A MULTIPROCESSOR FOR THE FINITE DIFFERENCE SOLUTION OF FIELD EQUATIONS

by

JOHN HOLME

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

OF THE UNIVERSITY OF LEICESTER

SEPTEMBER 1987

A MULTIPROCESSOR FOR THE FINITE DIFFERENCE SOLUTION OF FIELD EQUATIONS

JOHN HOLME

SEPTEMBER 1987

FOR OUR DAVE

ACKNOWLEDGEMENTS

Professor MacLellan, Head of the Department of Engineering at Leicester University, for allowing my research to take place.

Dr. A.C. Baxter my supervisor for his help during the project and the constructive critisism he has offered in the production of this thesis.

My family, to whom mere words cannot express the debt I owe them, and the thanks that they are due for the support and help they offered when I needed it the most.

Dr. E.M. Warrington a friend who has sacrificed much of his own time to help me on many occassions during this work.

Mr. Philip Brown and Dr. Dimitrious Papadakos for providing me with the graphics routines which enable my results to be displayed.

Dr. E.C. Thomas both for advise and the many useful discussions on the basic principals of my work and help in the preparation of the programs on microfiche.

Mr. Stephen Rowlinson in the University library for his help in locating articles and information.

Messrs A. Brown, C. Cleg and T. Manchester the always helpful technicians of R-Block.

A special mention for Miss Harumi Takahashi, now Mrs Holme, for making my life complete and providing spiritual contentment.

CONTENTS

page no.

CHAPTER 4 Software

CHAPTER 5 Results

CHAPTER 6 Discussion

CHAPTER 7

APPENDICES

REFERENCES

PROGRAM LISTINGS

MICROFICHE LISTINGS

(Contained in the envelope in the binding of the thesis)

CHAPTER 1

CHAPTER 1

INTRODUCTION

1.0 BACKGROUND

The solution of problems governed by field equations is one of the more important areas of scientific computation that is currently limited by the performance of present day computers. Examples of this type of problem include the prediction of the weather, analysis of stress fields in large structures and propagation of electromagnetic radiation. Even the performance of the most powerful computers of the present day, the so called supercomputers, are heavily taxed by the many computational tasks involved in the solution of field equations (Fagan [36] and Fortune [39]). These include the Cray XMP and Cray-2, manufactured by Cray Research Inc., the Cyber 205, manufactured by Control Data Corporation, the Fujitsu VP, manufactured by Fujitsu Ltd., the Hitachi SA-810/20, manufactured by Hitachi, and the NEC SX, manufactured by NEC Inc.[1].

Until recently the state of the art in the computing world has been dominated by the architecture and performance of supercomputers, which are constantly being improved (Perrenod [65]). The supercomputers of the 1960's, like the CDC 6600, were prototypes of what are now called RISC (Reduced Instruction Set Computing) machines, which operate with a much simplified instruction set enabling operations to be speeded up. Present day examples of machines architectured in this way may be found by reference to

---

[1] Further information on the use, performance and architectures of these machines may be found by reference to Russel [71], Watanabe [87], Carruthers [20] and Mendez [62].

Circhanowski [22], Furber and Wilson [42], Hennessy [45] and Brain [17]. RISC machines are becoming popular now with the class of so-called mini-supercomputers, machines that perform and cost an order of magnitude less than the bigger computers. In the future it is possible that recently developed parallel systems may dictate the state of the art machines.

2.0 A PARALLEL FUTURE

In recent years there has been a continuous and rapid development in semiconductor technology in the field of large scale integrated circuits (see Barron [9]). In this respect the processing power of microprocessors has been consistently increased to the point of equalling, and even exceeding, that of much more expensive traditional minicomputers. Placing these processors in some type of parallel architecture could make supercomputer performance levels available, at a cost potentially much less than that of a conventional system.

The technology is now available to form powerful parallel processing building blocks. New VLSI chip making techniques such as triple diffusion fabrication (the so called 3D chips), coupled with computer-aided design, offer the possibility of placing multiple processing elements on a single chip. The MA717 produced by GEC Hirst Research Centre, which introduces parallelism at the bit level, is based on a gated array of full-adders for a number of important arithmetic and signal processing functions (other examples may be seen in Ahmed [3], Dew [32], Kung [53] and Mead and Conway [61]).

Devices such as the Inmos "Transputer" based on VLSI are capable of very high speed calculation, concurrency and the ability to communicate with

other transputers. The Inmos transputer is an example of a specialized VLSI RISC chip, designed with the aim of being linked with other units of the same kind to facilitate parallel processing. The Transputer and its associated programming language OCCAM have been the subject of many articles in recent time[2], all of which indicate that the Transputer will fulfil the essential requirements of a parallel processing element. With a purpose built architecture for an application it is possible that VLSI parallel devices will have a great impact in the multiprocessing world and possibly lead to the predicted "desk top CRAY".

Recent work in the field of superconducting chips (a summary of which can be found in Fagan [36]) is expected to yield a chip which will superconduct at room temperature. The potential increase in the processing power of conventional architectures incorporating these chips is vast, but the potential of a parallel machine comprised of such chips is unimaginable.

3.0 THE PARALLEL WORLD

In 1981 Japan announced her programme of research into parallel processing, dubbed the "fifth generation" of computers, although other countries were engaged in similar research. Parallel systems are currently an active research topic, recent work includes that of Bhuyan and Algrawal [14], Bowra and Torng [16], Colon et. al. [24], Dettmer [29], Dew [31], Dew [32],

---

[2] Transputer hardware and applications may be referenced through Brain [17], Brain [18], Coles [23], Jessope [50], Petre [6], Mattos [60] and Taylor [84].

Reference to OCCAM may be found in Curry [26], Dettmer [30], Fay [37] and Hoare [48].

Fuller et. al. [40], Hillis [46], Purcell [67], Ostland et .al. [64], Leiserson [57], Searle and Freberg [73], Snyder [80], Speitz [81] and Taylor [84] which are not referenced elsewhere; other examples may be found in the text to follow.

Networked computer systems are common place and the practise of attaching a second processor in most home microcmputers is increasing in popularity. Several other types of parallel systems, referenced above, have reached the prototype stage or beyond, these include the Carnegie multimicroprocessor (Wulf and Bell [96]), the Minerva Multiprocessor (Widdows [91]) and the Heidleburg POLYP (Manner [59]). In the supercomputer league, a multiple ransputer system is commercially available with ransputer elements in the form of plug in modules, while Christ and Terrano [21] are claiming supercomputer performance using standard (but powerful) processors in an MIMD architecture (described later).

In Britain the Science and Engineering Research Council (SERC) funded a special research programme on Distributed Computing Systems (DCS), which ended in December 1984. During period of the SERC DCS programme (further details of which may be found in SERC [77] and SERC [78]) a great deal of was given to the problem of programming parallel machines and parallel algorithms. Work in Britain focused mainly on "associative programming"(see Lauer and Hemishere [56]), where problems reduced to arithmetic expressions form the basis of a programming language (Abramsky [1] and Abu-Surfad [2]).

A team at Manchester has built a prototype (MIMD) ring-structured data-flow machine which may have a much wider range of applications than vector or array processors. Imperial College is developing a parallel computer based on applicative languages in an attempt to match the architecture to the needs of the data. Fault tolerant systems are being investigated in broad

programs of research into the design of distributed systems at Newcastle University which can operate satisfactorily despite suffering from problems. Barlow and Evans [7] also produced multiprocessor hardware and Grimsdale et. al. [43] proposed a multiprocessor architecture for real-time applications.

The same period saw the introduction of specialized parallel processing programming languages such as ADA and OCCAM. ADA was introduced as a concurrent language designed for the writing of operating systems, whereas OCCAM is by design the optimum programming language of the Inmos Transputer.

4.0 ARCHITECTURE TYPES

In 1972 Flynn [38] categorised the various types of parallel architecture in an attempt to obtain a macroscopic viewpoint of available computer structures. Flynn's stratification uses the concept of a stream which can simply be defined as being a continuum of data or instructions depending on the context in which 'stream' is being used. These categories have since been used in numerous surveys of computer architectures, the most prolific being Baer [5], Barlow and Evans [6], Bolognin et. al. [15], Haynes et. al. [44], Kuck [52], Reyling [69], Seigel et. al. [72], Seitz [74], Stone [83] and Weissberger [89]. Flynn's main categories of parallel machine are:

(1) **Single-Instruction Stream Single-Data stream (SISD)**

This is the Von Neuman architecture employed by the majority of computers in use today, the architecture of which can be seen in figure 1.1;

(2) **Single-Instruction Stream Multiple-Data stream (SIMD)**

This section may be further sub-divided and are described later.

(3) **Multiple-Instruction Stream Single Data stream (MISD)**

A structure which has a very restricted and specialised use, where actions within an instruction cycle may be overlapped with different actions of consecutive cycles to achieve a higher rate of instruction execution (an example of which is a CYBER 205).

(4) **Multiple-Instruction Stream Multiple-Data stream (MIMD)**

Various types of MIMD architectures exist in this category, these are described later.

Of these types of architecture, it is the SIMD and MIMD type of systems which are showing the most promise in the parallel processing world. These classes of machine, and the subcategories into which they may be split, are now described further[3].

---

[3] Diagrams taken from Bolognin [15] whose work synthesises that of others.

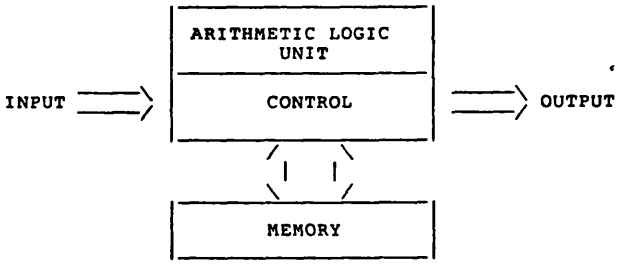


FIGURE 1.1
VON NEUMAN ARCHITECTURE

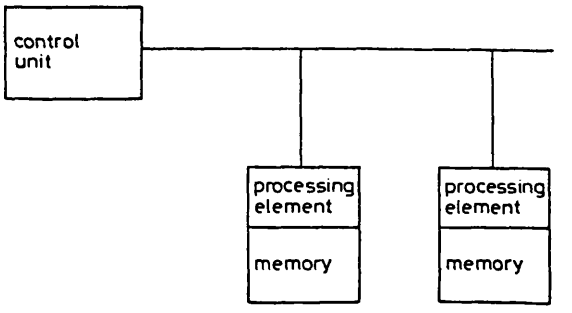


FIGURE 1.2 ARRAY PROCESSOR

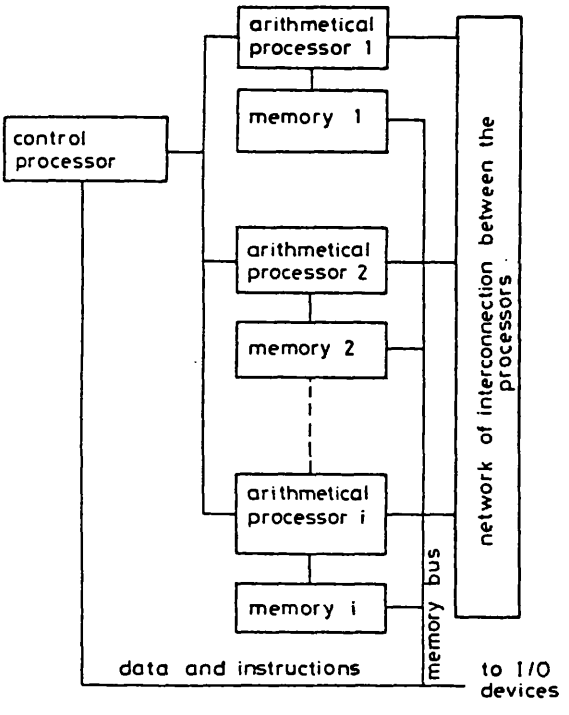


FIGURE 1.3
GENERAL STRUCTURE OF SIMD PROCESSORS

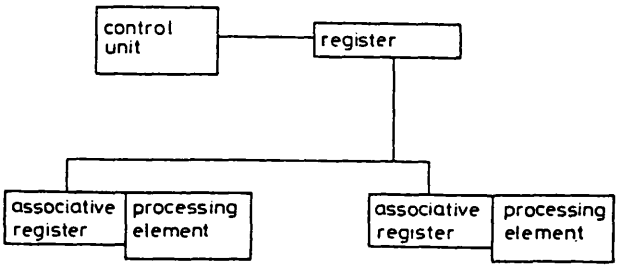


FIGURE 1.4
GENERAL STRUCTURE OF AN ASSOCIATIVE PROCESSOR

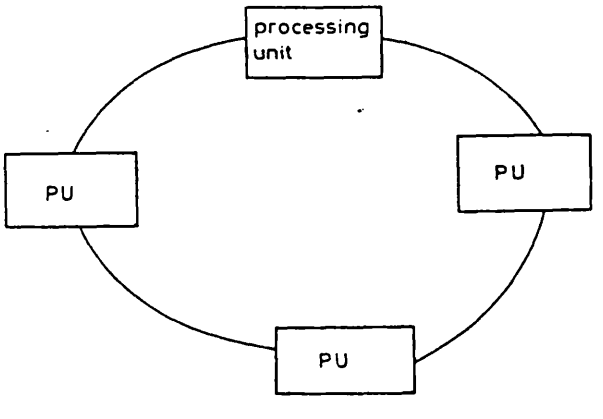


FIGURE 1.5 RING STRUCTURE

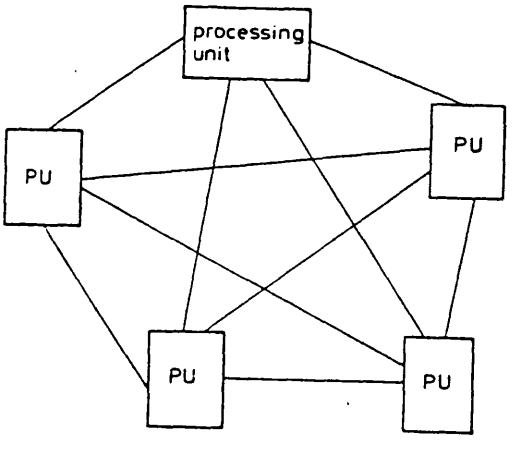


FIGURE 1.6
COMPLETELY INTERCONNECTED STRUCTURE

4.1 SIMD ARCHITECTURES

The SIMD architecture has also been called a "parallel processor" because the same instruction is carried out simultaneously by a vector of processors on a vector of data. Yau [97] provides a comprehensive and detailed look at the architectures of this section, which may be sub-divided as follows:

Array processors Are currently the most efficient form of SIMD architectures from the point of view of performance/cost ratio. The array processor shown in figure 1.2 is essentially a two-dimensional array processor. Such add on units are available for standard machines such as Digital Equipment Corporation VAX computer systems to increase performance. Vector processors also exist, capable of dealing with an n x n vector of data simultaneously. The ILLIAC IV was an early example of a vector processor, further details of which may be found in Barnes et. al. [8]. The ICL Distributed Array Processor (DAP) discussed in the work of Ducksbery [34] and the CLIP machine proposed by University College London (Shaw et. al. [82]) are finding applications in the world of image processing, where the bit level processing of the system ideally suits the digitized image. A large array consisting of 65536 processors in this type of architecture, has been built, and is described in Hillis [46].

