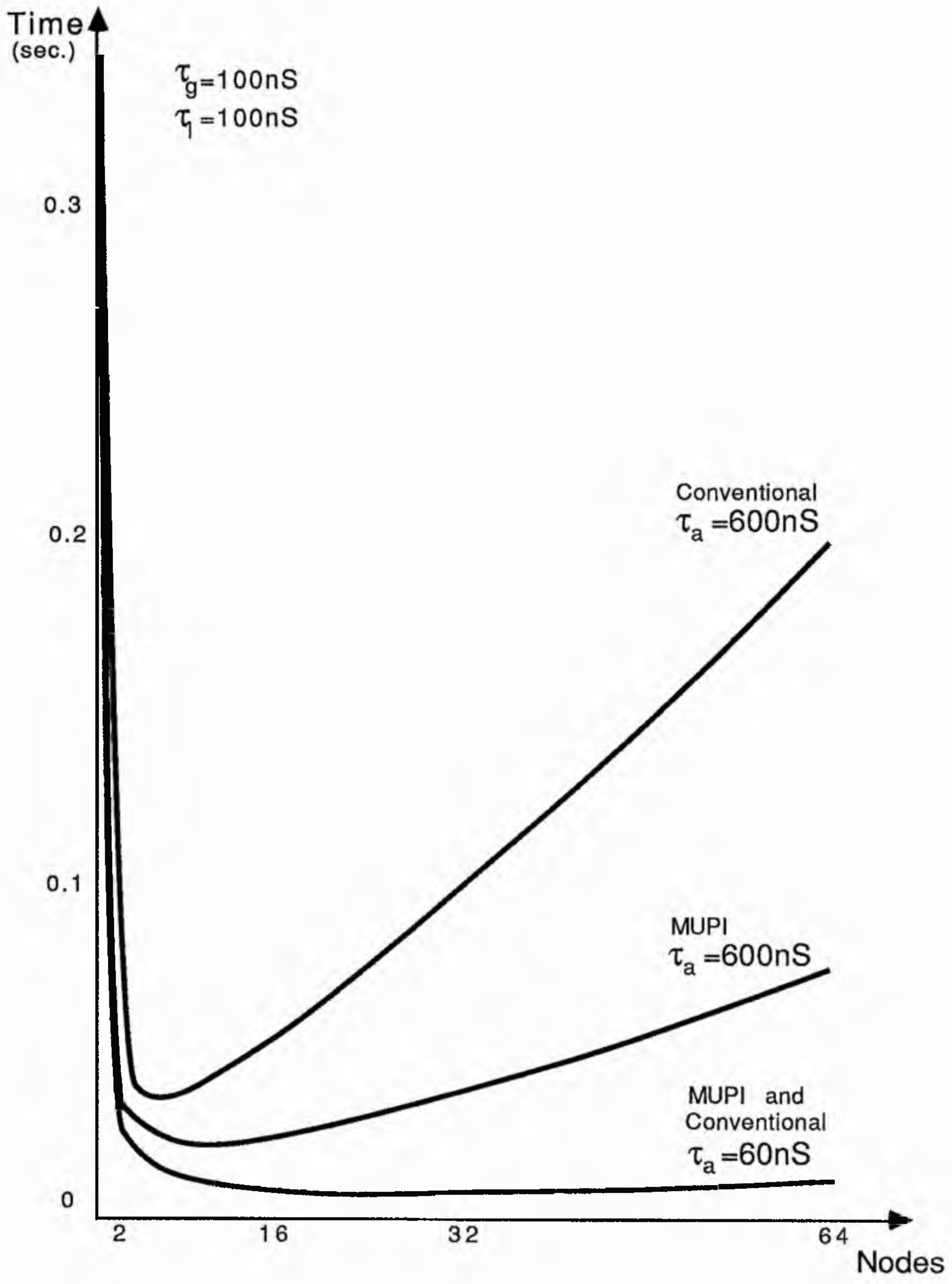


Figure 5.5 Graphs for τ_g =100nS and τ_l =100nS.

From the graphs in figure 5.2 it can be seen that the MUPI/MINNIE system performs better than the conventional system for both situations of τ_a= 600nS. All four graphs show a minimum point for execution time which shows the variation in the number of processors required to achieve this minimum time. The variation in the number of processors required is largely due to the difference in the communications overheads as the computation time is the same for all 4 cases, i.e. the computation time is dependent on τ_l which is equal to 1000nS. When the number of processors in the system are increased the communications overhead increases and the computation time decreases. When the communications overheads is equal to the computation time the minimum execution time is achieved. This can be shown in equation 5.21 which equates the computation time with the communications overheads.

$$AN^{-1} = BN \tag{5.21}$$

where A = 0.768,

and B = 0.0013.

The constants A and B above are for the MUPI/MINNIE system with τ_l =1000nS, τ_g =1000nS and τ_a =600nS. From equation 5.21 the minimum number of processors can be calculated as:

$$N = (A/B)^{1/2} \tag{5.22}$$

For the above conditions this gives the number of processors for the minimum execution time as 24.3. But, as the number of processors in a system cannot be a fraction the number can be given as 24. Equation 5.22 is the same as equation 3.11, the equation for the optimum number of processors which was derived from differentiating the equation for the total execution time.

As τ_a is reduced to 60nS for both systems the communications overheads are reduced and the minimum execution time is reached later, i.e. more processors are required, and is actually smaller than when τ_a =600nS.

The graphs in figures 5.3 to 5.5 show the variations in the systems when both the local and global cycle times are reduced. The τ_g time is a contributing factor in the communications overheads where as τ_l is a contributing factor in the computation time. Figure 5.3 shows the systems with a reduced computation time which makes the systems more sensitive to communications than before, i.e. when τ_l =1000nS. Figures 5.4 and 5.5 both show a near perfect correlation between the MUPI and conventional systems for τ_a =60nS. This correlation is due to the computation time being the dominant factor as the communications overheads are neglible here. Since the difference between the two systems is due to the communications structure this correlation is expected when the communications overheads are very small or the computational part is very large, i.e. a large number of processors is required to achieve the minimum execution time.

It should be pointed out here that the minimum execution time is not always the most desireable factor as the efficiency of the system is very low at this point, i.e. the inclusion of one more processor here does not give a very high reduction in overall execution time. In order to illustrate this fact the conditions for the MUPI/MINNIE system in figure 5.2 with τ_a =600nS are assumed. Another assumption which is to be made is that 2 tasks can be executed on the same system independently of each other, i.e. the communications paths are isolated, this is an enhancment which is explained in chapter 6.

The efficiency E of a multiprocessor system is the ratio of the speed-up factor achieved divided by the number of processors used, i.e.

$$E = \left(\frac{t_1}{t_m}\right) \Big/ N \qquad (5.23)$$

Table 5.6 shows the execution times and efficiencies for 6, 12 and 24 nodes for the assumed conditions above.

Number of Processors	Execution Time	Efficiency
6	0.1359	94%
12	0.0796	80%
24	0.0632	51%

Table 5.6 Various timing results for efficiency comparisons.

The optimum number of processors required for the minimum execution time is given as 24 which gives an execution time of 0.0632 seconds. If the system in question has only 24 processors and it is required to execute the task twice, with different constants say, then the system can be utilised twice with 24 processors. But this would give a total execution time of 0.1264 seconds where as if both tasks where to be executed together using 12 processors each the total execution time is given as 0.0796 seconds i.e. a further reduction of almost 40%.

It can be shown, Al-Dabass [1976b], that the efficiency of a system which is operating at its minimum execution time is given as 50% as this is the point when computations is equal to communications.

From this it can be seen that when deciding the number of processors to use in executing a task a large number of factors have to be considered. When the parallelism of a task is determined automatically by a compiler the computation time and communications overheads can only be approximated. This approximation can be achieved by considering the sizes of the subtasks, i.e. the number of machine cycles, and the total ammount of communications required. When these approximations are made the compiler can allocate the various subsections to various processors within the system. The system may be executing a task at present and so the compiler may only have 5 processors at its disposal when it may require 15. This is where the bottlenecks may occur as the compiler may not know which is the best course of action, i.e. does it allocate the subtasks to only 5 processors or does it wait for the present executing task to finish when it can use 15 processors. It is clear that the area of parallel compilers is a complex and fertile area for research as there are numerous factors to consider.

CHAPTER 6

SUMMARY, CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

CHAPTER 6

SUMMARY, CONCLUSIONS

AND

SUGGESTIONS FOR FUTURE WORK

6.1 Summary

Software programmable total interconnectivity at hardware (bus) speed is the underlying concept explored in this thesis. Associatively addressed input registers at each node are programmed with source addresses of nodes from which data are required. Once programmed, these input (destination) registers will recieve the updated contents of their respective source registers automatically. The update is carried out by a communication infra structure operating continuously and independently from local node programs. It takes place dynamically along a global bus triggered by the arrival of new data in output registers from their local node processors. A two layer mechanism governs the operation of the communication system: a daisy chain allocator in the form of hierarchical rings to detect an update request; and a broadcast system to enable input registers to capture data by multiple nodes simultaneousley.

A fundemental design feature was the integration of all the elements of the communication system on a single silicon device attatched to each node to form an interface unit (MUPI). A preliminary prototype was constructed and tested first using the XILINX programmable device with 4 nodes. This was followed by the development of a semi-custom gate array using the Micro Circuit Engineering (MCE) process for developing such devices for the education sector.

6.2 Conclusions

(i) The associative addressing scheme employed in the interface device has proved to be very effective as it allows a single node to transfer a data item in a single bus cycle to as many nodes as require the data. Clearly the effectiveness of associative addressing increases proportionally with the increasing number of destinations. In the extreme case where a data item is to be sent to all 64 nodes, the speed advantage over a conventional common memory system is a factor of 64.

(i i) A lower bound for the speed advantage is related to the ratio of input to output registers. At the optimum number of processors for a given application the speed advantage is related to the square root of this ratio. For the system implemented in this thesis the ratio is 4 which gives a minimum advantage of 2.

(i i i) The multi-level allocation scheme provided a clear advantage over a single level daisy chain. This advantage becomes more critical as more nodes are added to the system; in the 64 node system the maximum waiting time is reduced from 63 to 14 time units.

(i v) Removing all communications tasks away from processors and concentrating them within an interface unit attached to each node proved to be very successful; processors are now free from delays associated with bus allocation and data transfer leading to increased throughput.

(v) The integration of all communication circuits onto a single device proved to be very successful,- it reduced the component count which considerably simplified the PCB design and subsequent testing and diagnostics.

(vi) From the results presented in chapter 5 it is clear that the speed gain of parallel systems is not only application dependent, but may seem eratic as the number of processors is increased. This is due to the increase in communications which causes the total execution time to increase. Faster communications will normally enable more processors to be used to achieve lower execution time before the optimum is reached.

6.3 Suggestions for Future Work

The system may be further developed as follows:

(i) An automatic hardware restart mechanism is needed to eliminate the delay inherent in software restart.

(ii) The allocation system can be improved to use a binary tree. This would decrease the delays experienced by nodes trying to access the global bus.

(iii) Data bus width can be easily increased to provide higher data throughput.

(iv) Local processors can be upgraded to more powerful 16, 32 or 64 bit devices with on-chip floating point arithmetic units.

(v) The interface circuitry to the master processor may be enhanced to provide extra functions such as real time clock for the restart mechanism.

(vi) Error detection and correction for the global bus is needed. Parity bits may be added to enable the master processor to detect errors which can then be reported to the sending node. Sending nodes may then re-broadcast the data; while recieving nodes would not recognise data with incorrect parity.

(vii) Direct communications with nearest neighbours may be provided.

(viii) Processor and interface unit may be integrated on a single device to provide simpler construction of very large systems.

(ix) Complement the broadcast concept with a read facility, where a node may request to read the output of another without having to be pre-programmed to recieve it.

Appendix 1

MINNIE Control Program.

The program was written on the Master processor using the 6809 programming language PL9, see section A1.4. Various checks are incorporated to prevent faulty operation, e.g. the user will not be able to start the MINNIE system if no node addresses have been set up.

If the MINNIE system is put into the cyclic mode then the number of cycles is prompted for and at the end of each cycle the Master processor automatically restarts the next cycle unless all cycles have been completed.

There is also the facility for the user to start the allocators of the system using a single key command or clearing them, again with the use of a single key.

The menus in the program begin with the main menu which contains all of the basic control commands such as start, reset, cycle, etc. while there are two options which lead to other menus. One of the two options is to set up the node addresses within the system and the other is to provide access to local memories for reading and writing code/data via the keyboard or a file on the Master processor system.

The MINNIE control program that runs on the Master processor is composed of a selection of menus. The menus with a short explanation on their use follow:

A1.1 Main Menu

```
************    MAIN    MENU    ************

              (1) SET UP NODE ADDRESS

              (2) READ/WRITE CODE TO NODE

              (3) SEND ALLOCATOR TO MINNIE

              (4)  RESET

              (5) SLOW START

              (6) FAST START

            * (7) CYCLE 00

              (8) CLEAR ALLOCATOR

              (Q) QUIT


SELECT OPTION BY PRESSING CORRECT KEY
```

This is the main menu of the program which allows the user the ability to access any of the major functions that are neccessary in using the MINNIE system. The menu selections are chosen by single key strokes and two of the options refer to other menus, (i) SET UP NODE ADDRESS, and (ii) READ/WRITE CODE TO NODE.

The only other option within the main menu which may need some explanation is that of CYCLE. When the user wishes to use the cyclic mode this selection would be made. If the user is not already in the cyclic mode, this is known by the * not being on the menu as shown, then a prompt asking for the number of cycles will appear. The user would insert the number of cycles which will now appear alongside the CYCLE menu option. When the system is executing the number of cycles will be seen to be decreasing as each cycle is completed.

A1.2 Node Address Menu

```
************     NA   MENU      ************

          (1) READ ADDRESS          0 0

          (2) WRITE ADDRESS         0 0

          (3) GOTO NEXT NODE

          (4) RETURN TO MAIN MENU


          PRESENT NODE NUMBER       1


     SELECT OPTION BY PRESSING CORRECT KEY
```

Again, with all of the menus the use of the Node Address menu is straight forward. The first thing that should be noted is the PRESENT NODE NUMBER as this tells the user which physical node the user has access to for reading and writing to the node address register within its MUPI. The system always begins with physical node number 1 and increases through the nodes of the system as GOTO NEXT NODE is selected. When a node is chosen it automatically becomes active even if no address has been specified hence the user shall have to be certain how many nodes are required for a particular application.

When the user wishes to read the node address the selection READ ADDRESS is chosen, this will display the address on the menu next to this selection. The writing of a node address is again straight forward, the selection is made which will prompt the user for an address which should be in the range $00 - $3F which facilitates addresses 0 to 63. At the moment there is no function to check the address to see if it has already been chosen for another node. This checking feature is another fool proof procedure which may be implemented in the future.

When all of the neccessary addresses have been set up the user would select the RETURN TO MAIN MENU option which would reset the node address enable signal so as no other nodes are activated.

A1.3 Read/Write Menu

The read/write menu is used to allow the user to read and write sections of code and/or data to any of the active nodes within the MINNIE system. The first selection that the user should make is that of node address. When this selection is made the user is prompted for a node address within the range $00 - $3F. When the user inputs an address this effectivly switches the local memory (4K only) of the node chosen through to the memory map of the Master processor.

```
********* READ/WRITE MENU *********

        (1) NODE ADDRESS          0 0
        (2) FILE NAME             FRED
        (3) READ CODE
        (4) WRITE CODE
        (5) DUMP CODE
        (6) EXAMINE AND CHANGE CODE
```

(7) RETURN TO MAIN MENU

SELECT OPTION BY PRESSING CORRECT KEY

The user nows has several options to choose from. The local memory can be dumped onto the screen with the choice of DUMP CODE which will ask for two addresses which are the start and end addresses for dumping purposes. It must be mentioned here that the mapping of the local memory into the Masters memory map is not a direct relationship. The addresses do not correspond directly, the addresses are offset such that address $D000 in the local memory is actually address $A000 in the Masters map. Hence to dump addresses $D000 - $D0FF of the local memory onto the screen the user should specify addresses $A000 - $A0FF.

Another option the user can choose is that of memory examine and change. This option will again prompt the user for and address from which to start from remembering the same offsets as mentioned for the dump option. The user can then access the local memory byte by byte to examine or change it. A full selection of commands associated with this option will be displayed on the screen.

The two selections which will probably be used more often are the READ CODE and WRITE CODE options. These options will act upon a file which the user specifies with the FILE NAME option. The file name that the user specifies should not contain an extension as the control program assumes an extension of BIN for the file and DAT for the data.

The user wishing to pass a section of program code to a node would specify the node address, the code file name and would then select the WRITE CODE option which will write the BIN and DAT files associated with the selected files name to the selected node. The code is placed at address $D000 of the local memory and the data is placed at address $DF00 of the local memory.

The user wishing to retrieve any data etc. from the node would first specify the node address then would specify the file name. The option READ CODE would be chosen which would result in the local memory section $D000 - $DFFF being placed in the selected file on the Master processor giving it an extension of BIN.

A1.4 Listing of Control Program

```
/* MINNIE CONTROLLER */

CONSTANT     MAP_START      =  $0D,
             MAP_FINISH     =  $FA,
             NOTHING        =  $00,
             ONE            =  $01,
             YSAVE          =  $1000,
             YSAVE_HI       =  $10,
             YSAVE_LO       =  $00,
             NODE_NUMBER    =  $08,
             STACK_INIT     =  $1FFD;

AT  $FF8A  : BYTE  MEMAP;
AT  $E000  : BYTE  RESG, START, GBB, EOUT, NACL, ASTA, RAM;
AT  $E080  : BYTE  STP, NAA, SW, DUMDUM, ST, REN, CY, NCY;
AT  $C840 : BYTE  FCB, ERROR(319);
AT  $CC14 : INTEGER  LINE_POINTER;

ORIGIN = $2000;  STACK = *;

GLOBAL REAL  NOT_MAIN, NODE, MA1, FN, D, SAD, EAD, DM, NOT_RW, C, A, S:
        BYTE R_ADDRESS, W_ADDRESS, CHAR, ERFLAG, KEYCHAR, CN, RW_CHAR,
        SA(30), LO(30), FILE(15), NA_C(2), LO_D(30), S_ADDRESS(5),
        E_ADDRESS(5), DU(20), Q, COUNT, M(10), SADE(5), DUMMY, N_ADDRESS;

BYTE LOAD  "LOAD,A000,";
BYTE LOAD_DATA  "LOAD,AEFD,";
BYTE SAVE  "SAVE,";
BYTE DUMP  "DUMP,";
BYTE MEM_EXA  "MEM,";
BYTE ADDR  ",A000,AFFF";
BYTE EXT  ".DAT";
BYTE COMMA  ",";
```

```
INCLUDE 0.TRUFALSE.DEF;
INCLUDE 0.IOSUBS.LIB;
INCLUDE 0.TERMSUBS.LIB;
INCLUDE 0.HEXIO.LIB;
INCLUDE 0.REALCON.LIB;
INCLUDE 0.STRSUBS.LIB;


/***********************MISC      PROCEDURES**********************/

PROCEDURE SAVE_GLOBAL_POINTER;
        GEN $10, $BF, YSAVE_HI, YSAVE_LO;
ENDPROC;

PROCEDURE RESTORE_GLOBAL_POINTER;
        GEN $10, $BE, YSAVE_HI, YSAVE_LO;
ENDPROC;

PROCEDURE PUTF(REAL N): BYTE BUFFER(20);
        PRINT ASCII(INT(N), .BUFFER);
ENDPROC;

PROCEDURE DO_COMMAND(BYTE .STRING): BYTE COUNT, KHAR, BUFFER(80);
        COUNT = 0;
        REPEAT
            KHAR = STRING(COUNT);
            IF KHAR = 0 THEN KHAR = $0D;
            BUFFER(COUNT) = KHAR;
            COUNT = COUNT + 1;
        UNTIL KHAR = $0D;
        LINE_POINTER = .BUFFER;
        GEN $34, $20;                   /* PUSH Y */
        CALL $CD4B;                     /* FLEX "DOCMD" */
        GEN $35, $20;                   /* PULL Y */
        ERROR = ACCB;
ENDPROC;

PROCEDURE DELAY;
        CURSOR 24,22;
        PRINT "PRESS ANY KEY TO CONTINUE";
        CHAR = GETCHAR_NOECHO;
        CURSOR 00,19;
        ERASE_EOP;
ENDPROC;

PROCEDURE FIRQ;
        RESTORE_GLOBAL_POINTER;
        IF S = 1 THEN
            BEGIN
                IF C = 1 THEN
                        BEGIN
                            CN = CN -1;
                            CURSOR 37,15;
                            PUT_HEX_BYTE(CN);
                            IF CN = 0 THEN S = 0;
                            ELSE
                                 BEGIN
                                      GBB = ONE;
                                      RAM = ONE;
```

```
                                             ST = ONE;
                                             RAM = NOTHING;
                                             GBB = NOTHING;
                                        END;
                                END;
                          ELSE S = 0;
                       END;
                   IF S = 0 THEN
                       BEGIN                    /* DISABLE FIRQ WHEN FIRQ FINISHED */
                          GEN $35, $7E;            /*  PULS  A,B,DP,X,Y,U   */
                          GEN $36, $7E;            /*  PSHU  A,B,DP,X,Y,U   */
                          GEN $35, $02;            /* PULS A    :   CC TO A    */
                          GEN $34, $01;            /* PSHS CC :  NEW CC      */
                          GEN $37, $7E;            /*  PULU  A,B,DP,X,Y,U   */
                          GEN $34, $7E;            /*  PSHS  A,B,DP,X,Y,U   */
                          CURSOR 70, 00;
                          PRINT "DONE";
                          CURSOR 61, 21;
                   END;
             ENDPROC;

             /**************************************************************/



             /*************************NA      PROCEDURES**********************/

             PROCEDURE READ_ADDRESS;
                   CURSOR 22, 20;
                   ERASE_EOP;
                   R_ADDRESS = NAA AND $3F;
             ENDPROC;

             PROCEDURE WRITE_ADDRESS;
                   CURSOR 22, 20;
                   ERASE_EOP;
                   CURSOR 22, 20;
                   PRINT "ENTER ADDRESS : ";
                   RETRY_4:
                       CURSOR 56,20; PRINT "          "; CURSOR 56, 20;
                       CHAR = GET_HEX_BYTE;
                       IF ERFLAG THEN GOTO RETRY_4;
                       IF CHAR AND $C0 THEN GOTO RETRY_4;
                   W_ADDRESS = CHAR;
                   NAA = W_ADDRESS;
                   R_ADDRESS = NAA AND $3F;
                   A = 1;
             ENDPROC;

             PROCEDURE NEXT_NODE;
                   CURSOR 22, 20;
                   ERASE_EOP;
                   IF NODE = NODE_NUMBER THEN
                       BEGIN
                          REN = ONE;
                          EOUT = ONE;
                          STP = ONE;
                          EOUT = NOTHING;
                          NODE = 1;
```

```
            END;
        ELSE
            BEGIN
                STP = ONE;
                NODE = NODE + 1;
            END;
ENDPROC;

PROCEDURE STATUS_UPDATE;
        CURSOR 55, 06;
        PUT_HEX_BYTE(R_ADDRESS);
        CURSOR 55,08;
        PUT_HEX_BYTE(W_ADDRESSS);
        CURSOR 55,15; PRINT "                    " ;
        CURSOR 55, 15: PUTF NODE;
ENDPROC;

PROCEDURE NODE_ADDRESS;
        REN = ONE;
        EOUT = ONE;
        GBB = ONE;
        RAM = ONE;
        STP = ONE;
        EOUT = NOTHING;
        NODE = 1;
        HOME; ERASE_EOP;
        CURSOR 22 ,01;
        PRINT   "*************     NA   MENU     .*************";
        CURSOR 27, 06;
        PRINT "(1) READ ADDRESS";
        CURSOR 27, 08;
        PRINT"(2) WRITE ADDRESS";
        CURSOR 27, 10;
        PRINT "(3) GOTO NEXT NODE";
        CURSOR 27, 12;
        PRINT "(4) RETURN TO MAIN MENU";
        CURSOR 27, 15;
        PRINT "PRESENT NODE NUMBER";
        NOT_MAIN = 1;
        AGAIN_1:      STATUS_UPDATE
                      CURSOR 22, 20;
                      PRINT "SELECT OPTION BY PRESSING CORRECT KEY";
        RETRY_2:      CHAR = GETCHAR_NOECHO
                      IF CHAR
                           CASE '1 THEN READ_ADDRESS;
                           CASE '2 THEN WRITE_ADDRESS;
                           CASE '3 THEN NEXT_NODE;
                           CASE '4 THEN NOT_MAIN = 0;
                      ELSE GOTO RETRY_2;
        IF NOT_MAIN = 1 THEN GOTO AGAIN_1;
        REN = ONE;
        NODE = 0;
        GBB = NOTHING;
        RAM = NOTHING;
ENDPROC;

/*******************************************************************/
```

```
/*************************CODE    R/W    PROCEDURES*******************/

PROCEDURE CODE_NODE_ADDRESS;
        CURSOR 22, 20;
        ERASE_EOP;
        PRINT "ENTER NODE ADDRESS [$00 - $3F] : ";
        RETRY_6:
            CURSOR 56, 20; PRINT "           "; CURSOR 56,20;
            CHAR = GET_HEX_BYTE;
            IF ERFLAG THEN GOTO RETRY_6;
            IF CHAR AND $C0 THEN GOTO RETRY_6;
        N_ADDRESS = CHAR;
        SW = $00;
        D = 0;
        T :
            D = D + 1;
            IF D < 32 THEN GOTO T;
        SW = N_ADDRESS OR $40;
        CURSOR 55, 04;
        PUT_HEX_BYTE(N_ADDRESS);
ENDPROC;

PROCEDURE FILENAME;
        CURSOR 22,  20;
        ERASE_EOP;
        PRINT "ENTER FILE NAME : ";
        INPUT(.FILE,9);
        CURSOR 55, 06;
        PRINT "          " ;
        CURSOR 55, 06;
        PRINT .FILE;
        STRCOPY (.SA, .SAVE);
        STRCOPY (.LO, .LOAD);
        STRCOPY (.LO_D, .LOAD_DATA);
        STRCAT (.SA, .FILE);
        STRCAT (.LO, .FILE);
        STRCAT (.LO_D, .FILE);
        STRCAT(.LO_D, .EXT);
        STRCAT (.SA, .ADDR);
        FN = 1;
ENDPROC;

PROCEDURE READ_CODE;
        CURSOR 00, 19;
        ERASE_EOP;
        IF FN = 1 THEN
            BEGIN
                DO_COMMAND .SA;
                CURSOR 00, 19;
                ERASE_EOP;
            END;
        ELSE
            BEGIN
                CURSOR 24, 20;
                PRINT "FILE NAME NOT SPECIFIED";
                DELAY;
            END;
ENDPROC;

PROCEDURE WRITE_CODE;
```

```
          CURSOR 00, 19;
          ERASE_EOP;
          IF FN = 1 THEN
               BEGIN
                    DO_COMMAND .LO;
                    DO_COMMAND .LO_D;
                    CURSOR 00, 19;
                    ERASE_EOP;
               END;
          ELSE
               BEGIN
                    CURSOR 24, 20;
                    PRINT "FILE NAME NOT SPECIFIED";
                    DELAY;
               END;
ENDPROC;

PROCEDURE START_ADDRESS;
          CURSOR 22,20; ERASE_EOP;
          PRINT "ENTER START ADDRESS : ";
          INPUT (.S_ADDRESS,4);
          CURSOR 55, 04;
          PRINT .S_ADDRESS;
          SAD = 1;
ENDPROC;

PROCEDURE END_ADDRESS;
          CURSOR 22,20; ERASE_EOP;
          PRINT "ENTER END ADDRESS : ";
          INPUT (.E_ADDRESS,4);
          CURSOR 55, 06;
          PRINT .E_ADDRESS;
          EAD = 1;
ENDPROC;

PROCEDURE DUMP_MEMORY;
          IF SAD = 1 .AND EAD = 1 THEN
               BEGIN
                    STRCOPY (.DU, .DUMP);
                    STRCAT (.DU, .S_ADDRESS);
                    STRCAT (.DU, .COMMA);
                    STRCAT (.DU, .E_ADDRESS);
                    HOME; ERASE_EOP;
                    DO_COMMAND .DU; DM = 1;
                    CURSOR 22 ,20; ERASE_EOP;
               END;
          ELSE
               BEGIN
                    CURSOR 22, 20; ERASE_EOP;
                    PRINT "ADDRESSES NOT SPECIFIED COMPLETELY";
               END;
          DELSY;
ENDPROC;

PROCEDURE DUMP_CODE;
D_CODE_AGAIN:
          MA1 = 1; DM = 0; NOT_RW = 1;
          HOME; ERASE_EOP;
          CURSOR 20, 01;
          PRINT  "************    DUMP   ROUTINE    ************";
```

```
        CURSOR 22, 04;
        PRINT "(1) START ADDRESS";
        CURSOR 22, 06;
        PRINT "(2) END ADDRESS";
        CURSOR 22, 08;
        PRINT "(3) DUMP MEMORY";
        CURSOR 22, 10;
        PRINT "(4) RETURN TO READ/WRITE MENU";
        IF SAD = 1 THEN BEGIN CURSOR 55, 04; PRINT .S_ADDRESS; END;
        IF EAD = 1 THEN BEGIN CURSOR 55, 06; PRINT .E_ADDRESS; END;
AGAIN_3:
        CURSOR 22, 20;
        PRINT "SELECT OPTION BY PRESSING CORRECT KEY";
RETRY_7:
        CHAR = GETCHAR_NOECHO;
        IF CHAR
            CASE '1 THEN START_ADDRESS;
            CASE '2 THEN END_ADDRESS;
            CASE '3 THEN DUMP_MEMORY;
            CASE '4 THEN NOT_RW = 0;
        ELSE GOTO RETRY_7;
        IF DM = 1 THEN GOTO D_CODE_AGAIN;
            ELSE IF NOT_RW = 1 THEN GOTO AGAIN_3;
ENDPROC;

PROCEDURE MEM_CODE;
        MA1  =  1;
        HOME; ERASE_EOP;
        CURSOR 02, 02;
        PRINT "ENTER START ADDRESS : ";
        INPUT(.SADE,4);
        STRCOPY(.M, .MEM_EXA);
        STRCAT(.M, .SADE);
        DO_COMMAND .M;
        DELAY;
ENDPROC;

PROCEDURE READ_WRITE_CODE;
        GBB = ONE;
        RAM = ONE;
        N_ADDRESS = $00;
        NOT_MAIN = 1;
MENU_AGAIN_1:
        MA1  =  0;
        HOME; ERASE_EOP;
        CURSOR 22, 01;
        PRINT  "*********    READ/WRITE   MENU     *********";
        CURSOR 27, 04;
        PRINT "(1) NODE ADDRESS";
        CURSOR 27, 06;
        PRINT "(2) FILE NAME";
        CURSOR 27, 08;
        PRINT "(3) READ CODE";
        CURSOR 27, 10;
        PRINT "(4) WRITE CODE";
        CURSOR 27, 12;
        PRINT "(5) DUMP CODE";
        CURSOR 27, 14;
        PRINT "(6) EXAMINE AND CHANGE CODE";
        CURSOR 27, 16;
```

```
          PRINT "(7) RETURN TO MAIN MENU";
          CURSOR 55, 04;
          PUT_HEX_BYTE(N_ADDRESS);
          IF FN = 1 THEN
               BEGIN
                    CURSOR 55, 06;
                    PRINT .FILE;
               END;
AGAIN_2:
          CURSOR 22, 20;
          PRINT "SELECT OPTION BY PRESSING CORRECT KEY";
RETRY_5:
          CHAR = GETCHAR_NOECHO;
          IF CHAR
               CASE '1 THEN CODE_NODE_ADDRESS;
               CASE '2 THEN FILENAME;
               CASE '3 THEN READ_CODE;
               CASE '4 THEN WRITE_CODE;
               CASE '5 THEN DUMP_CODE;
               CASE '6 THEN MEM_CODE;
               CASE '7 THEN NOT_MAIN = 0;
          ELSE GOTO RETRY_5;
          IF MA1 = 1 THEN GOTO MENU_AGAIN_1;
          IF NOT_MAIN = 1 THEN GOTO AGAIN_2;
          SW = $00;
          RAM = NOTHING;
          GBB = NOTHING;
ENDPROC;

/**************************************************************/



PROCEDURE ALLOCATOR: BYTE DUM;
          DUM = 0
          REPEAT
               DUM = DUM + 1;
          UNTIL DUM = 50;
          NACL = NOTHING;
          NACL = ONE;
          ASTA = ONE;
          ASTA = NOTHING;
ENDPROC;



PROCEDURE CLEAR_A;
          NACL = NOTHING;
ENDPROC;



PROCEDURE RE_SET;
          RESG = ONE;
          D = 0;
          T:
               D = D + 1;
               IF D < 10 THEN GOTO T;
          RESG = NOTHING;
```

```
        C = 0;
        S = 0;
ENDPROC;



PROCEDURE S_START;
        GBB = ONE;
        RAM = ONE;
        ST = ONE;
        RAM = NOTHING;
        GBB = NOTHING;
        S = 1;
        CURSOR 70, 00;
        PRINT "" ;
        CURSOR 61, 21;
ENDPROC;

PROCEDURE F_START;
        START = ONE;
        START = NOTHING;
        S = 1;
        CURSOR 70, 00;
        PRINT "" ;
        CURSOR 61, 21;
ENDPROC;



PROCEDURE CYCLE;
        GBB = ONE;
        RAM = ONE;
        IF C = 0 THEN
            BEGIN
                CY = ONE;
                C = 1;
                CURSOR 21, 23;
                PRINT "ENTER NUMBER OF CYCLES (HEX)          ";
                RETRY_3:
                        CURSOR 51, 23; PRINT "          "; CURSOR 51, 23;
                        CHAR = GET_HEX_BYTE;
                        IF ERFLAG THEN GOTO RETRY_3;
                CN = CHAR;
            END;
        ELSE
            BEGIN
                NCY = ONE;
                C = 0;
            END;
        RAM = NOTHING;
        GBB = NOTHING;
ENDPROC;



PROCEDURE QUIT;
        CURSOR 22, 20; ERASE_EOP;
        CURSOR22, 21;
        PRINT "ARE YOU SURE (Y/N)";
RETRY_QUIT:
        CHAR = GETCHAR_NOECHO;
        IF CHAR
            CASE 'Y THEN Q = ONE;
            CASE 'N THEN Q = NOTHING;
        ELSE GOTO RETRY_QUIT;
ENDPROC;




/********************MAIN     MENU     PROCEDURE********************/

PROCEDURE MAIN_MENU;
        R_ADDRESS = NOTHING; W_ADDRESS = NOTHING;
        Q = NOTHING; S = 0;
MAIN_MENU_P:
        HOME; ERASE_EOP;
        CURSOR 22, 01;
        PRINT  "************     MAIN   MENU     ************";
        CURSOR 27, 03;
        PRINT "(1) SET UP NODE ADDRESS";
        CURSOR 27, 05;
        PRINT "(2) READ/WRITE CODE TO NODE";
        CURSOR 27, 07;
        PRINT "(3) SEND ALLOCATOR TO MINNIE";
        CURSOR 27, 09;
        PRINT "(4) RESET";
        CURSOR 27, 11;
        PRINT "(5) SLOW START";
        CURSOR 27, 13;
        PRINT "(6) FAST START";
        CURSOR 27, 15;
        PRINT "(7) CYCLE;
        IF C = 1 THEN
            BEGIN
                CURSOR 25, 15;
                PRINT  "*";
            END;
        CURSOR 27, 17;
        PRINT "(8) CLEAR ALLOCATOR";
        CURSOR 27, 19;
        PRINT "(Q) QUIT";
        CURSOR 37, 15; PUT_HEX_BYTE(CN);
        CURSOR 20, 21;
        PRINT "SELECT OPTION BY PRESSING CORRECT KEY";
        IF A = 1 .AND S = 1 THEN
            BEGIN
                GEN $1C, $BF;
            END;
        RETRY_1:
            IF S = 0 THEN
                BEGIN
                        CURSOR 70, 00;
                        PRINT "DONE";
                        CURSOR 61, 21;
                END;
            CHAR = GETCHAR_NOECHO;
            IF CHAR
```

```
                CASE '1 THEN NODE_ADDRESS;
                CASE '2 THEN READ_WRITE_CODE;
                CASE '3 THEN ALLOCATOR;
                CASE '4 THEN RE_SET;
                CASE '5 THEN S_START;
                CASE '6 THEN F_START;
                CASE '7 THEN CYCLE;
                CASE '8 THEN CLEAR_A;
                CASE 'Q THEN QUIT;
            ELSE GOTO RETRY_1;
        IF Q = NOTHING THEN GOTO MAIN_MENU_P;
ENDPROC;

/*****************************************************************/


/*******************MAIN        PROGRAM*******************************/

PROCEDURE CONTROL;
        SAVE_GLOBAL_POINTER;
        MEMAP = MAP_START;          /* ALTER MEMORY PAGING TO LOOK AT MINNIE */
        RE_SET;                     / * RESET MINNIE */
        MAIN_MENU;
        MEMAP = MAP_FINISH;    /* RETURN MEMORY PAGING TO NORMAL */
        GEN $1A, $40;
ENDPROC;

/*****************************************************************/
```