Pipeline processors May be considered as a temporal-mulitplexed version of an array processor. Figure 1.3 shows a general SIMD architecture. A certain number of executive functional units are arranged in an assembly line. Each unit accepts new data every δt, thus if there are n units the execution of a process takes ndt units of time. However, there are n processes active at any time, each one residing in one

of n stages through which the evolution of the process passes, hence $(n+k-1)\delta t$ units of time are required to complete k processes (serial execution for comparison would take $nk\delta t$ units of time). The technique is fundamental to the operation of a systolic array processor (described in Kung [54] and Kung [55]), and is also displayed in interleaved memory systems (Burnett and Coffman [19]). It can be seen that pipelining is a very useful tool in the parallel processing trade and is employed as a fundamental aspect of supercomputer operation which incorporate mulitiple pipelines - along with other techniques.

Associative processors Are a type of array processor in which the data elements are not directly addressed. Figure 1.4 shows a typical (but simplified) associative architecture. Processors are activated when a certain relationship, between the contents of a register which is loaded by the control unit and the data contained in the associative registers is satisfied. In general, an associative processor is categorised by the possibility of accessing data through part of it's contents, making it an ideal choice for database application. Other examples of this type of architecture are the Parallel Ensemble Processing Engine (PEPE) and STARAN. The PEPE architecture is described in Yau [97], while the architecture and use of STARAN can be found in Batcher [11], Batcher [12], Rudolf [70] and Davis [27].

4.2 MIMD ARCHITECTURES

In this type of system, parallelism of functions is achieved through the execution of independent tasks simultaneously. Efficiency of the system may depend mainly on synchronization between processes and distribution of tasks to the processors. Speitz [81] used communication between processors to measure how loosely or closely coupled a system may be. In general

further classification of MIMD architectures may be achieved by analysis of the ease of which communication between processors takes place. Speitz's classification was based on the coupling between processors, the more closely coupled a system the more it resembles a multiprocessor network in which each processor is in communication with all others in the network. Examples of MIMD systems follow:

Ring structures These normally exhibit unidirectional flow of messages and low fault tolerance, figure 1.5 shows the ring structure. The Distributed Computing System (DCS) at the University of California uses switches as well as multimessage sends to increase fault tolerance. One of the best known ring structures in this country is the Cambridge ring of linked computers, a second example would include the IBM attached processsor localized support system where up to four System/360 or /370 computers may be linked by Input/Output channel couplers.

Completely interconnected structures These need a communication system of some kind which normally consists of serial links between processors. Serial links are used in preference to bus connections to reduce the number of wires and hence reduce the apparent system complexity, an example may be seen in figure 1.6.

Shared Memory Structure This is a common technique in which communication between processors is achieved by links through a common storage medium (see figure 1.7). Here the common storage is used merely a communication mechanism, rather than fulfilling its usual role as data memory. Performance in such systems increases more slowly with added processors, although there are no obvious bottlenecks in the communication mechanism.

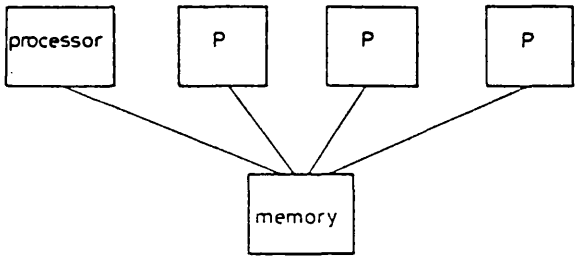


FIGURE 1.7

SHARED MEMORY STRUCTURE

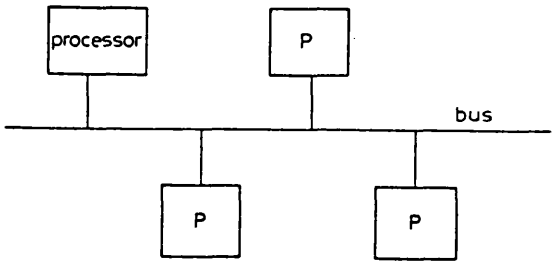


FIGURE 1.8

GLOBAL BUS STRUCTURE

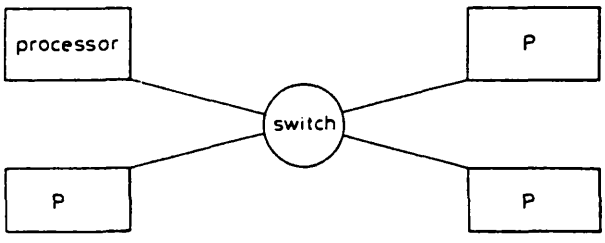


FIGURE 1.9 STAR STRUCTURE

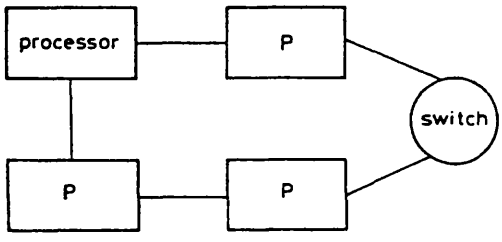


FIGURE 1.10

RING STRUCTURE WITH SWITCH

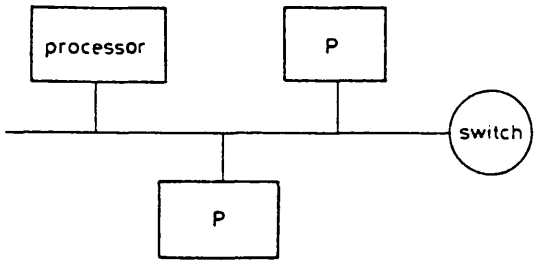


FIGURE 1.11
BUS STRUCTURE WITH SWITCH

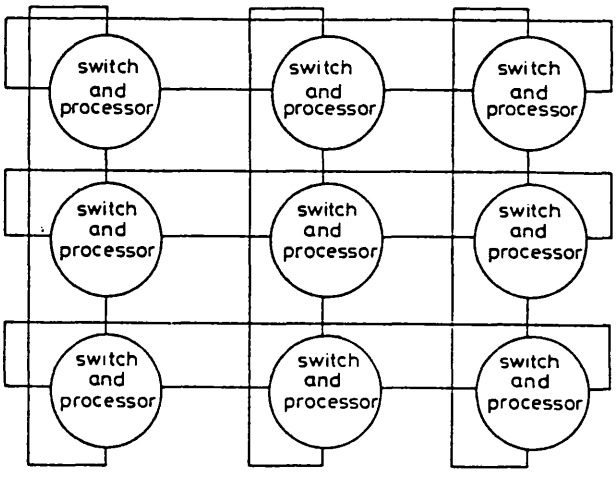


FIGURE 1.12

REGULAR NETWORK STRUCTURE

Global bus structures These are shared bus networks, an example may be seen in figure 1.8, these were widely used in the aerospace industry (1977), see Enslow [35]. With only one bus between n processors, if large amounts of communication need to take place, the bus becomes even more of a potential bottleneck than in a typical SISD Von Neuman architecture.

Indirectly connected structures There exists a structure for communication of messages known as a 'switch' which makes a processor independent of the physical architecture, one of the best interpretations of a switch may be obtained from Haynes et al. [44]. A switch is also able to protect processing elements from possible harmful external effects and reroute communications if faults are detected. This section can again be divided:

Star Structure - This has a switch in the mid point of all communication paths. An example of this is the VAX cluster hardware, of figure 1.9, installed at Leicester University in 1985. Here the switch is a hardware connection linking the two VAX computers and intelligent memory controllers. IBM's Network/440 is also closely linked to this type of structure, having all communication lines passing to a central controller.

Ring with Switch - This structure can be seen in figure 1.10, all messages pass through the switch and are correctly addressed. The switch however is prone to jamming if a large amount of communication takes place. An example of this might be SPIDER, a data communication system used in the Bell laboratories in use in 1975.

Bus with Switch - This type of architecture is not widely used. Systems of this type (figure 1.11) have been constructed for the US Navy, see Enslow [35] and another similar network called ALOHA using a radio bus and based on an IBM 360/65 was also built.

Regular networks - Ring structures can be considered a special case of regular networks, pictured in figure 1.12. They exhibit good fault tolerance. To increase processing power a certain number of processors must be added.

Irregular networks - These are specialised and tend to be problem oriented, a diagram is therefore of no real value here. An example of a tree architecture of this type can be seen in Despain and Patterson [28].

5.0 METHODS OF INTERCONNECTION

A shared bus This is the simplest method of connecting processing units, and the least expensive. It has a disadvantage in that a shared bus could become a bottleneck if large numbers of processors are used.

A serial/parallel link A communication link may be constructed between processing units, which may be serial or parallel. These communication links are optionally fed into switches as seen in the previous section.

A multi-port system This method of interconection uses memory, which can be shared between a number of processors, as a communication medium.

6.0 ADVANTAGES OF THE MULTIPROCESSOR SYSTEM

The advantages of the multiprocessor system are:

6.1 High performance/cost ratio;

6.2 Excellent reliability and ability to function in a degraded manner in case of technical trouble;

6.3 Modular nature and ease of expansion.

These advantages become more or less important according to the type of architecture being considered; they also exist for conventional computer networks but are intensified in microprocessors due to their low cost. The following section looks at these advantages in more detail.

6.1 Performance/cost ratio:

Figure 1.13 shows the performance/cost of various systems as a function of cost itself. The line U displays the average performance/cost ratio for single processor systems within the data. The line IM shows the ideal relationship of the performance/cost ratio with cost in multiprocessor systems, with every increase in cost being paralleled with a corresponding increase in performance. The actual relationship of multiprocessor systems can be seen as the line RM, where it is claimed that the increased cost of connections and problems of synchronization mean the performance rises more slowly, resulting in a degradation from the ideal.

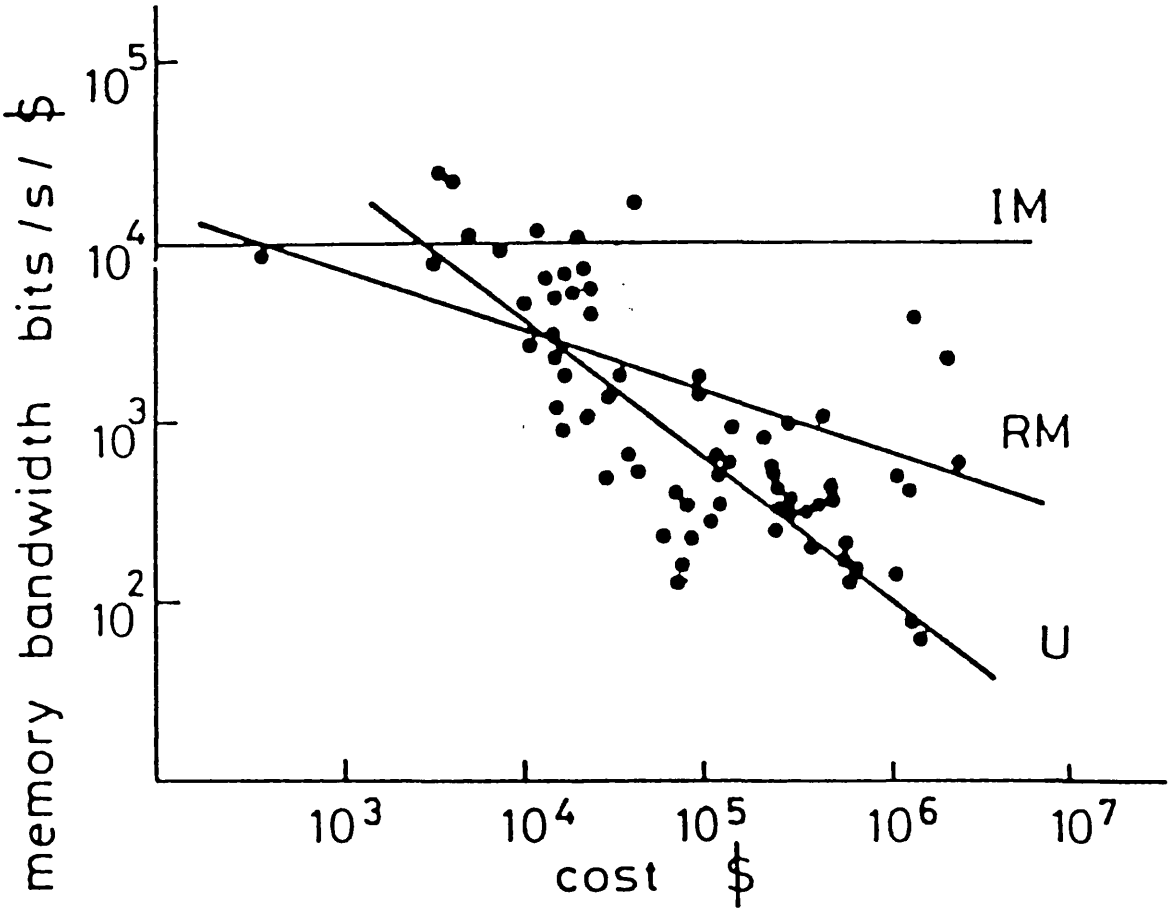


FIGURE 1.13 PERFORMANCE/COST AGAINST COST

6.2 Reliability and fault tolerance

Fault tolerance is an important consideration in a Muti-processor system where the Achilles' heel may turn out to be a small fraction of the processing or communication hardware. In order to present statistics on the reliability of the architectures discussed certain data is usually quoted (further details of which may be found in the literature):

a. Mean Time To Failure (MTTF).

b. Mission time. This is the time to reach a certain degraded performance.

c. Probability of fault avoidance.

d. Maximum number of failures.