Appendix 2

Node Monitor Program

The monitor executes in such a way that when the system is switched on if the user wishes to engage the use of the monitor a key is pressed on the keyboard which gives the user complete control of the node. When the user has finished with the use of the monitor the system command is given in order that the monitor program can regain control. The system command is a single key stroke, pressing the letter S.

When the monitor is in system mode it is constantly interrogating its MUPI for a start command. When a start command is recognised the monitor will read the table of input addresses within its local memory and write them to its MUPI, it will then jump into the subtask that the Master processor has created for it. When the subtask is completed the monitor will deactivate the start signal in its MUPI by writing a done command to it. The monitor then returns to the point in the program where it interrogates both the keyboard for user intervention and its MUPI for another start command.

The monitor program also contains an interrupt service routine which the monitor itself is not concerned with. The interrupt routine is such that if an executing subtask were to read one of the input data registers of MUPI with the interrupt enabled then depending on the state of the register an interrupt will be received. The interrupt is such that if there is no valid data within the register i.e.the data flag associated with the register is not set, then on reading the register the local processor will be interrupted. The interrupt routine will alter the program counter so that on returning from the interrupt routine the instruction which triggered the interrupt i.e. the reading of the data register, will be re-executed. The instruction may yet again result in another interrupt which will give rise to it being executed once more. This cycle of events will continue until data has arrived in the data register which sets the data flag and does not result in an interrupt. This interrupt routine is only an option and can be disabled if the local subtask does not wish to be interrupted.

The use of this interrupt routine is very convenient in that the processor does not have to check the validity of previously accessed data.