6.2.1 Fault Tolerant Machines

Due to the increased complexity normally associated with a multiprocessor system, the probability of failure in the system is usually high. In order to increase reliability within a system various proposals have been suggested and/or implemented, these include hardware redundancy techniques which employ multiple copies of system elements, these copies being switched in once a fault is detected. Examples of fault tolerant systems, the techniques of which are discussed in Siewiorek [75] and [76], include:

(1) Triple Modular Redundancy;

(2) 'n' Modular Redundancy;

(3) Adapting Voter Networks;

(4) Threshold Voter Networks.

There are many examples of fault tolerant systems: The Sperry Rand UNIVAC

110 which allows on-line maintenance, CDC's CYBER with dual central processing units, the US army's PEPE used in conjunction with ballistic missile monitoring and STARAN - further details of which may be found in Enslow [35] and Fung [41]. Fault tolerance is also discussed in respect of communication Seigel [72] and the iAPX processor hardware in Cox et al. [25] and Witten [95].

6.2.2 'Graceful Degradation'

Graceful Degradation is an important concept in fault tolerant systems. If a fault occurs rather that the entire system being rendered useless, the idea is to keep the entire system running with the faulty unit but in a degraded manner. If this can be done the system is then said to have been gracefully degraded.

6.3 Modular nature and ease of expansion

A multiprocessor system is by definition modular, since it is comprised of component processors. The benefits of a multiprocessor system are evident: The modular nature of a multiprocessor system means that the system offers ease of expansion and a system built for one function can be upgraded easily by the addition of similar component modules. It is also possible that a multiprocessor system can be reconfigured either by adjusting hardware or through software. Examples of this can be seen in Dove [33] and Jessop [51].

7.0 DISADVANTAGES OF MULTIPROCESSOR SYSTEMS

In parallel systems the ideal performance is degraded by difficulties which cause problems in system realization. In brief these problems are as

follows, and further details of these can be found in the literature:

a. Synch ronization and Communication between processors, discussed by Seigel [72], Baskett and Smith [10], Barlow and Evans [7] and Anderson and Jenson [4];

b. The partitioning of problems which have inherently serial sections, see Haynes et al. [44] and Kuck [52];

c. Efficient use of hardware resources (Shoja [79]);

d. Branching problems in relation to performance, see Flynn [38];

e. General lack of operating systems and overseeing of the functions of the component processors;

f. Choice of topology of connection.

CHAPTER 2

CHAPTER 2

OBJECTIVES

1.0 A DESCRIPTION OF A FIELD EQUATION

Mathematically field equations are those equations relating to a field of interest which may be characterized by a certain mathematical equation called Laplace's equation. This can be seen here, in rectangular coordinates, for a conservative field:

$$\frac{\partial^2\phi}{\partial x^2} - \frac{\partial^2\phi}{\partial y^2} = 0 \quad .$$

In a non-conservative field, the potential function becomes Poisson's equation (further details of which can be found in most of the relevant literature). Depending on the physical properties of the field, the equation which describes the field may become very complex. The Navier Stokes equation of fluid flow (even when simplified by the assumption of constant viscosity) is an example of this and can be seen as follows:

$$\rho \frac{DV}{Dt} = \rho g - \nabla p + \mu \nabla^2 V \quad ;$$

the derivation of which can be found in White [90] and further details of which may be found in the literature. Solution of the Navier-Stokes equation of fluid flow yields solutions of pressure and velocity at points within the fluid, vital in such applications as weather forecasting and the aerodynamic design of high lift aerofoils or turbine blades. The solution of the Navier-Stokes equation by finite difference techniques is an example of one of the greatest demands made of computers.

The solution of field equations may be found by three methods: A direct mathematical solution; a Finite Element method or a Finite Difference method. A direct mathematical solution of complex field equations is not possible with present day mathematics. Finite element methods are now emerging as a means for the solution of field equations but this work concentrates on the finite difference method of solution, as this method promises a greater potential benefit in a parallel computer architecture.

2.0 A PARALLEL SOLUTION OF FIELD EQUATIONS BY FINITE DIFERENCE TECHINQUES

The solution of field equations by finite differences involves covering the field of interest with a hypothetical mesh and reducing the field equation, which is difficult to solve, to a set of finite difference approximations at the nodes of the mesh, which may be solved using data at local nodes only. Solution begins by imposing a set of boundary conditions on appropriate nodes of the mesh. The classic solution using conventional general purpose machines consists of iterative passes of the whole mesh, a node at a time, until the solution advances to a point where a steady state condition is reached. If the problem is dynamic, as in weather forecasting, the same procedure must be repeated for different time increments. Clearly the sheer number of calculations that have to be made, due to the iterative nature of the calculations along with the need to keep the finite difference field equations stable, means that the time taken to reach a solution on even the largest supercomputers is appreciable. Trivial problems have execution times which can be measured in hours on a typical minicomputer.

The process is essentially a series of operations on the whole mesh, although each node can be processed in parallel. The method of solution,

which involves repeated iteration involving updating of various parameters at the nodes of a mesh, lends itself to a parallel solution. The inherent parallelism of these algorithms coupled with the high performance/cost ratio of present processors indicates that a multiprocessor machine built specifically to perform these algorithms is feasible. Such a machine has the potential of providing a cheap high power field equation solver, but because the algorithm is only similar and not identical at each node or iteration, a special architecture may provide the best performance.

3.0 OBJECTIVES

In brief the aim of this work was to propose a design for a high performance field equation solving parallel computer. To gain the experience needed to put forward such a design the proposal is to build a prototype system, based on current microprocessors, which will perform the task at much lower cost than a mainframe computer. The architecture of such a parallel machine should be matched as closely as possible to the method by which algorithms solve the field equations. The hardware of the prototype machine should ensure that the system does not degrade the potential performance of the algorithms. In order to demonstrate the efficiency of the prototype, within a limited time span, one set of the many sets of field equations (see chapter 4) which could be chosen, will be implemented on the processing element array being proposed. Since the solution of field equations by finite differences uses the same basic technique, a purpose built multiprocessor for field equations will be able to implement the solution of any field equations set.

4.0 THE CHOICE OF ARCHITECTURE

Using finite differences the updating of the various parameters at each node can be achieved by using the values of these various parameters held at that node and neighbouring nodes only. Thus, if the field of interest is subdivided with each subdivision allocated to its own processing element (PE) to perform the updates on the nodes of that subsection of the field, the time taken to reach an overall solution will be greatly reduced by virtue of the fact that many nodes can be updated in parallel. The stability of the finite difference equations being solved may be increased by a parallel solution, due to reduced propagation of errors.

An MISD type of architecture was disregarded as an option for the machine as it would not offer the required parallel access to the data that is required. Of the SIMD types only the array processors would be able to meet the requirements. It is this type of architecture that currently dominate the area of field equation solution. The inherent synchronous operation of this architecture is incompatible with the need for a different method of calculating values at the boundary nodes of a problem. This architecture would also impose restrictions and a degree of inflexibility in the choice and use of the grid on which solutions will be found. SIMD architectures are based round a single processor, the performance of the system thus tends to be dominated by the power of this processor. Any improvement in the performance of the system can only be achieved by using a more powerful processor and building a larger system. The performance of a MIMD system on the other hand can be improved simply by adding extra processing modules. An MIMD architecture offers the flexibility required for more general solutions but would have to be carefully chosen so as to avoid the many degradations that can befall such systems. The degradations reported by many workers in MIMD architectures can be attributable to factors of

communication and shared resources, each of which becomes more or less important depending on the type of MIMD architecture under consideration.

Of the many types of MIMD architectures available a 4 x 4 array of PEs of the MIMD type was chosen for the parallel solution of fields equations. In many respects the architecture chosen to implement the parallel execution of field equations bears a close resemblance to the architecture suggested for the parallel solution of the Navier-Stokes equation in 1977 (see RAND [68]). The report contains suggestions for a cell based architecture capable of direct parallel computation of the Navier-Stokes equations; A similar machine is suggested by Weiman and Grosch [88] later in the same year. In the light of the technological change within the semiconductor industry specifically in the area of VLSI, the architecture being proposed here, where a PE deals with more than one node (and the data associated with it), promises to be an improvement over the system proposed by RAND [68]. The architecture chosen is also similar to that of Christ and Terrano [21], but the reduced complexity of both the processing element and the connection network being proposed here, are expected to keep any degradations in the system to a minimum.

A 4 x 4 array size is large enough to demonstrate the potential of a larger array and yet still be cheap enough to build on a limited budget. The array is to communicate via Shared Memory blocks with nearest neighbours. Using shared memory means that the nodes on the boundary of a PE can be accessed with the minimum of overhead, which can occur in some systems which employ 'test and set' means of access to shared resources. Shared memory also has the advantage that the data being accessed is as up to date as possible, since the nodes in shared memory are part of the problem mesh within the PE. This architecture will not only closely match the algorithms used in the solution but also promises to exhibit the least degradation in the

maximum possible performance of such a system. Adopting this architecture also means that:

(i) Data are easily updated and always available in the most up to date values by a requesting PE;

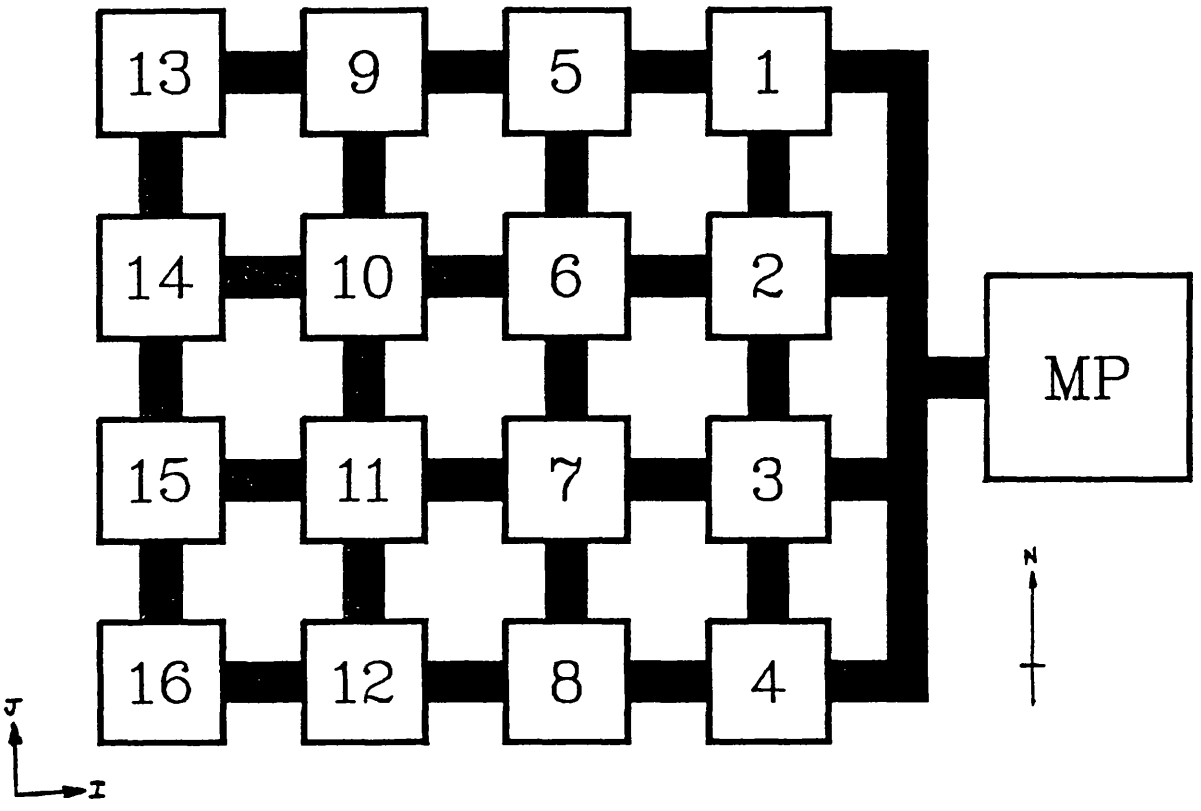
(ii) The system can be easily expanded.

To provide the necessary input and output for the array a Master Processor (MP) is interfaced to one edge of the array. Because of the unsteady nature of the finite difference equations the MP will also serve as an overseer as calculations progress for the more complex forms of field equation. Addition of the MP gives rise to two distinct types of communication which may be seen in the system: Global and Local communication. These are defined as follows:

Global Communication - data passing through SM whose destination or origin is the MP;

Local Communication - cross border nodal data accesses passing between PE.

The proposed MIMD system can be seen in figure 2.1 which shows the PE array and the Master Processor. Ideally the MP would be able to communicate directly with each of the PEs, however the increased cost and the extra complexity this would cause in the basic PE board, coupled with the fact that global communication will be minimized, lead to the adoption of the architecture of figure 2.1. The adopted architecture does mean that the PEs nearer the MP have a greater commitment to global communication since they



SCHEMATIC DIAGRAM OF THE MULTIPROCESSOR

Processing elements labelled 1–16
Master processor labelled MP

FIGURE 2.1 PROPOSED MIMD ARCHITECTURE

have to act as the link to the MP for all the PEs more remotely connected from the MP than itself, and it is for this reason global communication will be minimized. The effect of excessive amounts of global communication can be seen later in this chapter. The choice of an asynchronous system means that all PEs are as active as possible, and none are halted due to contentions taking place elsewhere in the PE array.

5.0 THE POTENTIAL SPEEDUP OF THE SYSTEM

Following the work of Baxter and Holme [13] the potential speedup of the multiprocessor may be found by consideration of the probabilities of contentions and consequent lockouts in the system. If the speedup is defined as the increase in processing power over that of a single processor, then the effects of local and global communication on the potential speedup of the system can seen to be dependent on three factors:

a. The way data are stored;
b. The number of nodes each PE handles and the number of neighbours surrounding an element;
c. The amount of global communication.

These factors are now analysed such that a model can be developed and from this the potential speedup of the multiprocessor can be estimated, and it s performance compared with other systems.