A2.1 Listing of Node Monitor Program

```
/* START-UP PROGRAM FOR NODE */

CONSTANT    PROM_BASE      =  $F800,
            I_ST           =  $DF00,
            STACK_INIT     =  $CFFD,
            YSAVE          =  $CFFE,
            YSAVE_HI       =  $CF,
            YSAVE_LO       =  $FE,
            IO_BASE        =  $E8,       /* TOP 8-BITS OF I/O ADDRESS AREA */

            ACIA_SET = $10, ACIA_RESET = $03,
            RX_DATA_FULL = $01, PARITY_STRIPPER = $7F,
            TX_DATA_EMPTY = $02,

            CR = $0D, LF = $0A, SP = $20, BEL = $07,

            INT_ON_MASK = $EF,

            TRUE = -1, FALSE = 0, ZERO = $30, ONE = $31;


AT  $8004 : BYTE  ACIA_CTRL(0), ACIA_STAT, ACIA_DATA;
AT  $A00C : BYTE  IAR(4);
AT  $A000 : BYTE ST_DONE;

/ *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * THE MAIN BODY OF THE PROGRAM CODE STARTS HERE *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* /

ORIGIN = PROM_BASE;
```

```
STACK = STACK_INIT;

/ *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * THE FIRST ITEM DECLARED WILL BE AT THE BASE OF *
        * THE STACK i.e. THE STACK WILL GROW DOWN FROM  *
        * THIS POINT.                                  *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* /

GLOBAL     BYTE       JUMP_REGISTERS(0), CONDITION_CR, A_ACCUMULATOR,
                      B_ACCUMULATOR, DIRECT_PAGE:

           INTEGER    X_REGISTER, Y_REGISTER, U_REGISTER, PROGRAM_CTR,
                      RTS_VECTOR:

           BYTE       ERFLAG, KEYCHAR, .B_POINTER, IN, MENU:

           INTEGER    FIRQ_COUNT, IRQ_COUNT;

DPAGE = IO_BASE;      /* SHORTENS CODE WHEN ADDRESSING 'AT' VARIABLES */

/ *
        * * * * * * * * * * * * * *
        * I/O SUBROUTINES *
        * * * * * * * * * * * * * *
* /
PROCEDURE INITIALIZE_CONSOLE_ACIA;
        ACIA_CTRL = ACIA_RESET;
        ACIA_CTRL = ACIA_SET;
ENDPROC;


PROCEDURE PUTCHAR(BYTE OUTCHAR);
        REPEAT UNTIL ACIA_STAT AND TX_DATA_EMPTY;
        ACIA_DATA = OUTCHAR;
ENDPROC;


PROCEDURE GETCHAR:BYTE INCHAR;
        REPEAT UNTIL ACIA_STAT AND RX_DATA_FULL;
        INCHAR = ACIA_DATA AND PARITY_STRIPPER;
        PUTCHAR(INCHAR);
ENDPROC INCHAR;


PROCEDURE MACHINE_PUT;
        PUTCHAR(ACCA);
ENDPROC;


PROCEDURE MACHINE_GET;
        ACCA=GETCHAR;
ENDPROC;


PROCEDURE GET_UPPER_CASE:BYTE INCHAR;
        INCHAR = GETCHAR;
        IF INCHAR >= 'a .AND INCHAR <= 'z
```

```
                THEN INCHAR = INCHAR - $20;
ENDPROC INCHAR;


PROCEDURE CRLF;
        PUTCHAR(CR);
        PUTCHAR(LF);
ENDPROC;


PROCEDURE PRINT(BYTE .STRING);
        WHILE STRING
            BEGIN
                IF STRING = '\ THEN
                        BEGIN
                             .STRING = .STRING + 1;
                             IF STRING
                                  CASE 'N THEN CRLF;
                                  CASE 'B THEN PUTCHAR(BEL);
                        END;
                ELSE PUTCHAR(STRING);
                .STRING = .STRING + 1;
            END;
ENDPROC;


/ *
        * * * * * * * * * * * * * * * *
        * BIT DUMP ROUTINE *
        * * * * * * * * * * * * * * * *
* /
BYTE  MASK $01, $02, $04, $08, $10, $20, $40, $80;


PROCEDURE BITSOUT(BYTE BITCHAR):
        BYTE COUNT;
        COUNT = 8;
        REPEAT
            IF BITCHAR AND MASK(COUNT - 1) = 0
                THEN PUTCHAR(ZERO);
                ELSE PUTCHAR(ONE);
            COUNT = COUNT - 1;
            IF COUNT = 4
                THEN PUTCHAR(SP);
        UNTIL COUNT = 0;
ENDPROC;



/ *
        * * * * * * * * * * * * * * * *
        * HEX I/O ROUTINES *
        * * * * * * * * * * * * * * * *
* /
PROCEDURE GET_HEX_NIBBLE:BYTE INCHAR;
        INCHAR = GET_UPPER_CASE;
        KEYCHAR = INCHAR;
        ERFLAG = TRUE;
        IF INCHAR >= '0 .AND INCHAR <= '9 THEN
            BEGIN
```

```
                    INCHAR = INCHAR - '0;
                    ERFLAG = FALSE;
                END;
            ELSE IF INCHAR >= 'A .AND INCHAR <= 'F THEN
                BEGIN
                    INCHAR = INCHAR - '7;
                    ERFLAG = FALSE;
                    END;
ENDPROC INCHAR;


PROCEDURE GET_HEX_BYTE:BYTE INCHAR;
        INCHAR = SHIFT(GET_HEX_NIBBLE,4);
        IF ERFLAG = TRUE THEN RETURN;
        INCHAR = INCHAR OR GET_HEX_NIBBLE;
ENDPROC INCHAR;


PROCEDURE GET_HEX_ADDRESS:INTEGER INCHAR;
        INCHAR = SWAP(INTEGER(GET_HEX_BYTE));
        IF ERFLAG = TRUE THEN RETURN;
        INCHAR = INCHAR OR INTEGER(GET_HEX_BYTE);
ENDPROC INCHAR;


PROCEDURE PUT_HEX_NIBBLE(BYTE OUTCHAR);
        OUTCHAR = (OUTCHAR AND $0F) + '0;   /* STRIP TOP 4 BITS TO MAKE ASCII */
        IF OUTCHAR > '9
            THEN OUTCHAR = OUTCHAR + '7;     /* A-F OFFSET */
        PUTCHAR(OUTCHAR);
ENDPROC;


PUT_HEX_BYTE(BYTE OUTCHAR);
        PUT_HEX_NIBBLE(SHIFT(OUTCHAR, -4));   /* FIRST DIGIT */
        PUT_HEX_NIBBLE(OUTCHAR);              /* LAST DIGIT */
ENDPROC;


PROCEDURE PUT_HEX_ADDRESS(INTEGER OUTCHAR);
        PUT_HEX_BYTE(SWAP(OUTCHAR));          /* FIRST TWO DIGITS */
        PUT_HEX_BYTE(BYTE(OUTCHAR));          /* LAST TWO DIGITS */
ENDPROC;


PROCEDURE PUT_ASCII_BYTE(BYTE CHAR);
        IF CHAR < $20 .OR > $7D
            THEN PUTCHAR('.);
            ELSE PUTCHAR(CHAR);
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * *
        * GLOBAL POINTER ROUTINES *
        * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE SAVE_GLOBAL_POINTER;
        GEN $10, $8F, YSAVE_HI, YSAVE_LO;  /* STY YSAVE */
ENDPROC;
```

```
PROCEDURE RESTORE_GLOBAL_POINTER;
        GEN $10, $BE, YSAVE_HI, YSAVE_LO;  /* LDY YSAVE */
        ACCA = IO_BASE;                    /* RESTORE THE DIRECT PAGE */
        GEN $IF, $98;                      /* TFR B, DP */
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * * * *
        * DUMP THE STACKED REGISTERS *
        * * * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE ONE_SPACE;
        PUTCHAR(SP);
ENDPROC;


PROCEDURE TWO_SPACES;
        ONE_SPACE;
        ONE_SPACE;
ENDPROC;


PROCEDURE REGISTER_DUMP(INTEGER STACKBASE):BYTE COUNT, CHAR, .B_POINTER;
        PRINT("\N\NAA BB DP XXXX YYYY UUUU PCPC SPSP  CC  EFHINZVC\N");
        .B_POINTER = STACKBASE + 1;        /* POINT AT 'A' */
        REPEAT
            PUT_HEX_BYTE(B_POINTER);
            IF .B_POINTER > STACKBASE + 3 THEN
                BEGIN
                        .B_POINTER = .B_POINTER + 1;
                        PUT_HEX_BYTE(B_POINTER;
                END;
            ONE_SPACE;
            .B_POINTER = .B_POINTER + 1;
        UNTIL .B_POINTER = STACKBASE + 12;
        PUT_HEX_ADDRESS(.B_POINTER);          /* STACK POINTER*/
        TWO_SPACES;
        .B_POINTER = STACKBASE;
        PUT_HEX_BYTE(B_POINTER);              /* CCR IN HEX */
        TWO_SPACES;
        BITSOUT(B_POINTER);                   /* CCR IN BITS */
        CRLF;
        CRLF;
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * SOFTWARE INTERRUPT SERVICE ROUTINES *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE SWI;          /* DUMMY RTI INSTRUCTION ONLY */
ENDPROC;


PROCEDURE SWI2;         /* DUMMY RTI INSTRUCTION ONLY */
ENDPROC;
```

```
PROCEDURE SWI3;         /* DUMMY RTI INSTRUCTION ONLY */
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * HARDWARE INTERRUPT SERVICE ROUTINES *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE NMI;
        RESTORE_GLOBAL_POINTER;
        REGISTER_DUMP(STACK);
ENDPROC;


PROCEDURE FIRQ;         /* DUMMY RTI INSTRUCTION ONLY */
ENDPROC;


PROCEDURE IRQ;               /* ALTERS THE RETURN ADDRESS */
        GEN $32, $6A;        /* LEAS 10,S      * /
        GEN $35, $10;        /* PULS X         * /
        GEN $30, $1C;        /* LEAX -4,X      * /
        GEN $34, $10;        /* LEAS -10,S     * /
ENDPROC;


PROCEDURE RESET;
        JUMP PROM_BASE;
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * HANDLE ARBITRARY PROGRAM COUNTER PULL FROM STACK *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE RETURN_FROM_SUBROUTINE;
        GEN $34 ,$FF;        /* PSHS CC,A,B,DP,X,Y,U,PC */
        GEN $8E, $FF, $FF;   /* LDX £$FFFF */
        GEN $AF, $6A;        /* STX 10,S ...... (PC IS NOW $FFFF) */
        RESTORE_GLOBAL_POINTER;
        REGISTER_DUMP(STACK);
        JUMP PROM_BASE;
ENDPROC;


/ *
        * * * * * * * * * * * * *
        * INITIALISE NODE *
        * * * * * * * * * * * * *
* /
PROCEDURE INIT(INTEGER REG_BASE):BYTE .R_POINTER;
        .R_POINTER = REG_BASE;          /* POINT AT FIRST ADDRESS */
        REPEAT
            IAR(.R_POINTER - REG_BASE) = R_POINTER;
            .R_POINTER = .R_POINTER + 1;
        UNTIL .R_POINTER = REG_BASE + 4;
        IN = TRUE;
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * *
        * RUN NODE PROGRAMME *
        * * * * * * * * * * * * * * * * * * *
* /
PROCEDUR RUN;
        SAVE_GLOBAL_POINTER;
        GEN $34, $7F;               /*  PSHS  S/U,Y,X,DPR,B,A,CCR  */
        CALL  $D000;                /* JUMP TO NODE PROGRAMME */
        GEN $36, $7F;               /*  PULS  S/U,Y,X,DPR,B,A,CCR  */
        RESTORE_GLOBAL_POINTER;
        ST_DONE = FALSE;            /* SEND DONE SIGNAL FOR THIS NODE */
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * * * * *
        * MEMORY EXAMINE AND CHANGE *
        * * * * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE ADDRESS_PROMPT;
        PRINT("\NADDRESS?  ");
ENDPROC GET_HEX_ADDRESS;


PROCEDURE MEMORY_EXAMINE_AND_CHANGE:
        BYTE READ_BYTE, WRITE_BYTE, ADDRESS;
        .ADDRESS = ADDRESS_PROMPT;
        IF ERFLAG = TRUE THE RETURN;
        PRINT("\N\N(+)   next\N(-)   prev\N(/)   again\N");
        TRY_AGAIN:
            CRLF;
        READ_AGAIN:
            PUTCHAR(CR);
            PUT_HEX_ADDRESS(ADDRESS);
            ONE_SPACE;
            PUT_HEX_BYTE(ADDRESS);
            ONE_SPACE;
            WRITE_BYTE = GET_HEX_BYTE;
            IF ERFLAG = TRUE
                THEN IF KEYCHAR = '- .OR KEYCHAR = '+ .OR KEYCHAR = '/
                        THEN BEGIN
                            IF KEYCHAR
                                CASE '- THEN .ADDRESS = .ADDRESS - 1;
                                CASE '+ THEN .ADDRESS = .ADDRESS + 1;
                                CASE '/ THEN GOTO READ_AGAIN;
                            GOTO TRY_AGAIN;
                        END;
            ELSE RETURN;
        ADDRESS = WRITE_BYTE;
        IF WRITE_BYTE <> ADDRESS
            THEN BEGIN
                PRINT("   ?\B");
                GOTO TRY_AGAIN;
            END;
        .ADDRESS = .ADDRESS + 1;
        GOTO TRY_AGAIN;
ENDPROC;


/ *
        * * * * * * * * * * * * * * *
        * HEX/ASCII DUMP *
        * * * * * * * * * * * * * * *
* /
PROCEDURE HEX_DUMP:BYTE .ADDRESS, COUNT, PASSES;
        .ADDRESS = ADDRESS_PROMPT;
        CRLF;
DUMP_AGAIN:
        PASSES = 0;
        REPEAT
            COUNT = 0;
            CRLF;
            PUT_HEX_ADDRESS(.ADDRESS);
            TWO_SPACES;
            REPEAT
                PUT_HEX_BYTE(ADDRESS(COUNT));
                ONE_SPACE;
                COUNT = COUNT + 1;
            UNTIL COUNT = 16;
            TWO_SPACES;
            COUNT = 0;
            REPEAT
                PUT_ASCII_BYTE(ADDRESS(COUNT));
                COUNT = COUNT + 1;
            UNTIL COUNT = 16;
            .ADDRESS = .ADDRESS + 16;
            PASSES = PASSES + 1;
        UNTIL PASSES = 16;
        PRINT(" MORE? ");
        IF GET_UPPER_CASE <> 'N THEN GOTO DUMP_AGAIN;
ENDPROC;


/ *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * PL/9 NODE CONTROL MAIN PROGRAMME *
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* /
PROCEDURE MINI_MONITOR:BYTE COUNT;
        IN = FALSE;

        SAVE_GLOBAL_POINTER;

        INITIALISE_CONSOLE_ACIA;

        RTS_VECTOR = .RETURN_FROM_SUBROUTINE;

        CCR = CCR AND INT_ON_MASK;      /* ENABLE IRQ */

        CRLF;

        GEN $34, $FF;
        REGISTER_DUMP(STACK);
        MENU + TRUE;
```

```
REPEAT

    REPEAT
        IF ST_DONE = TRUE THE
                BEGIN
                     IF IN = TRUE THEN RUN;
                     ELSE
                          BEGIN
                                INIT(I_ST);
                                RUN;
                          END;
    UNTIL ACIA_STAT AND TX_DATA_EMPTY;  /* KEY PRESSED */


TRY_AGAIN:
    PRINT("\N>  ");
    IF GET_UPPER_CASE
        CASE 'D THEN HEX_DUMP;
        CASE 'M THEN MEMORY_EXAMINE_AND_CHANGE;
        CASE 'R THEN
                BEGIN
                     INIT(I_ST);
                     RUN;
                END;
        CASE 'S THEN MENU = FALSE;

    IF MENU = TRUE THEN MENU_AGAIN;

FOREVER;
```

Appendix 3

Circuit Diagrams for

One Node, Board Buffers, and

Board/System Allocator System.

The following circuit diagrams are of the circuits on a single 8-node cicrcuit board. The one node circuit is repeated 8 times with the addition of the board buffers and the board allocator circuitry and circuitry to facilitate the system allocator.

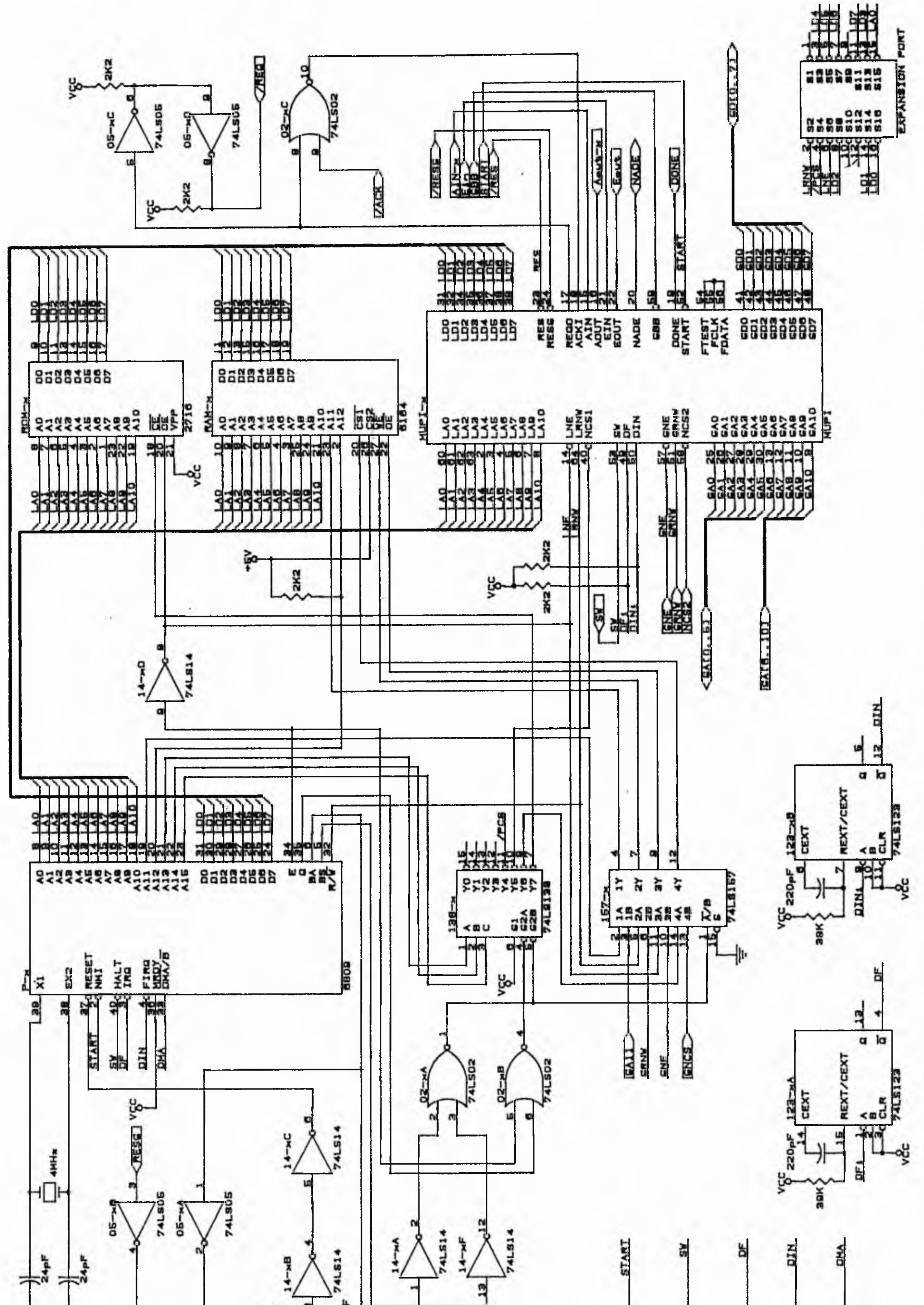
The order of the circuit diagrams is:

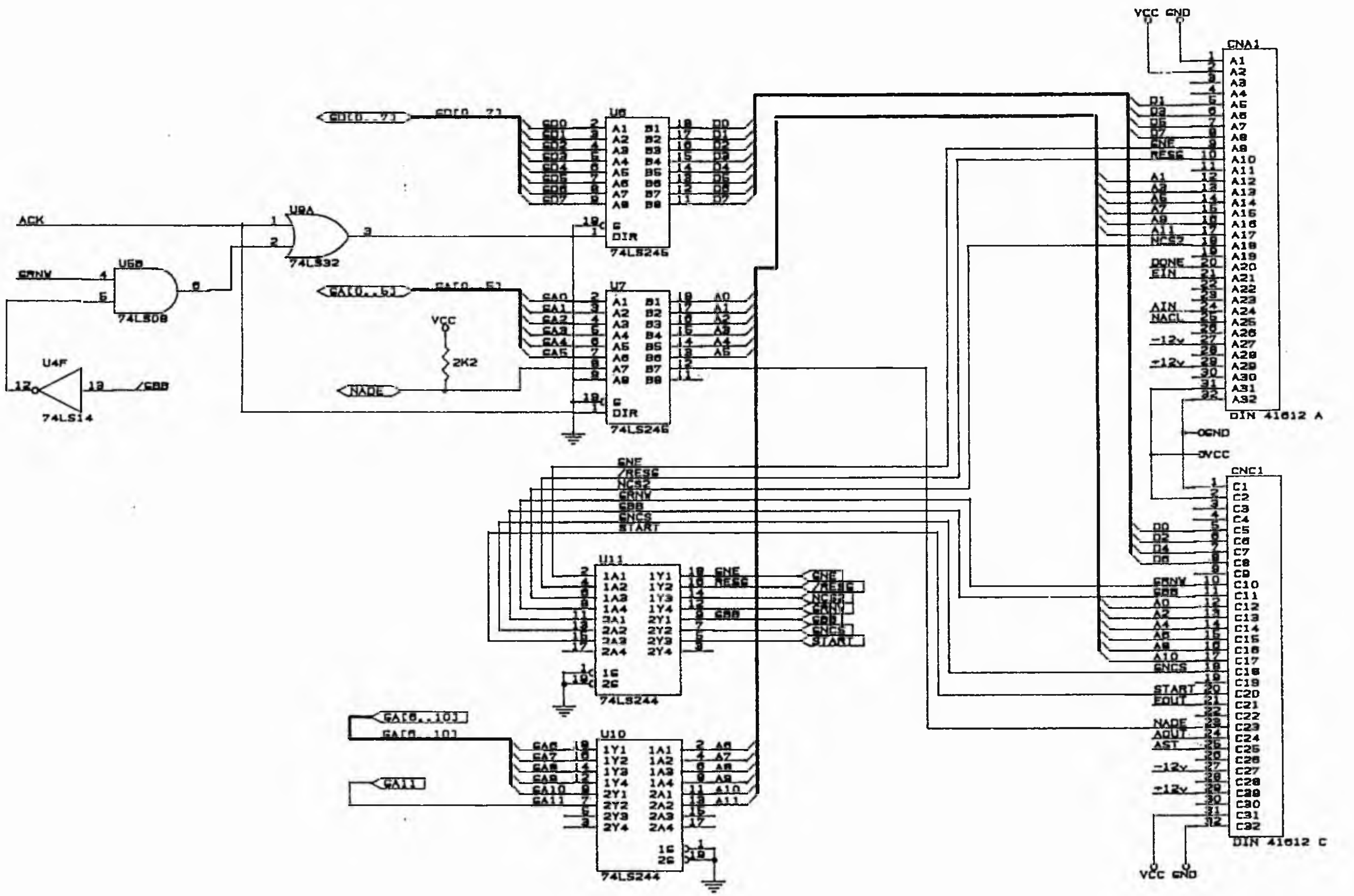
(a) Single Processing Node.

(b) Board Buffers.

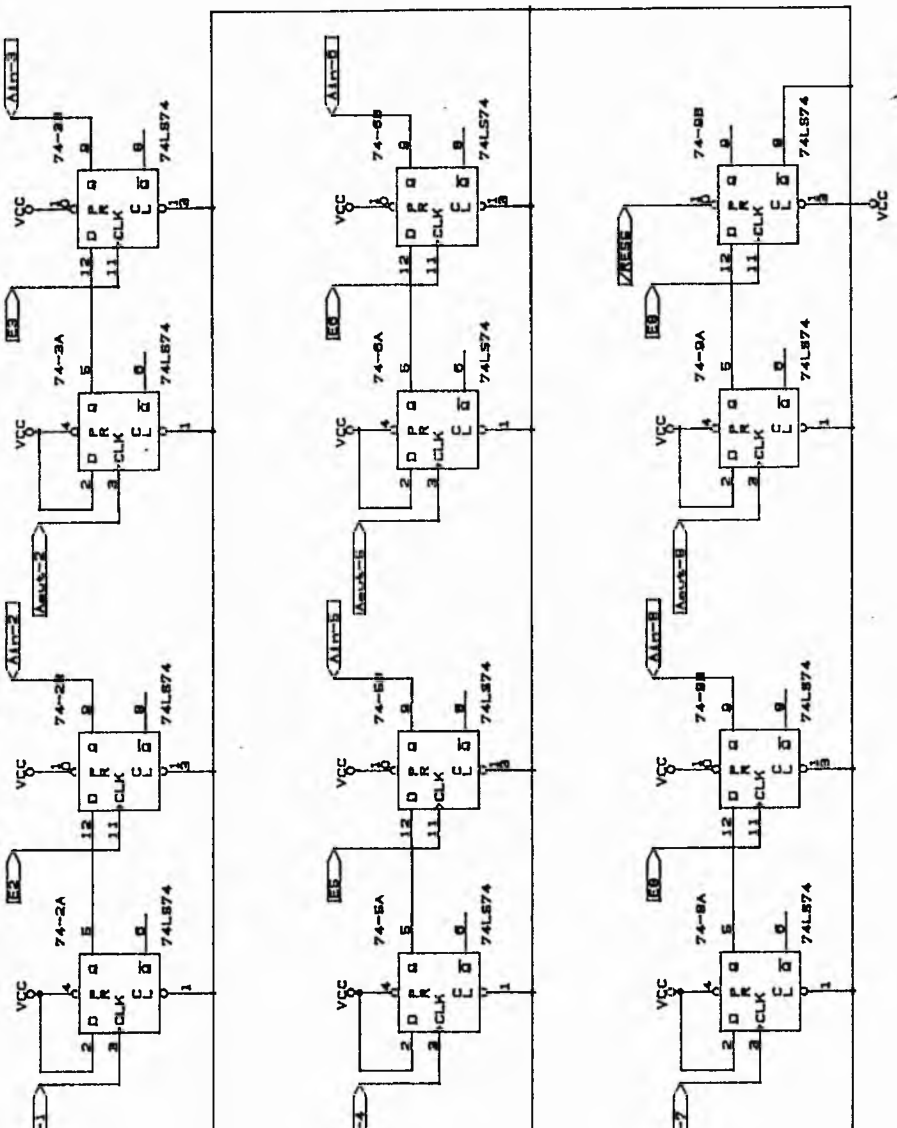
(c) Board Allocator System.

(d) System Allocator Circuitry on 8-node board.

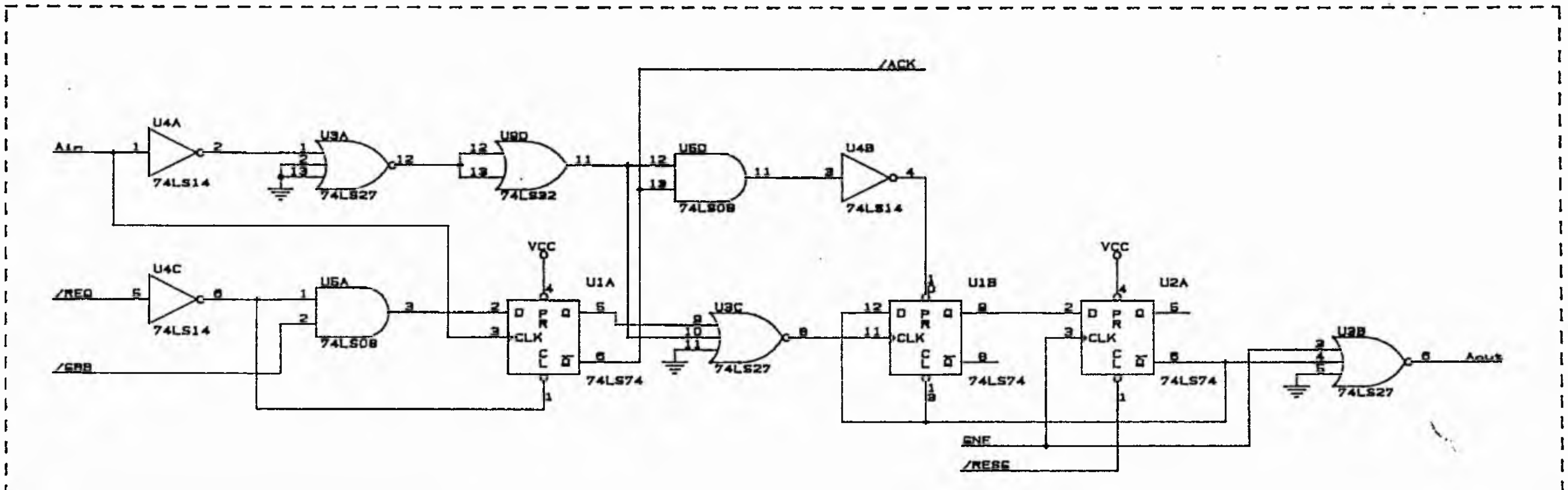

6809
2716
6164
MUPI
74LS138
74LS157
74LS123
74LS02
74LS05
74LS14
EXPANSION PORT
4MHz
VCC
2K2



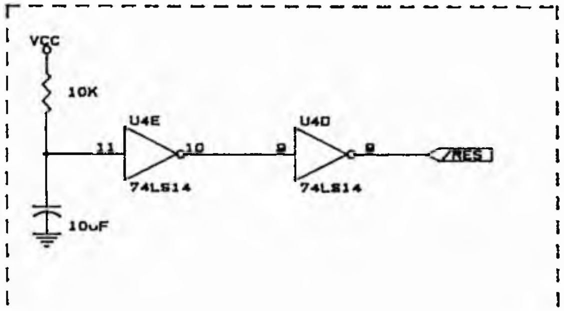
ACK
GRNW
U9A
74LS32
U6B
74LS08
U4F
74LS14
/GBB
GD[0..7]
GA[0..5]
NADE
VCC
2K2
U6
74LS245
U7
74LS245
U11
74LS244
U10
74LS244
GA[6..10]
GA11
GNE
/RESG
NCS2
GRNW
GBB
GNCS
START
CNA1
DIN 41612 A
CNC1
DIN 41612 C
VCC GND
0GND
0VCC
DONE
EIN
AIN
NACL
-12v
+12v
EOUT
AOUT
AST