5.1 The way data are stored

Consider an array of n x n PEs each of which may be referenced by an I and J component of a hypothetical grid covering the array (figure 2.1) and let

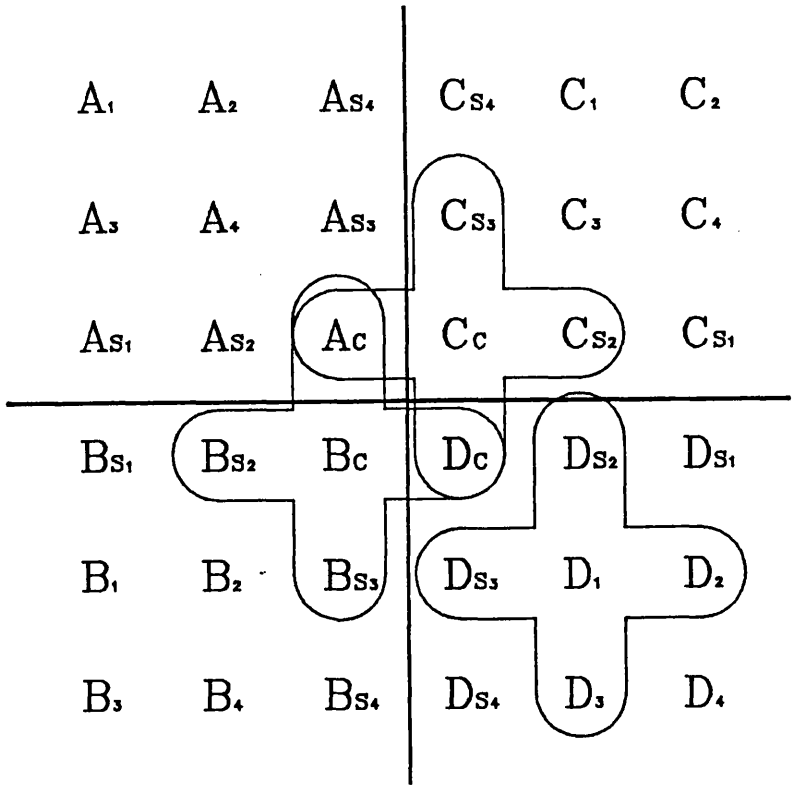
the number of grid points of a symmetrical 2-dimensional field problem that each PE is dealing with be p x p.

A general PE has 4 neighbours (N,E,S and W) each of which must have access to the nodes on the boundary of that processing element. Access to these points is provided by storage in the shared memory (SM) blocks, where boundary nodes to the East and South of a PE are held in SM blocks on the same printed circuit board as that PE, and the boundary points to the North and West are held in the SM blocks of the PE in that respective direction.

The way in which the problem mesh is partitioned and allocated to each PE means that the corner nodes of a PEs local mesh become a special case, because for an update to be performed the node needs to be accessed by three PEs and not at the most two as in the case of all of the other nodes. This special case can be seen in figure 2.2, which shows (schematically) the intersection of the local meshes of four PEs and the points usually required for an update to be performed. The ability to share memory between two PEs is relatively simple but expanding this to cope with three PE, which would be needed for the corner points of the p x p array of points would not be justified on the basis of a cost/usage factor. Instead the corner nodes are stored in local memory and copied to the appropriate areas of the shared memory block for access by a neighbouring PE. This can be seen in figure 2.3, and results in a maximum of two PE requiring access to any one node.

5.2 The number of nodes and the number of neighbours of a PE

Given that the data are stored as above and that each PE deals with p x p points then the probability of a PE operating in a specific SM block, given that it is not globally communicating, is simply the ratio of SM data



Updating D_1 requires D_{S2}, D_{S3}, D_3, D_2

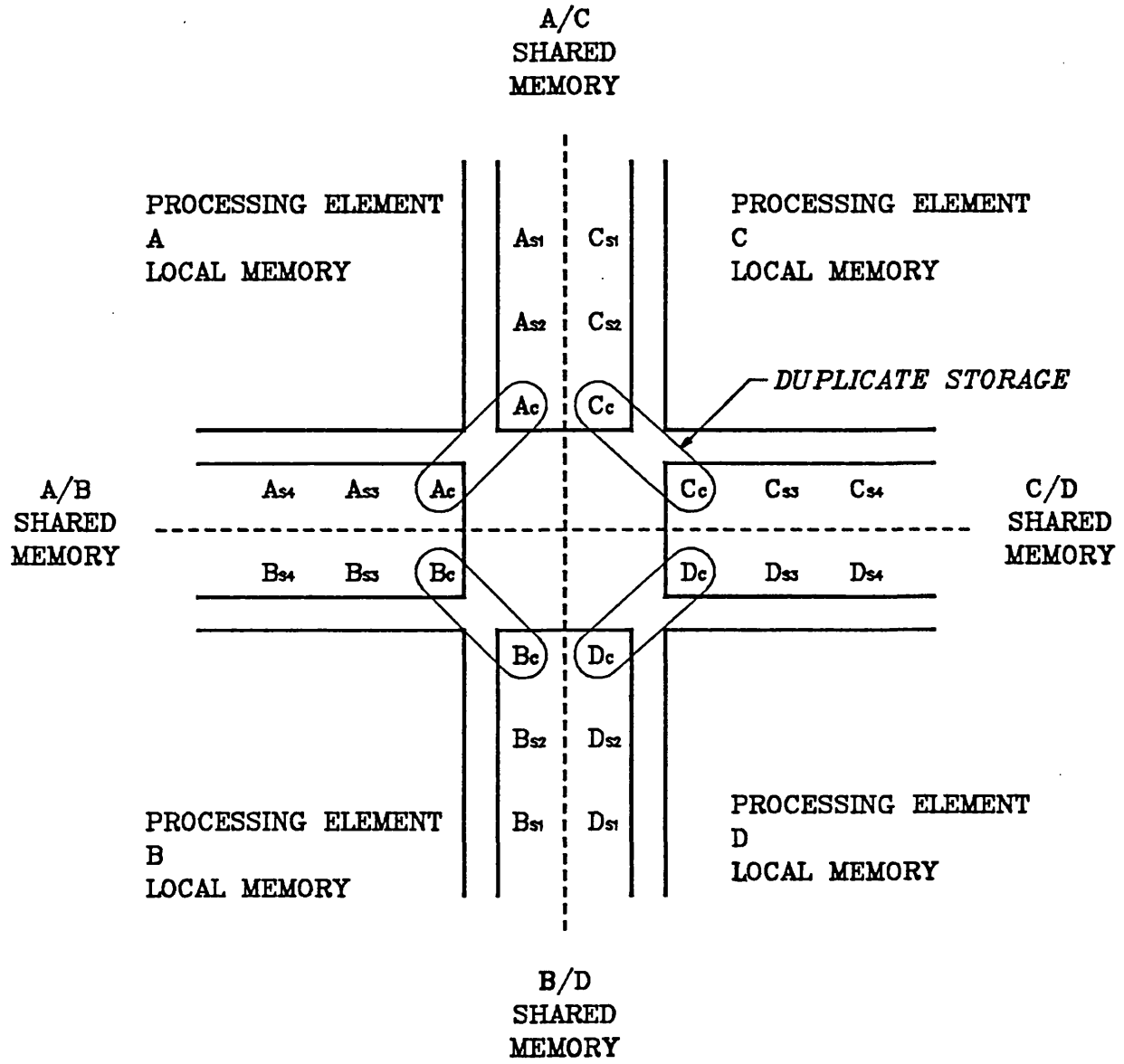
Updating C_C requires C_{S3}, A_C, D_C, C_{S2}

Updating B_C requires A_C, B_{S2}, B_{S3}, D_C

——— PE BOUNDARY

THE INTERSECTION OF FOUR PROCESSING ELEMENT BOUNDARIES SHOWING THE NEED TO SHARE CORNER NODES WITH THREE PROCESSING ELEMENTS, THE HOST AND TWO NEIGHBOURS.

FIGURE 2.2 CORNER NODES BEING SHARED BETWEEN THREE PROCESSING ELEMENTS



------ PROCESSING ELEMENT BOUNDARY

THE INTERSECTION OF FOUR PROCESSING ELEMENTS SHOWING THE DUPLICATE ST ORAGE OF CORNER NODES SUCH THAT EACH NODE IS ONLY ACCESSED BY A MAXIMUM OF TWO PROCESSING ELEMENTS. THE HOST AND ONE NEIGHBOURING PROCESSING ELEMENT.

FIGURE 2.3 STORAGE OF CORNER NODES TO ENSURE ACCESS BY TWO PROCESSING ELEMENTS ONLY

points/total data points. This ratio is also dependent on the number of adjacent PE since this alters the amount of SM a PE will need to use. These probabilities can be seen in table 2.1.

No. of neighbours	p(in a SM block given the PE is not globally active)
2	p/(p*p+1)
3	p/(p*p+2)
4	p/(p*p+4)

TABLE 2.1 Probabilities[1]

Let PS(N) denote p(in a SM block due to calculation) if the processor has N neighbours then, from the following the need to look at the amount of global communication is apparent:

p(in an SM block due to calculation) = p(in an SM block given the PE is not globally active) x p(not globally active).

5.3 Global communication

Let Cp be the fraction of the time spent processing by the most remotely connected PE from the MP along one row of the PE array and let Cc be the fraction of time spent in global communication where Cp+Cc=1. If PC(I,J,K) is defined to be the probability of the processor at reference node I,J using the SM block in direction N,E,S and W (K=1 for North, K=2 for East, K=3 for South etc) due to global communication, since global data is always transferred E-W this will always be zero in the case of SM blocks N and S hence:

[1] Needs a correction for p=1.

$$PC(I,J,2) = PC(I,J,4) = J*Cc*0.5 \quad \text{(EAST and WEST)}$$
$$PC(I,J,1) = PC(I,J,3) = 0 \quad \text{(NORTH and SOUTH)}$$

If NEIG(I,J) is then defined as being the number of neighbours of a PE at the grid reference I,J it can be seen that:

$$p(\text{processor } I,J \text{ is in a particular SM block}) = p(\text{in SM block due to calculations}) + p(\text{in SM block due to global communication})^{2}$$

$$= Cp*PS(\ NEIG(I,J)\ ) + PC\ (I,J,K)$$

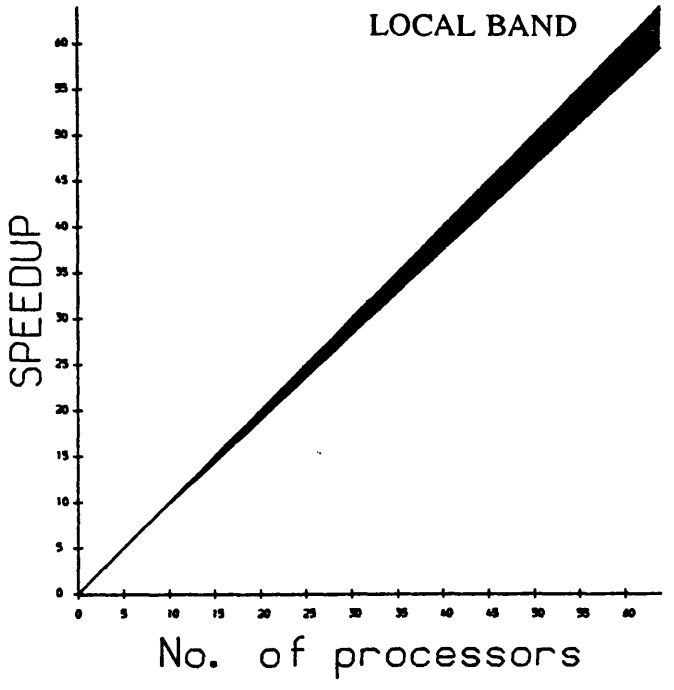
$$= (1-Cc)*PS(\ NEIG(I,J)\ ) + PC\ (I,J,K)$$

5.4 Potential speedup

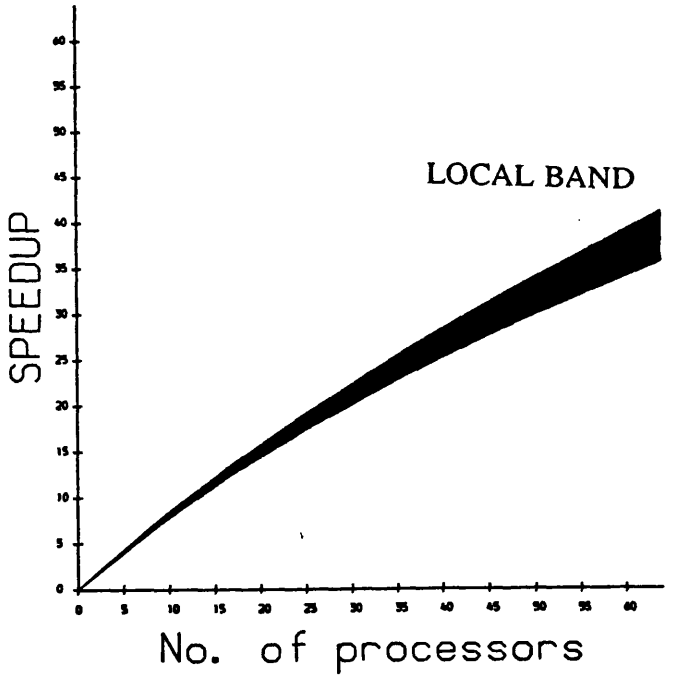
The ability to calculate the probability of being in any SM block means that it is possible to calculate the probability of a contention in a SM block, by finding the likelihood of two PEs requesting use of the same block of SM at a given time. If this data is summed for all possible contentions over the architecture it is then possible to calculate the number of lockouts there are likely to be at a given time. A lockout is the consequence of a contention over a shared resource since only one PE may access the shared resource at a time, the PE which is made to wait until the other PE is finished is said to be locked out. Lockout data gives a measure of the efficiency of the system, since this is the only cause of degradation in the hypothetical system, from which a speedup figure can be found. A program (microfiche listing 1) based on this model using probabilities was developed to enable speedup data to be found. Speedup data was found for a range of global communication factors and for a range of architecture sizes. The results may be seen in figure 2.4, where if

---

[2] Again a correction factor is needed for PEs which only communicate globally to the East.

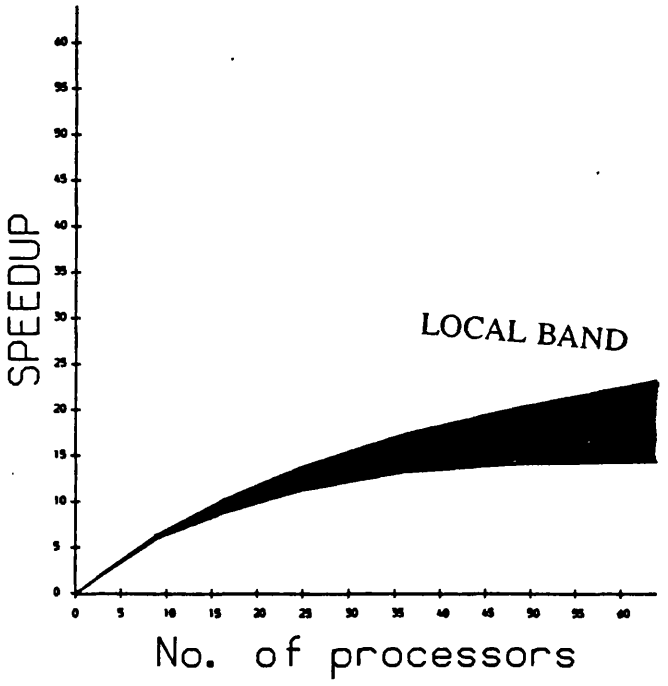


(a) *ESTIMATED BAND OF OPERATION WITH A GLOBAL COMMUNICATION FACTOR OF 0 %*

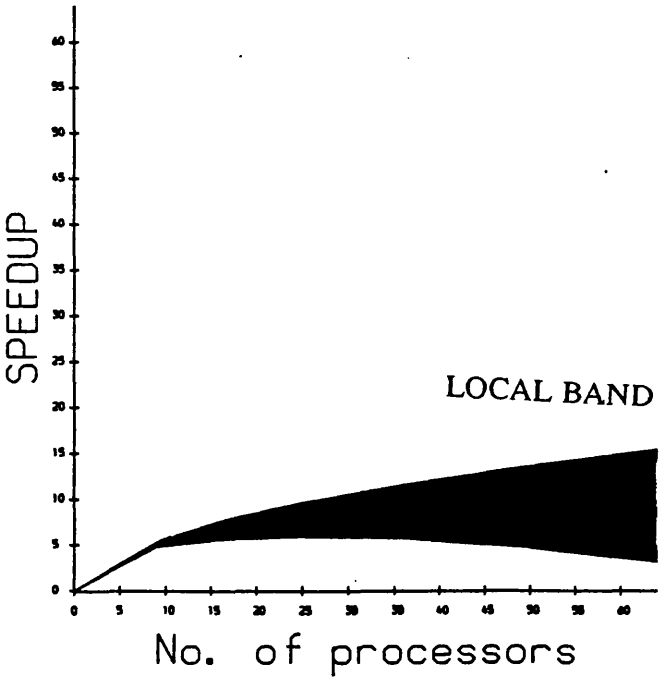


(b) *ESTIMATED BAND OF OPERATION WITH A GLOBAL COMMUNICATION FACTOR OF 5 %*

FIGURE 2.4 THE EFFECT OF GLOBAL AND LOCAL COMMUNICATION



(c) *ESTIMATED BAND OF OPERATION WITH A GLOBAL COMMUNICATION FACTOR OF 10 %*



(d) *ESTIMATED BAND OF OPERATION WITH A GLOBAL COMMUNICATION FACTOR OF 15 %*

global communication is minimized the expected speedup factor for a 16PE system is 15. Figure 2.4 displays the dramatic effect of different levels of global communication in degrading potential speedup of the system. It is clear that global communication must be kept to a minimum if significant speedups are to be obtained. The effect of local communication is displayed as a band, in which the system will operate at that level of global communication.

6.0 THE EXPECTED PERFORMANCE

The level of global communication which will be needed to properly oversee the system is dependent on the problem being run, but is not expected to be above the level where Cc=10%. Performance figures based where possible on bench marks, and where not on similar criterion, were calculated for six commercially available machines ranging from microcomputers to minicomputers. The expected performance of the multiprocessor based on the estimate of Cc=10% was also calculated. Figure 2.5 shows these performance figures where it can be see that the machine is expected to operate out of the main band (shown in light shading) of commercially available machines with a performance better than that of a minicomputer and at a fraction of the cost.

7.0 SUMMARY

Calculations of the effects of inter-processor communication and process control have shown that the degradation of performance reported by many workers can be avoided by careful design. An array of asynchronous parallel PEs of the Multiple Instruction-stream Multiple Data-stream (MIMD) type, dedicated to the solution of field equations, with communication limited to nearest neighbours, would not only closely match the algorithms but also

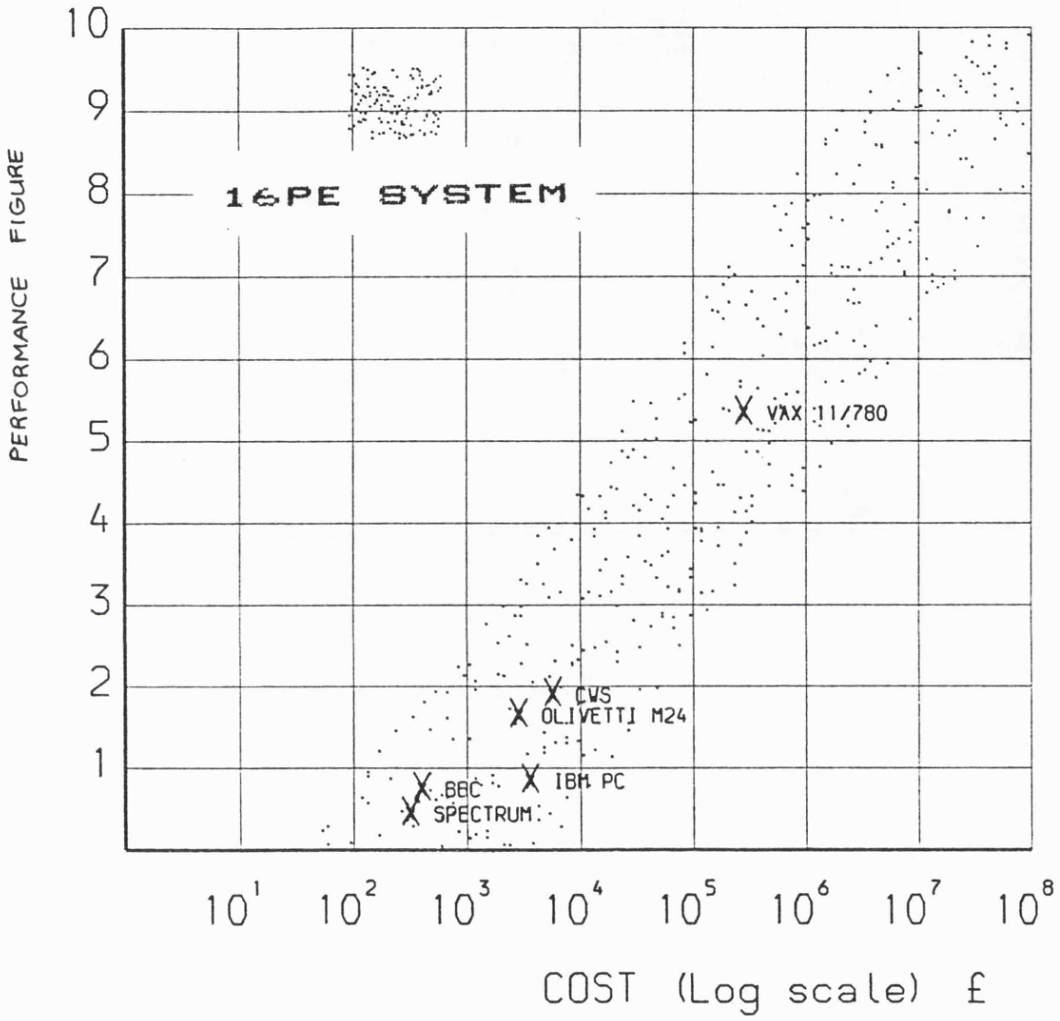


FIGURE 2.5 THE EXPECTED PERFORMANCE OF THE MULTIPROCESSOR

exhibit very little degradation in the maximum possible performance of the array. The results from the prototype system should therefore display linear speedup for increasing numbers of processors, and thus pave the way for a more powerful array to be constructed.

Performance tests are expected to show that microprocessors in parallel can achieve the same processing power as that of a powerful minicomputer, at a fraction of the cost. More powerful processors than those employed are available (eg. the Inmos Transputer) and are expected to yield a field equation solving machine many times more powerful than the supercomputers in use today, at a price that could be afforded by any company involved in the area.

CHAPTER 3

CHAPTER 3

HARDWARE

1.0 INTRODUCTION

The hardware implementation of a multiprocessor for field equations involved the construction of a 4 x 4 array of PEs, along one edge of which was interfaced a Master Processor for overseeing purposes. This can be seen schematically in figure 3.1 (reproduced from figure 2.1 of the previous chapter). Interfacing the Master Processor to the processing element array necessitated the construction of an interface board. A status monitor board was constructed to enable the status of the PEs within the array to be viewed dynamically. The hardware relating to a PE board, the Master Processor interface board and the status monitor board is described in this chapter.

2.0 THE PROCESSING ELEMENT ARRAY

The method proposed in the previous chapter, for the optimum solution of field equations by finite differences, involves equal portions of the problems being allocated to each PE of the multiprocessor, with the nodes at PE boundaries held in shared memory boundaries. To implement this a 4 x 4 array of PEs was chosen as it contains enough PEs to be representative of a larger system and still be constructed with acceptable costs. A general PE needs to be able to access the boundary nodes of a problem held in shared memory to the North, East, South and West of itself. Each PE resides on its own printed circuit board (pcb)[1] with shared memory to the East and South being located on the PE board, while shared memory to

the North and West are held on the respective neighbouring PEs. Two ideas for the physical construction of the multiprocessor array were considered, one involved construction of a mother board into which all the PEs would be plugged and the second in which all the connections would be made using ribbon cable. Due to both the complexity and cost of a mother board system, a system using ribbon cable was adopted. The 4 x 4 array of PEs can be seen schematically in figure 3.1. Figure 3.1 also shows the numbering scheme for each of the PEs, this unique identification is held in the Read Only Memory of a PE[2] and is used in the packet switching of messages between the Master Processor and the array. The orientation of the array elements with respect to the Master Processor is also shown. The physical PE array can be seen in figure 3.2, where the slots for the boards can be seen to be offset from each other in the same row to minimize the connection distance between PE.

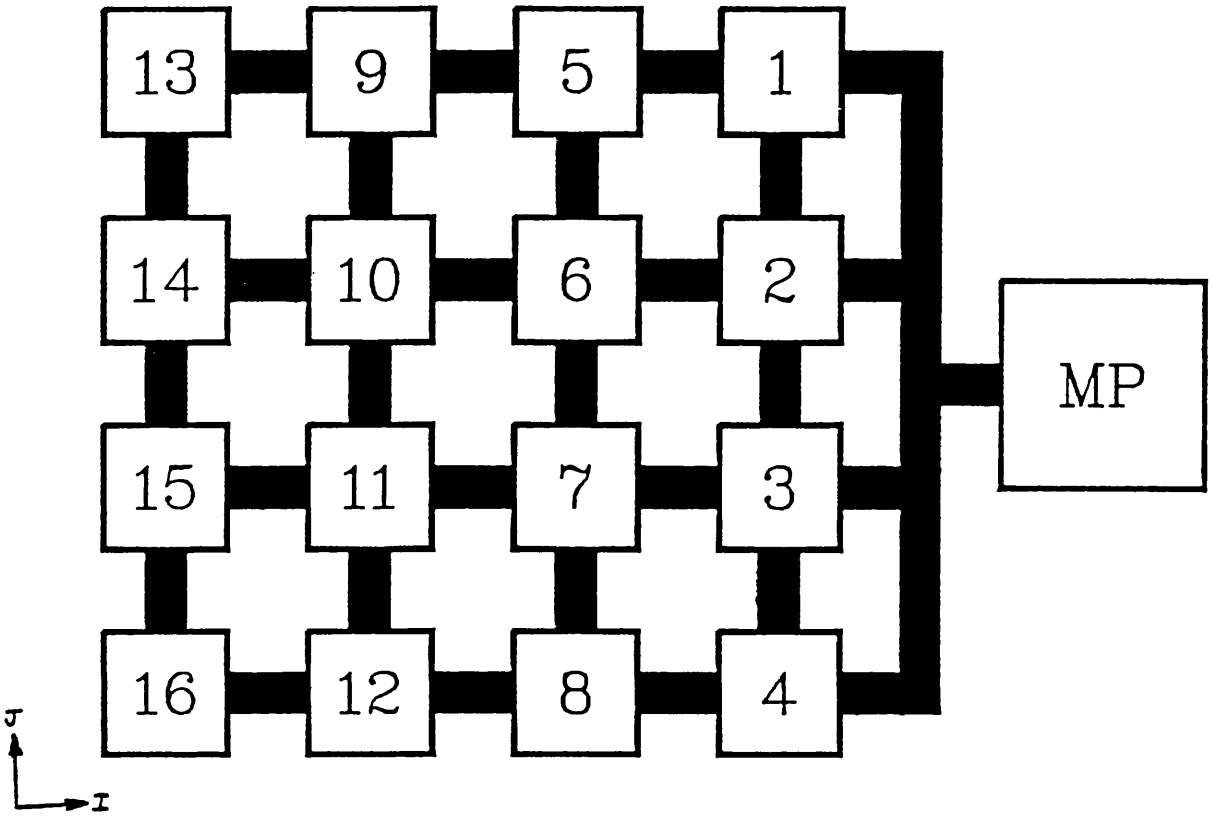
The PE pcb thus has four ports to the North, East, South and West. The connector chosen was a 34-way IDC type, which gave the required number of connections for shared memory accesses, control signals for shared memory access and interrupt signals to pass between PEs. The position of these ports can be seen schematically in figure 3.3, their actual position can be seen in figure 3.4 (which shows the position of the I.C.s) and in figure 3.5 which shows a populated board. The pin connections for ports where shared memory is resident on the board and off the board can be seen in appendix A.

The ratio of shared memory to the local memory (designated for data usage) within a PE was chosen to make the solution of non trivial sized problems

---

[1] The pcb was designed on the Racal-Redac REDBOARD system.

[2] The unique identifier is a byte quantity, stored at $FFF4.