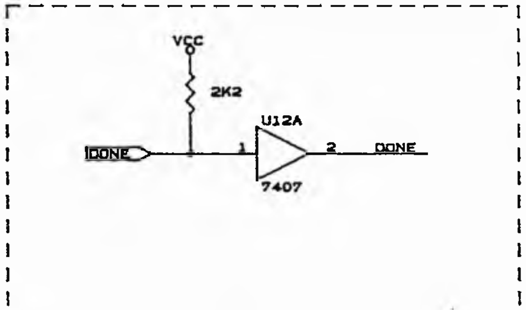
74LS74
74-1A
74-1B
74-2A
74-2B
74-3A
74-3B
74-4A
74-4B
74-5A
74-5B
74-6A
74-6B
74-7A
74-7B
74-8A
74-8B
74-9A
74-9B
VCC
CLK
Ain-1
Ain-2
Ain-3
Ain-4
Ain-5
Ain-6
Ain-7
Ain-8
Aout-1
Aout-2
Aout-3
Aout-4
Aout-5
Aout-6
Aout-7
Aout-8
E1
E2
E3
E4
E5
E6
E7
E8
E9
/RESE
/ABTA
NAGL



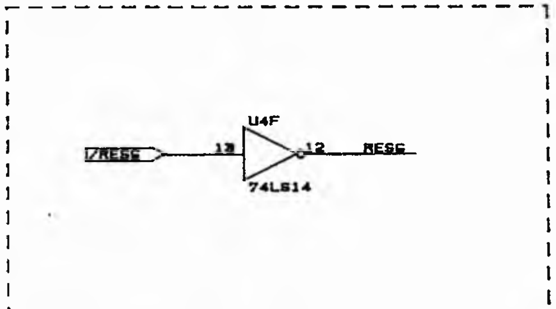
SYSTEM ALLOCATOR CIRCUIT WITH BUS ACKNOWLEDGE



RESET FOR MUPI= ON BOARD



BUFFER FOR DONE SIGNAL

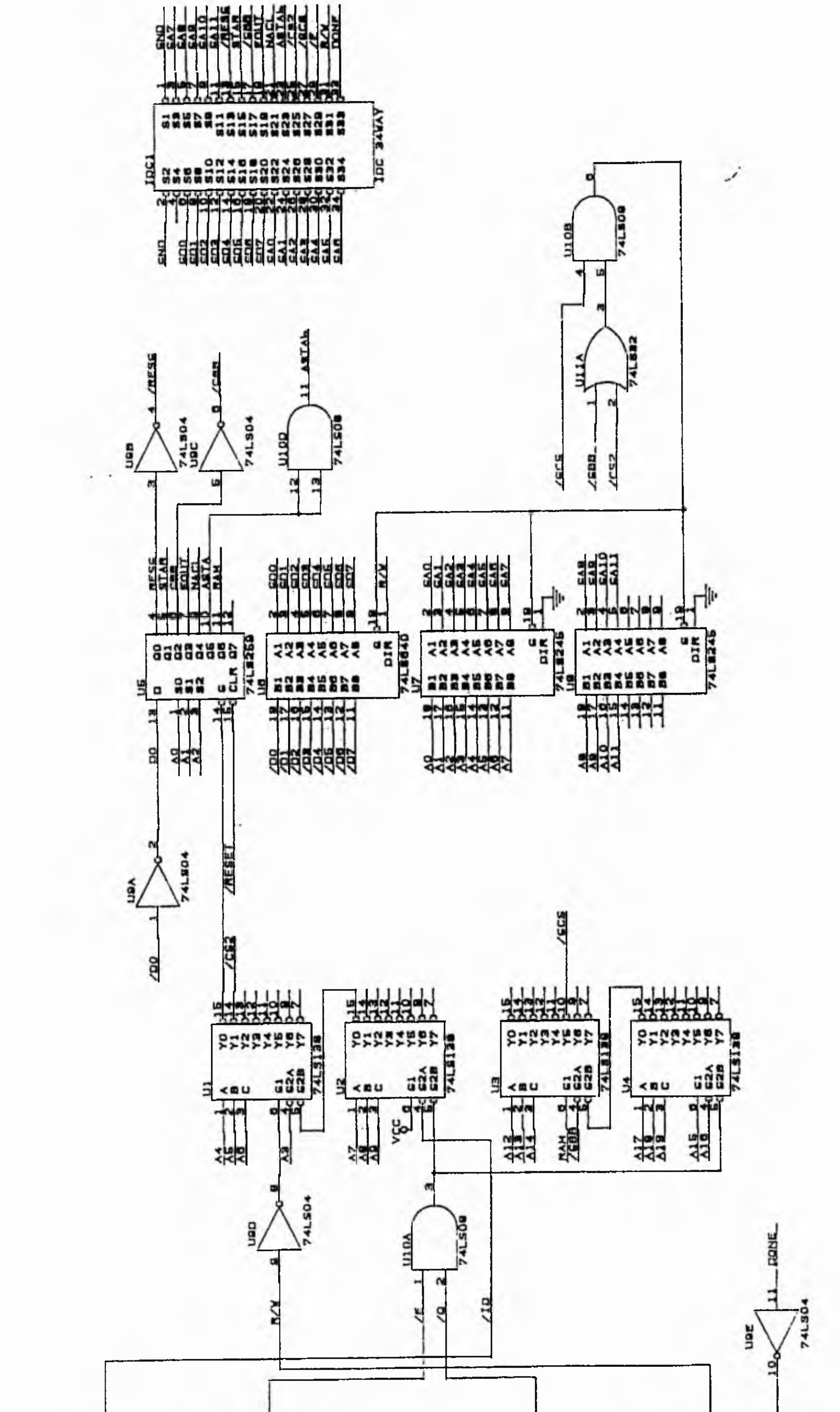


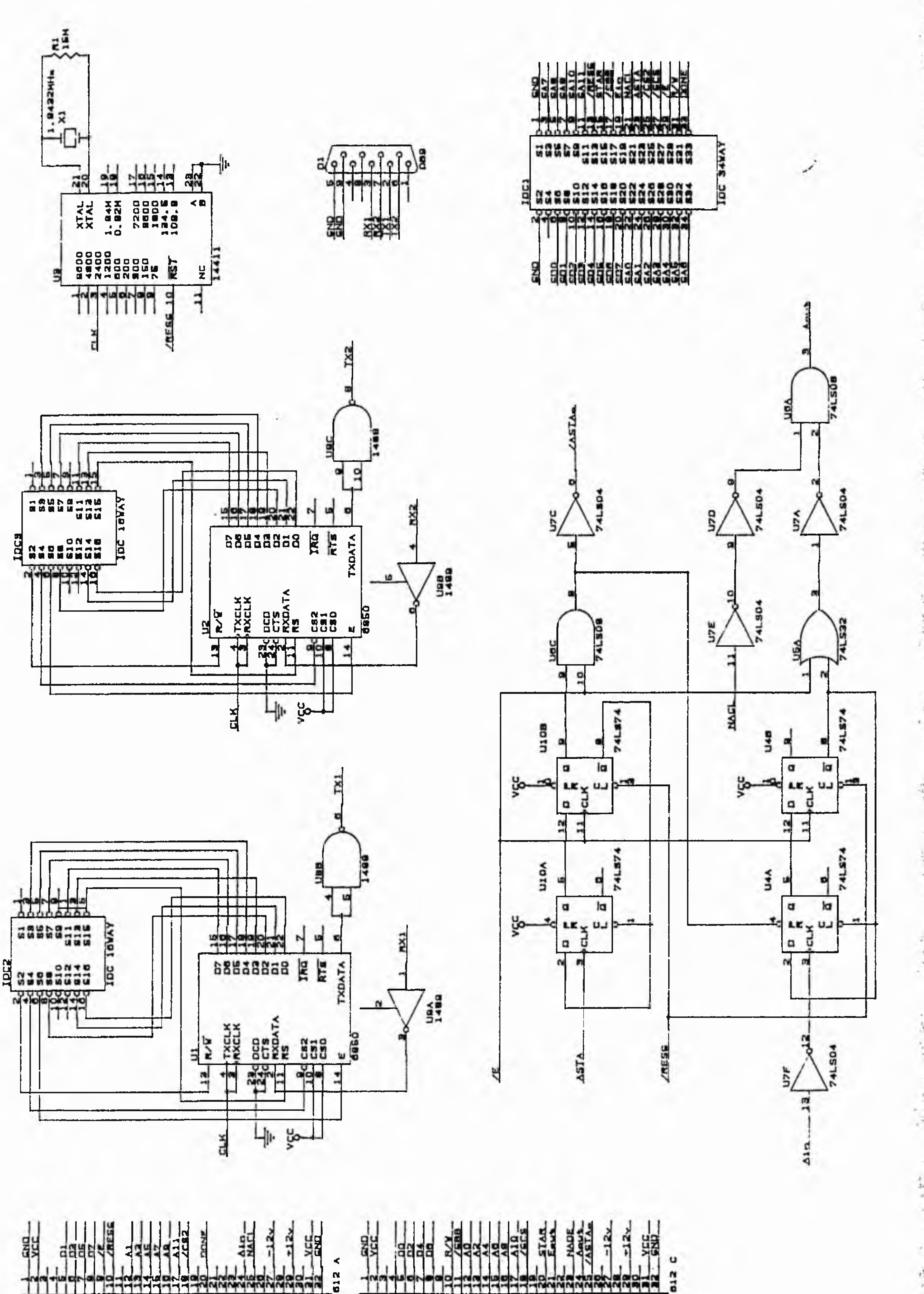
INVERTED GLOBAL RESET SIGNAL

Appendix 4

Circuit Diagrams for Windrush and MUPI Interface Boards.

The following circuit diagrams are of the two interface boards which support the MINNIE system. The first board is the interface board which sits in the Windrush rack giving the Master processor control over the MINNIE system. The second board is the MINNIE interface board which sits in the MINNIE rack and supports the system allocator for the MINNIE system as well as providing two terminal interface adaptors for the system which any node in the system can be linked to.


IDC1
IDC 34WAY
U9B
74LS04
U9C
74LS04
U10D
74LS08
U10B
74LS08
U11A
74LS32
U9A
74LS04
U6
74LS259
U8
74LS640
U7
74LS245
U9
74LS245
U1
74LS138
U2
74LS138
U3
74LS138
U4
74LS138
U9D
74LS04
U10A
74LS08
U9E
74LS04
VCC
/RESET
/CS2
/SCS
DONE
R/W
/IO
/E
/Q
DIN 41612 A
DIN 41612 C



U9
14411
X1
1.8432MHz
R1
15M
D1
D89
IDC1
IDC 34WAY
IDC2
IDC 16WAY
IDC3
IDC 16WAY
U1
U2
6850
U8A
U8B
U8C
1489
1488
U7A
U7C
U7D
U7E
U7F
74LS04
U6A
U6C
74LS08
U5A
74LS32
U4A
U4B
U10A
U10B
74LS74
TX1
TX2
RX1
RX2
CLK
VCC
/RESG
/E
ASTA
/ASTAe
NACL
Ain
Aout
DIN 41612 A
DIN 41612 C

Appendix 5

Memory Maps for MUPI,

Single Node

and Master Processor.

A5.1 MUPI Memory Map

The memory map for both the local and the global sides of the MUPI device is as follows:

Local Bus Side of MUPI:

LOCATION	READ ONLY	WRITE ONLY
0 0	START	DONE
0 4	DATA FLAG 0	CLEAR ALL DATA FLAGS
0 5	DATA FLAG 1	- - - - - - - -
0 6	DATA FLAG 2	- - - - - - - -
0 7	DATA FLAG 3	- - - - - - - -
0 8	DATA IN 0	DATA OUT 0
0 9	DATA IN 1	- - - - - - - -
0A	DATA IN 2	- - - - - - - -
0B	DATA IN 3	- - - - - - - -
0C	IVA 0	IVA 0
0D	IVA 1	IVA 1
0E	IVA 2	IVA 2
0F	IVA 3	IVA 3

Global Bus Side of MUPI:

LOCATION	USE	MEANING
00	STP,	step node address enable on.
01	NAA,	node address (read/write).
02	SW,	local memory switch.
03	DUMDUM,	not used.
04	ST,	slow start location.
05	REN	reset enable signal.
06	CY,	set cyclic mode.
07	NCY,	set non-cyclic mode.

A5.2 Single Node Memory Map

The memory map of a single node is as follows:

LOCATION	USE
\$8000-\$9FFF	Asynchronous Communications Adaptor (Mirrored).
\$A000-\$BFFF	MUPI device (Mirrored).
\$C000-\$DFFF	RAM (8K).
\$E000-\$FFFF	EPROM (2K Mirrored).

It can be seen from the memory map that some of the devices are mirrored, this is because the decoding has not been broken down more than necessary, hence the MUPI device for instance can be accessed at locations \$A000-\$A00F, \$A010-\$A01F and so on.