SCHEMATIC DIAGRAM OF THE MULTIPROCESSOR

Processing elements labelled 1–16
Master processor labelled MP

FIGURE 3.1 THE PROPOSED SYSTEM

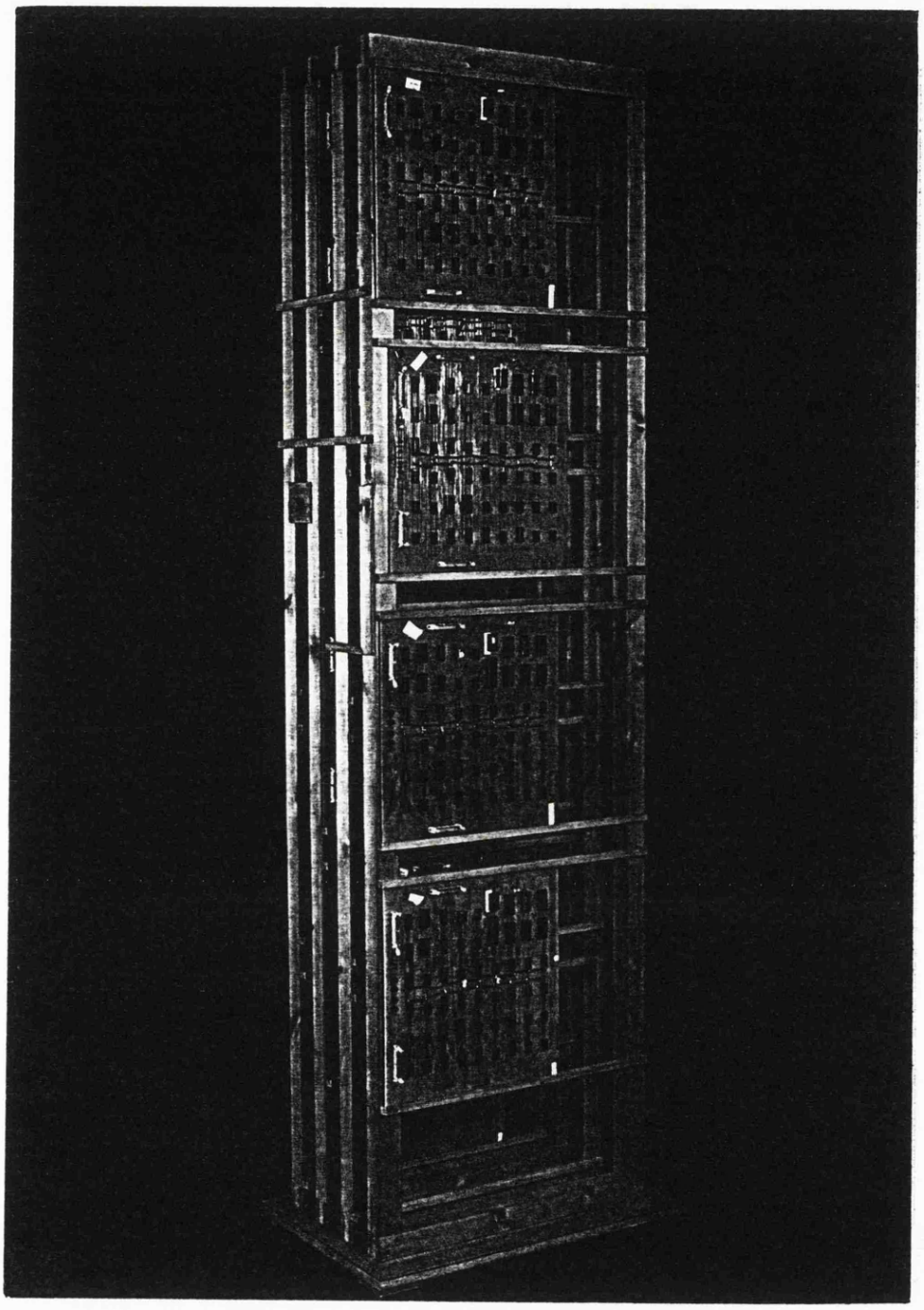


FIGURE 3.2 THE PHYSICAL MULTIPROCESSOR

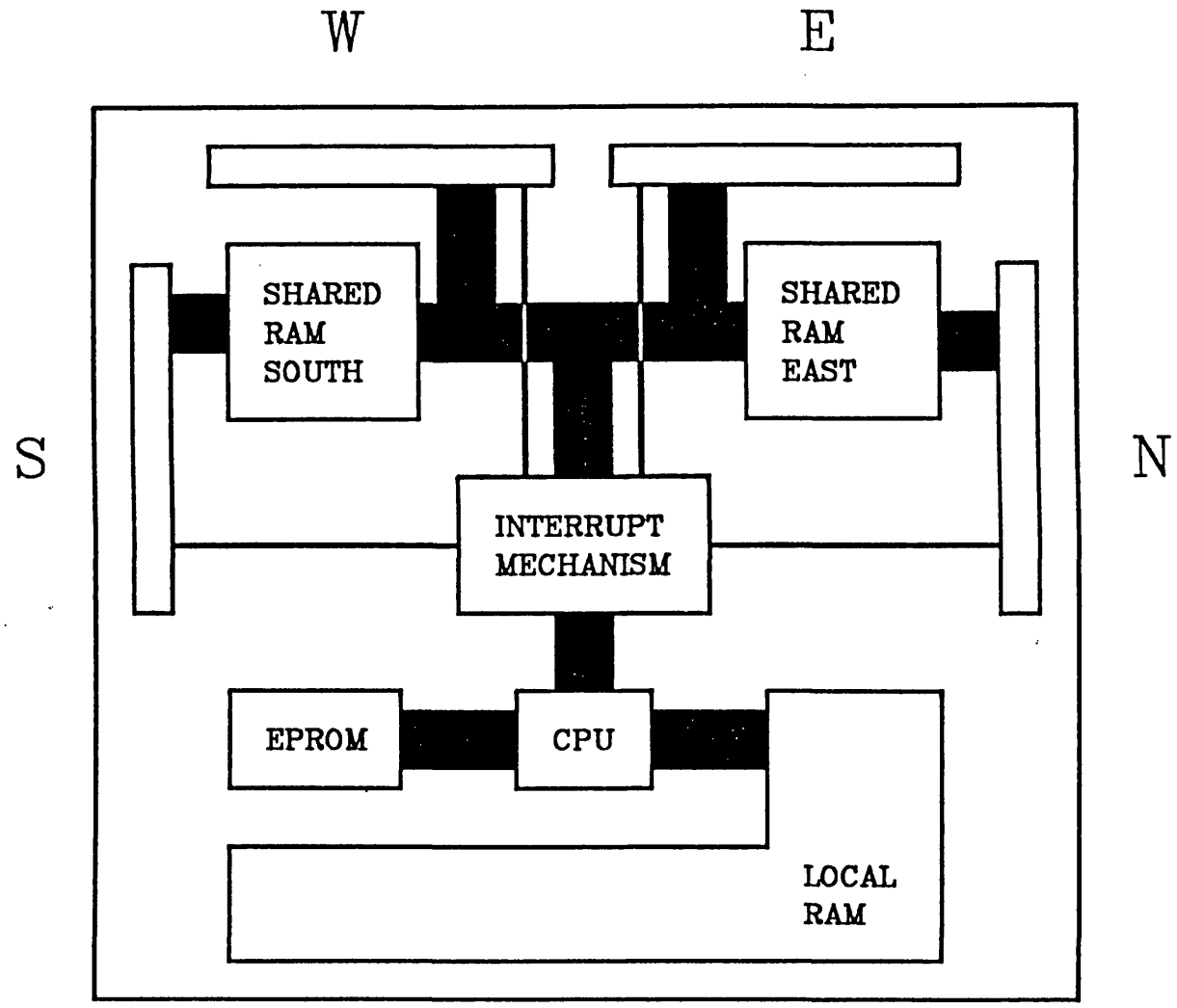
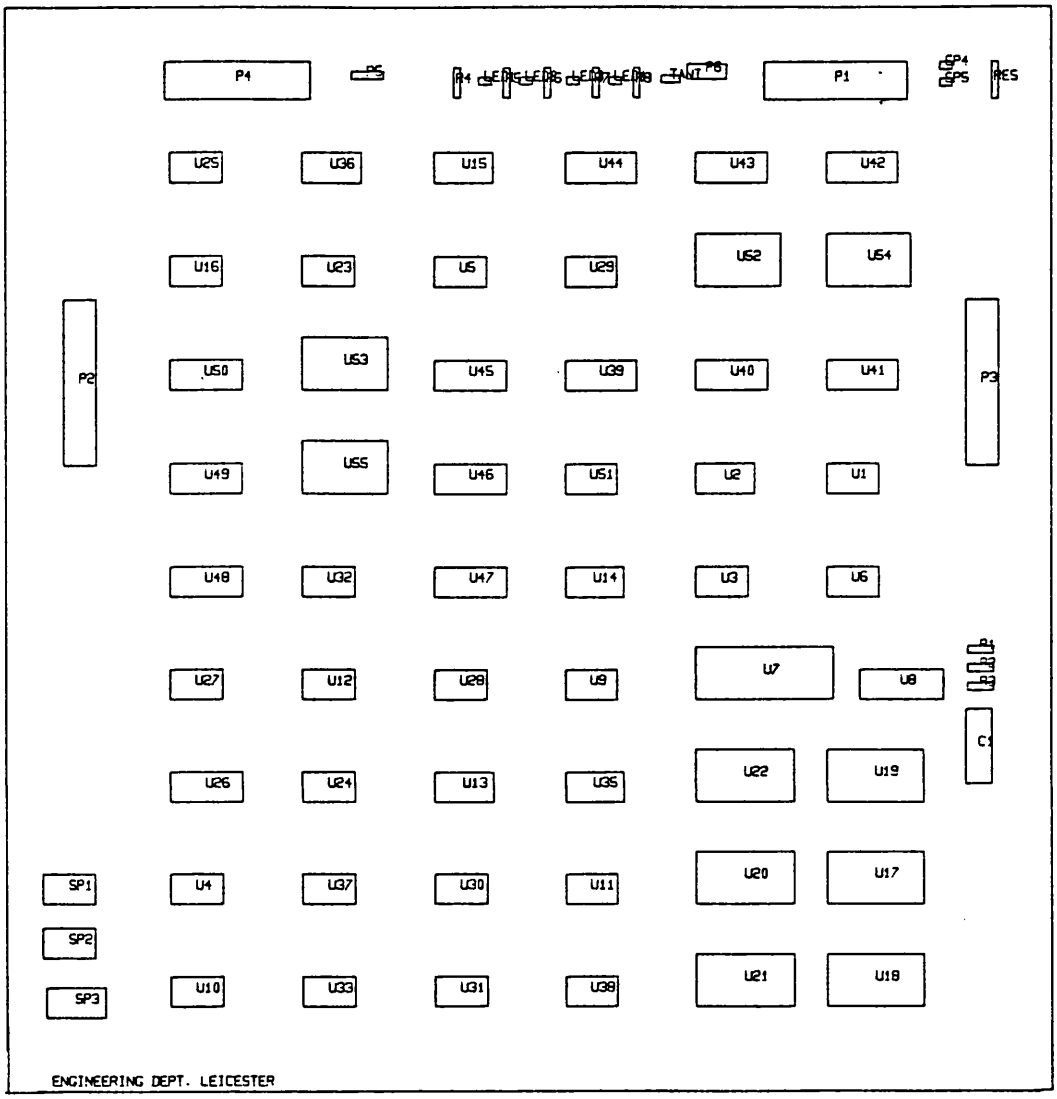


FIGURE 3.3 A PROCESSING ELEMENT



COMPONENTS PREFIXED WITH AN 'LED' ARE LIGHT EMITTING DIODES
COMPONENTS PREFIXED WITH A 'U' ARE INTEGRATED CIRCUITS
RES IS THE POSITION OF A POSSIBLE RESET SWITCH
COMPONENTS PREFIXED WITH A 'P' ARE CONNECTORS
COMPONENTS PREFIXED WITH A 'C' ARE CAPACITORS
COMPONENTS PREFIXED WITH A 'R' ARE RESISTORS
SP1-SP3 ARE SPARE DIL POSITIONS

FIGURE 3.4 POSITION OF COMPONENTS ON A PROCESSING ELEMENT

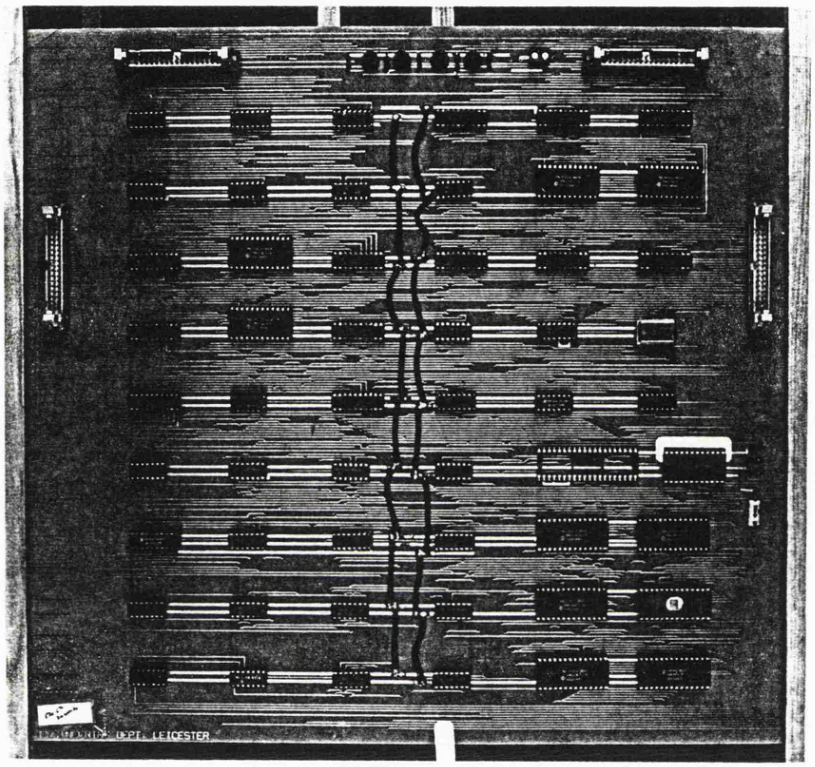


FIGURE 3.5 AN ACTUAL PROCESSING ELEMENT

possible, avoiding the size of the problem being dictated by either the amount of local memory or the amount of shared memory. The amount of shared memory in the system was increased from the minimum necessary to ensure that it is also capable of acting as the message passing mechanism, described in chapter 4.