A5.3 Master Processor Memory Map

The memory map of the master processor control part of the Windrush design system is as follows:

LOCATION (Page F)	USE	
$A000-$AFFF	$D000-$DFFF of Page 0 mapped here so as to access locations within MINNIE such as local memories.	
$E000-$E006	1 bit control registers on master interface board.	
$E000	RESG,	global reset.
$E001	START,	fast start.
$E002	GBB,	global bus busy.
$E003	EOUT,	enable out of master.
$E004	NACL,	clear all allocators.
$E005	ASTA,	start all allocators.
$E006	RAM,	control for MINNIE access.
$E080-$E087	Specific locations within the MUPI devices.	
$E080	STP,	step node address enable on.
$E081	NAA,	node address (read/write).
$E082	SW,	local memory switch.
$E083	DUMDUM,	not used.
$E084	ST,	slow start location.
$E085	REN	reset enable signal.
$E086	CY,	set cyclic mode.
$E087	NCY,	set non-cyclic mode.

For further information concerning the memory mapping of the Windrush System the reader should refer to the Hardware manual of the system [WIND39].

Appendix 6

Pinout Specification

# of the MUPI Device.

A6.1 Pin Description

A basic description of the pins of the MUPI device is covered as follows:

Symbol	Pin No.	Name	I/O
Vcc	33	Power supply	
Vss	1	Ground	
RES	23	Reset	I
RESG	24	Global Reset	I
LA0-LA3	60-63	Local Address Bus	I/O
LA4-LA10	2-8		
LD0-LD1	31-32	Local Data Bus	
I/O			
LD2-LD7	34-39		
LNE	14	Local Processor Clock	I
LRNW	64	Local Read/Write Signal	I
NCS1	40	Low enable local select	I
GA0-GA5	25-30	Global Address Bus (i)	I/O
GA6-GA10	13-9	Global Address Bus (ii) only part of the GA Bus is used to output addresses on the global bus for the use of other nodes.	I

GD0-GD7	41-48	Global Data Bus	I/O
GNE	57	Master Processor Clock	I
GRNW	51	Master Processor R/W	I
NCS2	58	Master low enable select	I
AIN	15	Allocator In	I
AOUT	16	Allocator Out	
O			
REQO	17	Request Out	O
ACKI	18	Acknowledge In	I
EIN	21	Enable In (for Node Address)	I
EOUT	22	Enable Out	O
SW	53	Memory Switch Signal	O
DF	49	Data Flag Interrupt (no data)	O
DIN	50	Data In Interrupt	O
NADE	20	Node Address/Data Enable	I/O
GBB	59	Global Bus Busy	I
DONE	19	Processor Done	O
START	52	Fast Node Start	I
FTEST	54	Various Test Pins for	
FCLK	55	Silicon test when	
FDATA	56	wafers are made.	

Appendix 7

Xilinx

Logic Cell Array.

Design and implementation techniques for semi-conductor electronic systems have evolved from discrete, to standard-logic, to custom and semi-custom devices.

Standard-logic devices are widely available from multiple sourcers, enjoy widespread familiarity among designers, and can be easily and quickly used to implement designs. However, they suffer from low levels of function integration, poor circuit-board utilization and high power- consumption.

Custom and semi-custom devices, on the other hand, offer the highest degree of function integration, low power consumption, excellent circuit-board utilization and are supported by semi-automated design tools. But, they suffer from long design and implementation leadtimes, high engineering costs and long production periods. In addition, issues of multiple sourcing and minimum order quatities prevent many potential users from qualifying with vendors of these devices.

The Xilinx family of Logic Cell Array (LCA) devices solves this dilemma. The LCA offers not only the ease of implementation of standard logic devices, but also the design flexibility, low power-consumption, and high function-density of custom and semi-custom devices. Moreover, LCA-based products can be designed, implemented and integrated into production systems very quickly, at a remarkably low cost.

A Logic Cell Array device contains three basic building blocks:

(1) Configurable Logic Blocks (CLBs).

(2) Input/Output Blocks (IOBs).

(3) Interconnect.

Figure A7.1 shows the organisation of an LCA device. This LCA version contains 64 CLBs (which is the version to be used in the implementation of the design), arranged in an 8-by-8 matrix, with each logic block identified by its row and column letters. Interconnects occupy the space between the rows and columns of the CLBs and between the CLBs and the surrounding IOBs. IOBs are numbered to match the number of the package pin to which they are connected.

Configuration is the process through which the functions to be performed by the CLBs, IOBs, and interconnect of the LCA are defined. Configuration is specified with the Xilinx XACT Design System, which produces configuration data. The configuration data are loaded into an LCA, enabling it to perform the functions. Without the configuration data, the LCA remains unconfigured. When all power is removed from the LCA, the LCA returns to the unconfigured state.

Configuration data can be passed to an LCA device from the following sources:

(a) External Memory - At any time using a parallel 8-bit stream.

(b) External Processor Or Another LCA - At any time using a serial bit- stream.

The method that will be adopted is that of the External Memory where an eprom containing the configuration data will sit next to the LCA on the circuit-board.

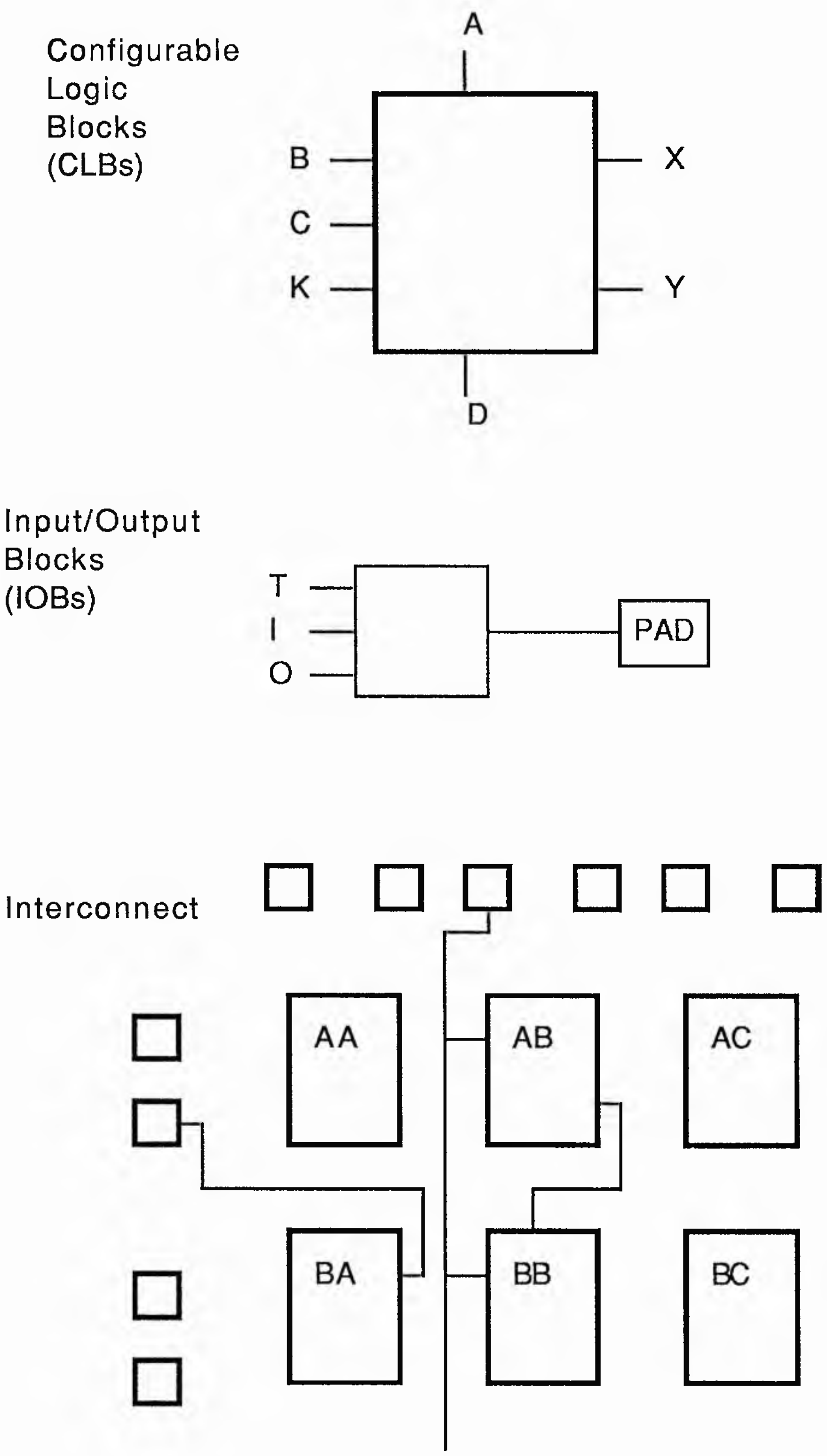


Figure A7.1 LCA Device organisation (XC-2064).

The Xilinx XACT Design System is an integrated package of design tools for developing configuration data for the LCAs. All aspects of configuration are specified through interactive graphics software.

The Xilinx software that is currently being used does not contain a logic simulator to simulate the logic configuration. But, the software does contain a Delay command which calculates delays within the logic including rise, fall, and set-up times of flip-flops, latches, and combinational logic.

For a more complete discription of the Xilinx system and the possible configurations of the CLBs, iOBs, and Interconnect the reader should refer to the Xilinx Manuals, Xilinx [1986a, b].

Appendix 8

Program Listings for

Evaluation of a

First Order Vector Non-linear System

The listings of the example programs for the evaluation of a first order vector non-linear system have been devided into four sections; (i) the common procedures for all nodes e.g. procedures for SIN and COS, (ii) the main procedure for the single node evaluation, (iii) the various main procedures for the 4 node evaluation, (iv) the various main procedures for the 8 node evaluation.

A8.1 Listing of Support Procedures

```
CONSTANT      PROM_BASE     =  $D000,
              BEL           =  $07,
              ESC           =  $1B,
              RDR_FULL      =  $01,
              TDR_EMPT      =  $02;

AT  $8004 : BYTE  ACIA_CONTROL(0), ACIA_STATUS, ACIA_DATA;
AT  $A000 : BYTE  ST_DONE;
AT  $A004 : BYTE  DF(4);
AT  $A008 : BYTE  DATA(4);
AT $DF04 : INTEGER  Q;
AT  $C000 : BYTE  A(256);
AT  $C100 : BYTE  B(256);
AT  $C200 : BYTE  C(256);
AT  $C300 : BYTE  D(256);
AT  $DA00 : BYTE  E(256);
AT  $DB00 : BYTE  F(256);
AT  $DC00 : BYTE  G(256);
AT  $DD00 : BYTE  H(256);
```

```
ORIGIN = PROM_BASE;

real _pio2 1.5707963;

procedure _poly(real op, .table: byte count): real temp;
        temp = table(count);
        repeat
            count = count - 1;
            temp = temp * op + table(count);
        until count = 0;
endproc real temp;

real sin_coeff
        1.0,
        -0.1666666,
        8.333332E-3,
        -1.9852E-4,
        -2.8255E-6,
        -3.70E-8;

procedure sin(real op): byte negative, quadrant;
        if op = 0
            then return real 0;
        quadrant = fix(int(op / _pio2));
        op = op - quadrant * _pio2;
        negative = quadrant and 2;
        if quadrant and 1
            then op = _pio - op;
        op = op * _poly(op * op, .sin_coeff, 5);
        if negative
            then op = -op;
endproc real op;


real cos_coeff
        1.0,
        -0.5,
        0.041666642,
        -1.3888397E-3,
        2.47609E-5,
        -2.605E-7;

procedure cos(real op): byte negative, quadrant;
        if op = _pio2
            then return real 0;
        quadrant = fix(int(op / _pio2));
        op = op - quadrant * _pio2;
        negative = 0;
        if quadrant = 1 .or quadrant = 2
            then negative = 1;
        if quadrant and 1
            then op = _pio2 - op;
        op = _poly(op * op, .cos_coeff, 5);
        if negative
            then op = -op;
endproc real op;
```

A8.2 Listing of Main Procedure for Single Node Evaluation

```
PROCEDURE MAIN;
        A(0) = 2;          /* INITIAL VALUES FOR ALL EQUATIONS */
        B(0) = 3;
        C(0) = 7;
        D(0) = 9;
        E(0) = 13;
        F(0) = 1;
        G(0) = 6;
        H(0) = 4;

        Q = 0;

        REPEAT
            A(Q+1) =A(Q) + 10 - C(Q) * COS(0.07*B(Q)) + D(Q);
            B(Q+1) = B(Q) + 22 - C(Q) * D(Q) * SIN(0.03*A(Q)) + A(Q);
            C(Q+1) = C(Q) + A(Q) + SIN(0.07*B(Q));
            D(Q+1) = D(Q) + 17 - SIN(0.07*A(Q));
            E(Q+1) = E(Q) + SIN(C(Q)) + COS(D(Q));
            F(Q+1) = F(Q) +B(Q) + SIN(0.05*B(Q));
            G(Q+1) = G(Q) + 0.02 * COS(C(Q));
            H(Q+1) = H(Q) + A(Q) * SIN(COS(0.09*A(Q)));

            Q = Q + 1;
        UNTIL Q = $00;
ENDPROC;
```

A8.3 Listing of Main Procedures for Four Node Evaluation

A8.3.1 Node 00 Listing for 4 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0) = B,
                                                DATA(1) = C,
                                                DATA(2) = D,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            BEGIN
                A(0) = 2;              /* INITIAL VALUE FOR A STORED IN MEMORY */
                F(0) = 1;              /* INITIAL VALUE FOR F STORED IN MEMORY */
            END;

        ELSE
            BEGIN
                Q = Q - 1;
                A(Q+1) =A(Q) + 10 - DATA(1) * COS(0.07*DATA(0)) + DATA(2);
                F(Q+1) = F(Q) +DATA(0) + SIN(0.05*DATA(0));
            Q = Q + 1;
        END;

    DATA(0)  =  A(Q);               /* OUPUT VALUE OF A TO OTHER NODES */
                                    /* THROUGH THE OUTPUT REGISTER      */

    Q = Q + 1;
ENDPROC;
```

A8.3.2 Node 01 Listing for 4 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0)  =  A,
                                                DATA(1)  =  C,
                                                DATA(2)  =  D,
* /


PROCEDURE MAIN;
    IF Q = 0 THEN
        BEGIN
            B(0)  =  3;             /* INITIAL VALUE FOR B STORED IN MEMORY */
            G(0)  =  6;             /* INITIAL VALUE FOR G STORED IN MEMORY */
        END;

    ELSE
        BEGIN
            Q = Q - 1;
            B(Q+1)  =  B(Q)  +  22  -  DATA(1)  *  DATA(2)  *  SIN(0.03*DATA(0))  +  DATA(0);
            G(Q+1)  =  G(Q)  +  0.02  *  COS(DATA(1));
            Q = Q + 1;
        END;

    DATA(0)  =  B(Q);               /* OUPUT VALUE OF B TO OTHER NODES */
                                    /* THROUGH THE OUTPUT REGISTER      */

    Q = Q + 1;
ENDPROC;
```

A8.3.3 Node 02 Listing for 4 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0)  =  A,
                                                DATA(1)  =  B,
                                                DATA(2)  =  D,
* /


PROCEDURE MAIN;
    IF Q = 0 THEN
        BEGIN
            C(0)  =  7;             /* INITIAL VALUE FOR C STORED IN MEMORY */
            E(0)  =  13;            /* INITIAL VALUE FOR E STORED IN MEMORY */
            END;

        ELSE
            BEGIN
                Q = Q - 1;
                C(Q+1) = C(Q) + DATA(0) + SIN(0.07*DATA(1));
                E(Q+1) = E(Q) + SIN(C(Q)) + COS(DATA(2));
                Q = Q + 1;
            END;

        DATA(0) = C(Q);                     /* OUPUT VALUE OF C TO OTHER NODES */
                                            /* THROUGH THE OUTPUT REGISTER      */
        Q = Q + 1;
ENDPROC;
```

A8.3.4 Node 03 Listing for 4 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0) = A,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            BEGIN
                D(0) = 9;                   /* INITIAL VALUE FOR D STORED IN MEMORY */
                H(0) = 4;                   /* INITIAL VALUE FOR H STORED IN MEMORY */
            END;

        ELSE
            BEGIN
                Q = Q - 1;
                D(Q+1) = D(Q) + 17 - SIN(0.07*DATA(0));
                H(Q+1) = H(Q) + DATA(0) * SIN(COS(0.09*DATA(0)));
                Q = Q + 1;
            END;

        DATA(0) = D(Q);                     /* OUPUT VALUE OF D TO OTHER NODES */
                                            /* THROUGH THE OUTPUT REGISTER      */
        Q = Q + 1;
ENDPROC;
```

A8.4 Listing of Main Procedures for Eight Node Evaluation

A8.4.1 Node 00 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0) = B,
                                                DATA(1) = C,
                                                DATA(2) = D,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            A(0) = 2;                   /* INITIAL VALUE FOR A STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                A(Q+1) =A(Q) + 10 - DATA(1) * COS(0.07*DATA(0)) + DATA(2);
                Q = Q + 1;
            END;

        DATA(0) = A(Q);                 /* OUPUT VALUE OF A TO OTHER NODES */
                                        /* THROUGH THE OUTPUT REGISTER     */
        Q = Q + 1;
ENDPROC;
```

A8.4.2 Node 01 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0) = A,
                                                DATA(1) = C,
                                                DATA(2) = D,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            B(0) = 3;                   /* INITIAL VALUE FOR B STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                B(Q+1) = B(Q) + 22 - DATA(1) * DATA(2) * SIN(0.03*DATA(0)) + DATA(0);
                Q = Q + 1;
            END;

        DATA(0) = B(Q);                 /* OUPUT VALUE OF B TO OTHER NODES */
                                        /* THROUGH THE OUTPUT REGISTER     */
        Q = Q + 1;
ENDPROC;
```

A8.4.3 Node 02 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0) = A,
                                                DATA(1) = B,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            C(0) = 7;                   /* INITIAL VALUE FOR C STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                C(Q+1) = C(Q) + DATA(0) + SIN(0.07*DATA(1));
                Q = Q + 1;
            END;

        DATA(0) = C(Q);                 /* OUPUT VALUE OF C TO OTHER NODES */
                                        /* THROUGH THE OUTPUT REGISTER      */
        Q = Q + 1;
ENDPROC;
```

A8.4.4 Node 03 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS      DATA(0) = A,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            D(0) = 9;                   /* INITIAL VALUE FOR D STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                D(Q+1) = D(Q) + 17 - SIN(0.07*DATA(0));
                Q = Q + 1;
            END;

        DATA(0) = D(Q);                 /* OUPUT VALUE OF D TO OTHER NODES */
                                        /* THROUGH THE OUTPUT REGISTER      */
        Q = Q + 1;
ENDPROC;
```

A8.4.5 Node 04 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS        DATA(0) = C,
                                                  DATA(1) = D,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            E(0) = 13;                  /* INITIAL VALUE FOR E STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                E(Q+1) = E(Q) + SIN(DATA(0)) + COS(DATA(1));
                Q = Q + 1;
            END;

        Q = Q + 1;
ENDPROC;
```

A8.4.6 Node 05 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS        DATA(0) = B,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            F(0) = 1;                   /* INITIAL VALUE FOR F STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                F(Q+1) = F(Q) +DATA(0) + SIN(0.05*DATA(0));
                Q = Q + 1;
            END;

        Q = Q + 1;
ENDPROC;
```

A8.4.7 Node 06 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS        DATA(0) = C,
* /
```

```
PROCEDURE MAIN;
        IF Q = 0 THEN
            G(0) = 6;                  /* INITIAL VALUE FOR G STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                G(Q+1) = G(Q) + 0.02 * COS(DATA(0));
                Q = Q + 1;
            END;

        Q = Q + 1;
ENDPROC;
```

A8.4.8 Node 07 Listing for 8 nodes

```
/ *     THE ALLOCATION OF INPUT AND OUTPUT REGISTER ADDRESSES IS SUCH THAT FOR
        INPUTS THROUGH THE INPUT REGISTERS     DATA(0) = A,
* /


PROCEDURE MAIN;
        IF Q = 0 THEN
            H(0) = 4;                  /* INITIAL VALUE FOR H STORED IN MEMORY */

        ELSE
            BEGIN
                Q = Q - 1;
                H(Q+1) = H(Q) + DATA(0) * SIN(COS(0.09*DATA(0)));
                Q = Q + 1;
            END;

        Q = Q + 1;
ENDPROC;
```

REFERENCES

Al-Dabass [1976a]: D.Al-Dabass , "An Evaluation of the Effectiveness of Multiprocessor Clusters in Real-Time Applications," IFAC/IFIP International Workshop on Real-Time Programming, Paris, 1976.

Al-Dabass [1976b]: D.Al-Dabass, "Parallel Processors in the Design and Simulation of Dynamical Systems," PhD Thesis, Dept of Electrical Engineering and Electronics, North Staffordshire Polytechnic, Staford, 1976.

Al-Dabass [1977]: D.Al-Dabass, "Microprocessor based parallel computers and their application to the solution of control algorithms," Proc. Inter. Comput. Symp., 1977, pp. 261-270.

Al-Dabass [1980]: D.Al-Dabass, "Common Memory Systems: Two Detailed Models," Control Systems Centre, UMIST, July 1980.

Allen and Cocke [1976]: F.E.Allen and J.Cocke, "A Program Data Flow Analysis Procedure," Communications of the ACM, vol. 19, no. 3, March 1976.

Artym and Mason [1988]: R.Artym and J.S.Mason, "XPXM/C: a taxonomy of processor coupling techniques," IEE Proc., vol. 135, pt. E, no. 3, May 1988.

Barnes et al.[1968]: G.H.Barnes, R.M.Brown, M.Kato, D.J.Kuck, D.L.Slotnick, and R.A.Stokes, "The Illiac IV computer," IEEE Trans. Comput., vol. C-17, pp.746-757, Aug. 1968.

Batcher [1980]: K.E.Batcher, "Architecture of a massively parallel processor," Proc. 7th Symp. Comput. Arch. SIGARCH8, May 1980, pp.168-173.

Bhuyan [1987]: L.N.Bhuyan, "Interconnection Networks for Parallel and Distributed Processing," IEEE Computer, pp. 9-12, June 1987.

Browne [1984]: J.C.Browne, "TRAC: an environment for parallel computing," COMPCON, Spring '84, pp. 294-298.

CPS [1988]: "Customer Procurement Specification," C.P.S. no. D/76057/D, MCE part no. MT76057, Trent Polytechnic, 1988.

Dennis [1980]: J.B.Dennis, "Data Flow Supercomputers,"IEEE Computer, pp.48-56, Nov. 1980.

Dimopoulos [1985]: N.J.Dimopoulos, "On the Structure of the Homogeneous Multiprocessor," IEEE Trans. on Comput., vol. c-34, no. 2, pp. 141-150, 1985.

Fairbairn [1982]: D.G.Fairbairn, "VLSI: A New Frontier for Systems Designers," IEEE Computer, pp.87-96, Jan. 1982.

Flynn [1966]: M.J.Flynn, "Very High-Speed Computing Systems," Proc. of the IEEE, vol. 54, no. 12, Dec. 1966.

Fung and Torng [1979]: F.Fung and H.Torng, "On the Analysis of Memory Conflicts and Bus Contentions in a Multiple Microprocessor System," IEEE Trans. on Comput., Jan. 1979.

Hellerman [1967]: H.Hellerman, "Digital Computer System Principles," pp.228-229, McGraw-Hill, New York, 1967.

Hockney and Jessope [1981]: R.W.Hockney and C.R.Jesshope, "Parallel Computers," Published by Adam Hilger Ltd, pp. 158-178, 1981.

Hoener and Roeder [1977]: S.Hoener and W.Roeder, "Efficiency of a Multiprocessor System with Time-Shared Busses," EUROMICRO 77, Sept. 1977.

Hwang and Briggs [1985]: K.Hwang and F.A.Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill Series in Computer Orginisation and Architecture, 1985.

Jenson [1978]: C.Jenson, "Taking another approach to supercomputing," Datamation, vol. 24, pp. 159-172, Feb. 1978.

Kaiser [1980]: D.Kaiser, "iAPX 432 Object Prime Preliminary Draft," Intel Corporation, Aug. 1980.

Krajewski [1985]: R.Krajewski, "Multiprocessing: an overview," Byte, May 1985, pp. 171-181.

Kruskal and Snir [1982]: C.P.Kruskal and M.Snir, "Some Results on Multistage Interconnection Networks for Multiprocessors," New York University, Comput. Sci. Dept., Tech. Rep. 51, 1982.

Kuck et ai. [1981]: D.J.Kuck et al., "Dependence Graphs and Compiler Optimisations," Proceedings 8th Symposium on Principles of Programming Languages, Jan. 1981.

Levy [1978]: J.V.Levy, "Buses, The Skeleton of Computer Structures," Computer Engineering: a DEC View of Hardware System Design by C.G.Bell, J.C.Mudge and J.E.McNamara, 1978.

Markenscoff [1985]: P.Markenscoff, "Markov models for a multiple processor system with a shared bus," IEE Proc., vol. 132, pt. E, no.6, pp. 316-322, 1985.

Marsan and Gregoretti [1981]: M.A.Marsan and F.Gregoretti, "Memory Interface Models for a Multiprocessor System, with a Shared Bus and a Single External Common Memory," EUROMICRO Journal, Feb. 1981.

Marsan and Gerla[1982]: M.A.Marsan and M.Gerla, "Markov Models for Multiple Bus Multiprocessor Systems," IEEE Trans. on Comput., vol. c-31, no. 3, pp. 239-248.

MCE [1985a]: Micro Circuit Engineering Ltd., "BX Simulator System User Guide," 1985.

MCE [1985b]: Micro Circuit Engineering Ltd., "Array Design Manual," 1985.

MCE [1985c]: Micro Circuit Engineering Ltd., "BX Simulator System Technical Reference Manual," 1985.

Reddaway [1973]: S.F.Reddaway, "DAP - A distributed array processor," Proc. 1st Annu. Symp. Comput. Arch., Florida, Dec. 1973, pp.61-65.

Slotnick et al. [1962]: D.L.Slotnick, W.C.Borck, and R.C.McReynolds, "The SOLOMON computer," Proc. AFIPS-FJCC, vol. 22, 1962, pp. 97-107.

Thurber et al. [1972]: K.J.Thurber, E.D.Jenson, L.A.Jack, L.L.Kinney, P.C.Patton and L.C.Anderson, "A Systematic Approach to the Design of Digital Bussing Structures," Proc. AFIPS Fall Joint Computer Conference 41, 1972.

Unger [1958]: S.H.Unger, "A computer oriented towards spatial problems," Proc. IRE, vol. 46, pp. 1744-1750, Oct. 1958.

Whiting et al. [1975]: R.H.Whiting, C.B.Chang and M.Athans, "On the State and Parameter Estimation for Manoeuvering Re-entry Vehicles," Proc. of the 6th Symposium on Non-linear Estimation, San Diego, 1975.

Willis [1978]: P.J.Willis, "Derivation and Comparison of Multiprocessor Contention Measures," IEE Journal of Computers and Digital Techniques, Aug. 1978.

Windrush {1985]: Windrush Micro Systems, "EURO-6X : Hardware manual," 1985.

Wulf and Bell [1972]: W.Wulf and C.G.Bell, "C.mmp - A muli-mini-processor," in Proc. AFIPS Conf., 1972, vol. 41, part II, pp. 765-777.

Xilinx [1986a]: Xilinx, "LCA User's Manual," 1986.

Xilinx [1986b]: Xilinx, "LCA Development System," 1986.

Zakharov [1984]: V.Zakharov, "Parallelism and Array Processing," IEEE Tran. on Comput., vol. c-33, no. 1, pp. 45-78.