The circuit diagrams which make up the circuitry of a PE can be seen in appendix B. The memory map of the PE is as follows:

$F000	ROM (containing the multiprocessor operating system)	4K
$D000	PROGRAM RAM 2	2x8K
$B000	PROGRAM RAM 1	
$A000	INTERRUPT MECHANISM	8 bytes
$9000	SHARED MEMORY (WEST)	4x4K
$8000	SHARED MEMORY (SOUTH)	
$7000	SHARED MEMORY (EAST)	
$6000	SHARED MEMORY (NORTH)	
$4000	DATA RAM 3	3x8K
$2000	DATA RAM 2	
$0000	DATA RAM 1	

The identifiable functions of each PE necessary for the implementation of the memory map and the functions provided at these memory mapped locations are described in the following sections. The functional blocks of the PE are listed here and described more fully below:

2.1 Processor and clock;

2.2 Local memory;

2.3 Decoding (chip select);

2.4 Interrupt mechanism;

2.5 Shared memory;

2.6 Contention unit;

2.7 Self monitoring function.

2.1 Processor and clock

The processor chosen for the multiprocessor system was the Motorola MC6809E. This was arguably the most powerful eight bit microprocessor on the market at the time of the system design; it provided all the functions necessary for the application to which it would be put and at a reasonable cost. Each PE generates its own clock, making the 16 PE array an asynchronous MIMD architecture. The processor requires two quadrature clocks to be fed to it, in the MC6809E these are termed the E and Q clocks. The E and Q clocks in the PE are derived from dividing down a high frequency clock in a counter. This counter (seen in appendix B) may be loaded with preset data, this provides the ability to freeze the clocks to the processor for short periods of time (Hitachi [47]). This is the means by which the contention units (for shared memory arbitration) provide the function for which they are designed (see section 2.6).

2.2 Local memory

There are two types of local memory resident on a PE pcb which can only be accessed by that CPU: Random Access Memory (RAM) and Read Only Memory (ROM). 4K bytes of local ROM are provided (from $F000-$FFFF) in which the multiprocessor operating system is held. 40K bytes of local RAM is provided, which is allocated to enable 24K bytes of data and 16K bytes of program. Data and program RAM are physically separated in this system to facilitate the sharing of data with other PE. Providing a definite area in which programs to be run can be loaded, makes it possible to share code with the multiprocessor operating system relatively easily. All the RAM in the system is static RAM: This reduces the number of microchips on the PE pcb since no refresh circuitry (for dynamic RAM) is needed; the static RAM is easier to use in a shared memory application; The operation of dynamic

RAM may have been affected by constant halting of clocks (due to the contention circuitry) and static RAM for the system was available relatively cheaply. The use of static RAM is not a feature of the method for sharing memory but in this particular design any possible causes of problems were eliminated if they were not fundamental to the design.

2.3 Decoding

Since the memory map is decoded into 4K byte segments the chip selects necessary to decode the memory map of the PE are provided by a four to sixteen line decoder, which may be seen in appendix B. The shared memory chip select signals are obtained by further decoding these signals with simple logic gates and an additional address line from the processor. This further decoding are necessary since the 4K bytes contained in each shared memory block is split into two 2K byte memories.

2.4 Interrupt mechanism

An essential feature of the architecture is the ability of the PEs to pass information of various kinds around the system efficiently. This is performed by packet switching which is described in chapter 4. The interrupt mechanism is an integral part of the message passing ability, hence interrupts to PE to the North, South, East and West of the PE are provided. The area of the PE memory map in which interrupts are located is further decoded as follows:

$A000 RESET INTERRUPT FROM NORTH
$A001 RESET INTERRUPT FROM EAST
$A002 RESET INTERRUPT FROM SOUTH
$A003 RESET INTERRUPT FROM WEST
$A004 INTERRUPT PE TO NORTH
$A005 INTERRUPT PE TO EAST
$A006 INTERRUPT PE TO SOUTH
$A007 INTERRUPT PE TO WEST

To interrupt a PE in a given direction a read or write is performed to address $A004 for North, $A005 for South etc.. This is decoded, and causes the output of a latch to be held in the low state. This low state is seen by the interrupt (IRQ) line of the PE in the required direction, causing an interrupt request. The IRQ line is reset to the normal high state when the interrupted PE reads or writes in a similar fashion to memory location $A000 for North, $A001 for South etc., in the direction of origin of the interrupt request.

2.5 Shared memory

The shared memory blocks are buffered to enable access by two PEs. Address buffers need only be uni-directional (ie providing access from one processor to the memory) while data buffers need to be bi-directional (ie providing access to/from the memory by a processor in order for a correct R/W to occur), this arrangement can be seen in appendix B. To ensure access by only one PE at a time the buffers to the memory are enabled using the outputs of an SR latch. The positive latch output enables one set of buffers and the negative output the others, this ensures that one enable signal is always the opposite of the other and thus access by one PE only. It is this latch, one for each of the shared memory blocks to the East and South of a PE, which defines the ownership of the memory as belonging to the PE whose address buffers are enabled.

2.6 Contention circuitry

The need for a contention circuit is evident, it provides the arbitration needed for the use of the memory since a shared resource such as shared memory may only be used by one processor at a time, if the data being

written or read to or from the memory is to remain uncorrupted. The function of the contention circuit is to resolve the critical problem of two processors simultaneously requesting the use of a shared resource and the problem of one PE requesting the memory while it is in use by another PE.

Several prototypes of contention circuit were considered, including one by Thomas [85] incorporating monostables. This was dropped in favour of a more efficient circuit in terms of the time taken to switch the memory, that of the contention circuitry of Warrington and Thomas [86]. In this design each processor which requires access to a particular shared memory has a Contention Unit (CU) associated with that resource. A general PE can then be seen to have four such units: North, South, East and West one of which can be seen in figure 3.6. The contention circuitry ensures ownership of the memory for a double byte read/write and even read/modify/write cycles, so that 16 bit integers can be passed without problems. The ACQUIRE signals from two CUs of the PEs requiring access to a particular piece of shared memory are fed into a simple SR latch. The output of this latch defines ownership of the shared memory. Only a PE which owns the memory may have access to it. If shared memory is owned by the processor which makes a subsequent access it must be emphasised that the contention circuitry is almost transparent to the request.

Using signals fed back from the memory ownership latch, the contention circuitry is able to deal with the following problems ar ising from the use of shared memory, discussed in more detail below:

a. Access of shared memory not in use by, but owned by, another PE;
b. Access of shared memory by a PE while it is already being accessed by another PE

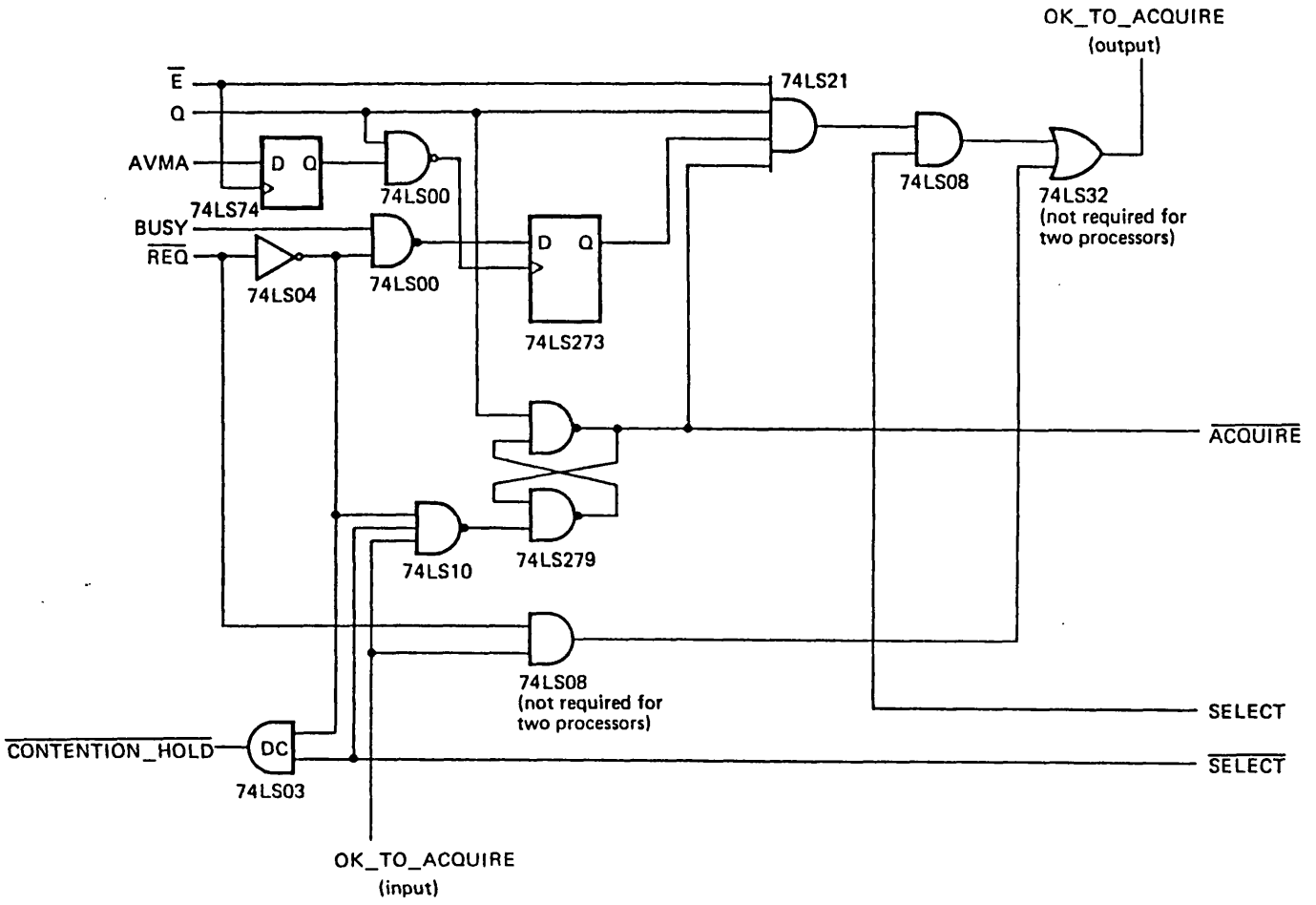


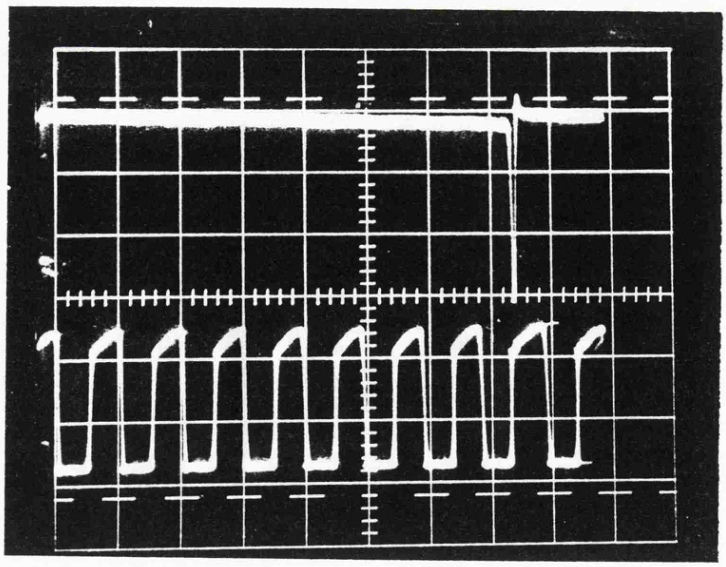
FIGURE 3.6 THE CONTENTION UNIT PROPOSED BY WARRINGTON AND THOMAS

c. Simultaneous access of the shared memory by two PE$.

2.6.1 Access of shared memory not in use by, but owned by, another PE;

In the case where a PE requests the use of shared memory which it does not own it is made to wait, using the CONTENTION_EXTEND line which halts the processor, until the latch dictating memory ownership is switched for use by the requesting PE. The ACQUIRE signal, which switches the memory ownership latch, is sent only when the PE which owns the memory at the time of the request is in the correct part of its clock cycle to release the memory, effectively synchronizing the two PEs when a memory switch occurs. In an asynchronous system, such as the 16 PE array, the synchronization between processors using the CU proposed by Warrington and Thomas fails because the CU allows synchronization to take place too near to the rising edge of the E clock of the PE which owns the memory when the memory is being requested by another PE. This leads to the memory being lost and regained within one processor cycle, and consequently at best a loss of data occurring and at worst an oscillation of CONTENTION_EXTEND signals between PEs leading to a crash of one or both of these PEs. The problem can be solved by the modification proposed by Holme and Warrington [49], which simply shifts the point at which synchronization takes place away from the rising edge of E. The PE pcb was designed using the CU proposed by Warrington and Thomas, with the modification proposed by Holme and Warrington implemented on the PE pcb as can be seen in the circuit diagrams of appendix B.

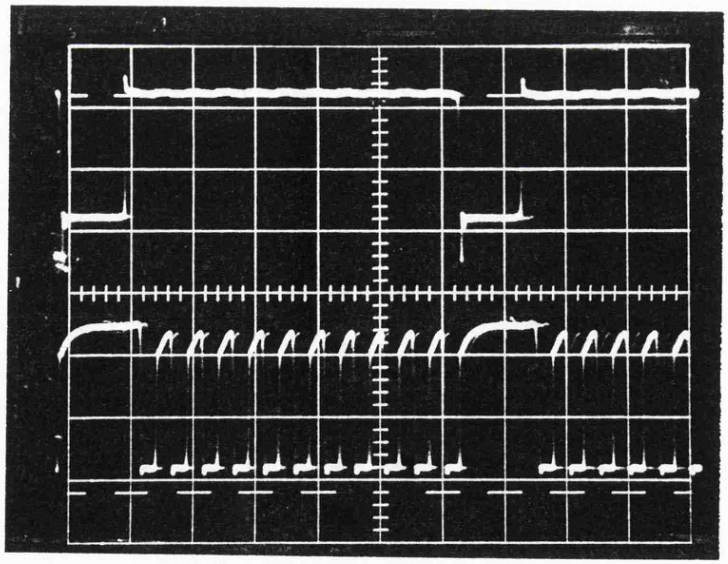
Figure 3.7 shows the effect of the contention unit in this case, where memory access is delayed while memory ownership switches[3]. The time taken for the switching can be seen as a glitch on the CONTENTION_EXTEND signal



top trace : CONTENTION_EXTEND signal
bottom trace: Q clock

oscilloscope settings: 2 V/division (both traces)
2 μs/division (time base)

FIGURE 3.7 SHARED MEMORY SWITCHING



top trace : CONTENTION_EXTEND signal
bottom trace: Q clock

oscilloscope settings: 2 V/division (both traces)
1 μs/division (time base)

FIGURE 3.8 SHARED MEMORY CONTENTION

which extends the clock cycle (only the Q clock is shown) by the switching time.

2.6.2 Access of shared memory by a PE while it is already being accessed by another PE

In this case the CU halts the PE trying to use the memory while it is used by another PE by taking CONTENTION_EXTEND low until the memory is relinquished by the PE which owns the memory and switching, as described above, can then take place. This can be seen in figure 3.8, where the CONTENTION_EXTEND line can be seen to hold the clock for two normal clock cycles corresponding to an integer write by the neighbouring PE with access to the shared memory. The PE clocks are halted for the duration of the CONTENTION_EXTEND signal and the subsequent switching time of the memory, the Q clock of which can be seen in the figure.

2.6.3 Simultaneous access of the shared memory by two PE

In the case of a simultaneous access the CU resolves the situation by granting access to the PE which currently owns the memory.

2.7 Self monitor function

Each PE board has circuitry which decodes the fetch of an interrupt vector and the reset of an interrupt, which can be seen in appendix B. Four signals are provided and fed to the small edge connector between shared memory ports. The fetch of an interrupt vector corresponds to global

---

[3] The photograph was obtained by running a program in each of the PEs having access to a shared memory as fast as possible.

communication of some kind taking place. In the early stages it was hoped that these signals could be fed directly into a digital voltmeter so that readings of the percentage of time spent in global communication could be dynamically obtained. This was shown to be possible in an early 2 PE system, but due to the reset of an interrupt signal in a PE of the multiprocessor having to be performed as quickly as possible, so that other interrupts are not missed, would mean that the signals resulting from the status monitor circuitry would not be representative of the communication within the PE. For this reason these signals are effectively redundant and are not used in the 16 PE prototype system.

3.0 THE INTERFACE BOARD

Figure 3.1 shows the Master Processor (MP), which, as has been stated, is interfaced to one edge of the two-dimensional array of PEs, using the shared memory connections already present on the board in the East direction (with respect to a PE). The MP provides the interface from the PE array to the outside world. It provides an environment where programs for the array can be developed, a means to run them on the array, and then to retrieve any results (which may be stored on disk for post processing if necessary). Other possible connections of the MP to the PE array were possible but this method was viewed as the best compromise when considering the complexity of the connections and the ability to oversee the functions of the PE array. The MICROBOX II was chosen as the MP for the multiprocessor array as it is based on the 6809E microprocessor, which would make it easy to interface to the 16 PE array, and was relatively cheap. For the Master Processor to communicate with the required amount of shared memory in each of the PEs to which it would communicate, using the interrupt driven message passing mechanism described in the following chapter, the interface board needed to provide the following functions:

(a) access to 16K bytes of shared memory (4K in each of the PEs to which it was connected

(b) an interrupt mechanism, enabling interrupts to pass in either direction between the master processor and the PEs to which it would communicate.

3.1 Realisation and implementation of the functions of the interface board

The MICROBOX II manual (Reference [63]) shows that the user expansion port of the MICROBOX II has two signals IO1 and IO2, decoded as follows:

$FF20-$FF3F (inc.) IO1

$FF40-$FF5F (inc.) IO2 .

Each shared memory port of the PE has 4K Bytes of memory associated with it. The MP is interfaced to 4 PEs along one side of the PE array and therefore needs to be able to access 16K Bytes of memory. The MICROBOX II provides 5 address lines at the user expansion port, this, coupled with the decoded signals IO1 and IO2, would allow access to only 64 Bytes of RAM. The interface board was then constructed to enable the MP to access the required amount of memory. The circuit diagrams for this can be seen in appendix C, which also lists the components, the components position on the board and the pin outs of the connectors.

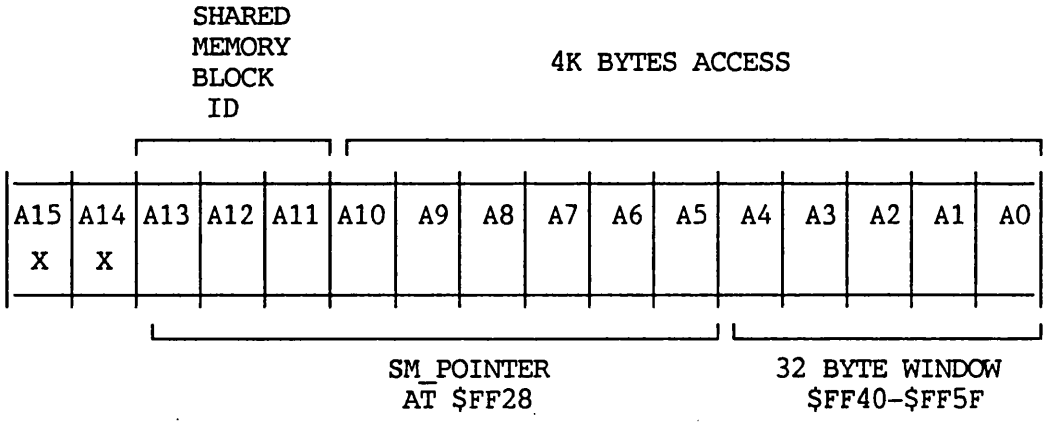
The 16K bytes of shared memory which the interface board makes available to the MICROBOX II (4K in each of the PEs to which it is interfaced) is accessed through a 32 byte wide window. To access the 16K bytes of contiguous memory requires address lines A0-A13 (inc.). Address lines A11,

A12 and A13 are used (as indicated in figure 3.9) to identify the 2K byte block of shared memory block in which a request will be made. The shared memory has the address lines A5-A10 (inc.) provided by latches at $FF28, into which the base address of the 32 byte window can be written (using the IO1 decode) from the software of the MP. Locations within the 32 byt- window are accessed through locations $FF40-$FF5F (inc.), which provide address lines A0-A4 (inc.), using the IO2 decode. In the multiprocessor software the integer variable SM_POINTER is defined in memory to be at $FF28, which makes the setting of the latches a relatively easy task.

Again an interrupt mechanism is needed for packet switching to take place. The sending and resetting of interrupts from the MP is achieved by using the decoded signal IO1 to activate a 3-to-8 line decoder the outputs of which are fed into SR latches, as can be seen in appendix C. The signal is decoded as follows which enables any of the four PE$to which the MICROBOX II is interfaced to be interrupted, or an interrupt from any of them to be reset:

```
$FF20 .......... Reset IRQ to port 1
$FF21 .......... Reset IRQ to port 2
$FF22 .......... Reset IRQ to port 3
$FF23 .......... Reset IRQ to port 4
$FF24 .......... IRQ to port 1
$FF25 .......... IRQ to port 2
$FF26 .......... IRQ to port 3
$FF27 .......... IRQ to port 4
$FF28
  "             SM_POINTER
$FF29

$FF40
  "             32 bytes of SM pointed to
  "             by SM_POINTER
$FF5F
```



X = value can be set to anything

FIGURE 3.9 ADDRESSING THE LATCHES OF THE INTERFACE BOARD

4.0 THE STATUS MONITOR BOARD

The status monitor board consists of a number of I.C.s and a 4 x 4 array of tri colour Light Emmitting Diodes (LEDs), and has proved invaluable in the debugging and maintenance of the 16 PE array. Each LED corresponds to a PE within the array, the colour of the LED reflecting the status of the PE (seen in table 3.1).

colour	status of PE
yellow	normal running
red	in a synchronize state or CRASHED
green	servicing an interrupt (ie active in global communication)

TABLE 3.1 LED status indicator

The signals which make the decode of the PE status possible are the Bus Available (BA) and Bus Strobe (BS) pins of the MC6809E. The way in which the processor status may be derived from these signals can be seen in the Hitachi microprocessor data book [47]. In order that the signals arising from global communication can be made visible on the LEDs they are stretched by monostables on the board to approximately 0.5s duration. The circuit diagrams for the status monitor board can be found in appendix D, which also lists the components, the position of the components on the board and connector details.

CHAPTER 4

CHAPTER 4

THE SOFTWARE

1.0 INTRODUCTION

The 16PE machine has been designed and built for the purpose of solving field equations, as described in chapters 2 and 3. The chapter first describes the programming language (PL/9) in which all of the multiprocessor software is written. The choice of the bench mark algorithm for the tests necessary to display the speedup of the system and the means by which this algorithm is coded for use on the multiprocessor is then described. Knowledge of how the bench mark software implements the solution algorithm of the problem, should then help in the understanding of the system software, which implements the input, output and running of a problem on the multiprocessor hardware. The system software consists of the multiprocessor operating system which runs in each of the PEs of the multiprocessor, and an overseer program running on the Master Processor, both of which are detailed at the end of the chapter.

2.0 THE PROGRAMMING LANGUAGE (PL/9)

PL/9 has been specifically designed for 'low level' control applications using the Motorola MC6809 microprocessor and as a result it takes full advantage of the architecture of this powerful processor; as no trade offs have been made to make the 'core' of PL/9 programs compatible with other processors. Library routines of all required functions such as input/output, floating point arithmetic and number conversion routines, are available and can be INCLUDED in a PL/9 program at any time.

PL/9 is a procedural language which makes use of BYTE, INTEGER and REAL variables, each of which may be defined globally (visible to all procedures) or locally (visible only within a procedure) as required by a program. Variables can also be defined to be AT a specific memory location, making writing to memory mapped peripherals relatively easy. Memory locations may also be defined as 'read only', enabling constants to be used anywhere in the program and accessed by a meaningful variable name. More detailed information about all aspects of the language may be found in Windrush [92], Windrush [93] and Windrush [94].

The interactive development system offered by the co-resident PL/9 editor/compiler/tracer runs under the FLEX disc operating system which offers a solid base for developing programs. The development system (a MICROBOX II, a MC6809E based microcomputer) has all the facilities needed to whilst the PL/9 the language which is ideally suited to the programming of the multiprocessor system.

3.0 BENCH MARK SOFTWARE FOR SPEEDUP FIGURES AND COMPARISON WITH OTHER MACHINES

3.1 Laplace's equation

One of the simplest field equations is Laplace's equation:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \quad ,$$

where ϕ is some property of a field in a two-dimensional xy plane. For a field of interest (or mesh) consisting of p x p nodes, the solution of

Laplace's equation by finite differences involves placing known boundary conditions in the appropriate nodes of the mesh (usually at the mesh boundaries) to define the area of interest. If finite difference approximations to the above equation are then made for the field property ϕ at the nodes of the mesh, the nodes of which may be referenced by an i and j vector, then Laplace's equation becomes:

$$\frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{\Delta x} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y} = 0 \quad ;$$

if $\Delta x = \Delta y$ = a constant throughout the mesh, then the property ϕ at reference node (i,j) can be expressed such that:

$$\phi_{i,j} = 0.25 \times (\phi_{i-1,j} + \phi_{i,j+1} + \phi_{i+1,j} + \phi_{i,j-1}) \quad .$$

With the boundary values of the mesh already prescribed, Laplace's equation can thus be solved by repeated passes over the mesh continually updating the nodes within that mesh with the average value of the nodes around it, where the new value of ϕ, that of ϕ^n, is given by:

$$\phi^n_{i,j} = 0.25 \times (\phi^n_{i-1,j} + \phi^{n-1}_{i,j+1} + \phi^{n-1}_{i+1,j} + \phi^n_{i,j-1}) \quad .$$

This iterative method of solution can continue until the error at each node reaches an acceptable value. Faster methods of obtaining convergence do exist, but for the purposes of a bench mark this method is adequate. The bench mark Laplace problem consists of a square mesh problem with prescribed boundary conditions (as can be seen in figure 4.1). The physical problem may be viewed as two dimensional heat conduction where the field quantity ϕ can be viewed as the temperature across a square metal sheet, whose initial temperature is 1.1° C which then has a constant heat source

```
B B B B B B B B B B B B B B B B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B A A A A A A A A A A A A A A B
B B B B B B B B B B B B B B B B
```

A TYPICAL SQUARE MESH PROBLEM.
IN THE CASE OF THE BENCH MARK PROBLEM THE BOUNDARY NODES (B) OF THE PROBLEM ARE INITIALISED TO THE VALUE 9, AND THE MID POINT NODES (A) ARE INITIALISED TO THE VALUE 1.1 .

FIGURE 4.1 THE SQUARE MESH PROBLEM

of 9° C placed at its boundaries. Thus in the bench mark problem all the boundaries of the problem are set to the value 9 and the internal nodes to 1.1. The solution of the Laplace problem will thus result in the value 9 in all nodes of the problem mesh. This seemingly trivial problem is nevertheless representative of the way field equations are solved by finite differences, and it is for this reason the problem has been chosen as the bench mark for the tests on the PE array. The choice of the value to which the internal nodes are initialized is arbitrary, but here it is chosen to give appreciable run times for the multiprocessor system and is consistently set to this value in all the programs of this chapter, to enable comparisons between the performance of different systems to be made.

The Laplace solving algorithm can be seen coded in FORTRAN for PRIME and VAX systems in program "LAPLACE" (microfiche listing 2); in FORTRAN for an OLIVETTI M24 in program listing 1 and in PL/9 for a MICROBOX II in listing 2. Where possible the coding of the problem makes use of the intrinsic ability of the host machine to handle two-dimensional arrays, with all data storage taken care of by the compiler. PL/9 does not support multi-dimensional arrays, these are implemented on the MICROBOX II using the method suggested in WINDRUSH [92], [93] and [94].

3.3 Implementation of a Laplace solver on the multiprocessor

The solution of the Laplace problem on the multiprocessor requires a two-dimensional array to be stored in different and non-contiguous areas of memory. Most nodes within a problem mesh are stored in local memory, but because each processing element (PE) shares the nodes at a PE boundary with a nearest neighbour, such nodes must be stored in shared memory. As was discussed in chapter 2 a special case of storage exists where a "corner node" needs to be stored in three different areas of memory, any

implementation of a Laplace solving algorithm on the multiprocessor must take this into account.

If the number of nodes in a problem was always constant then it would be possible to construct an architecture in which the hardware would map the two-dimensional array directly onto the appropriate piece of memory. Since the architecture of this multiprocessor system has been designed to solve a range of problem sizes, some other means of mapping the two-dimensional data onto the correct piece of memory must be found.

In the present case a procedure, called the 'get_address' procedure, was written. Before any access of data from the problem mesh, the i and j vectors of the node are fed to the 'get_address' procedure which returns the address at which the data corresponding to these vectors can be found. The procedure has been written for a general case and will accept a k component for three dimensional problems, for two-dimensional operation k should be zero. The Laplace program itself may be expanded to three-dimensions quite easily, since worries about the storage areas of the data have been dealt with. This method of mapping an abstract data type (such as the problem mesh) onto physical memory does involve a significant (but necessary) software overhead, the effects of which may be seen in the results of the speedup tests performed by the multiprocessor.

Earlier it was stated that shared memory was the optimum architecture for the solution of field equations, since data was made available in the most up to date form to a requesting PE, in the shortest possible time. The use of a get_address procedure does introduce an overhead in the fetch of addresses, but this fetch would be necessary in all the systems which were considered for the implementation of field equation solutions due to the constraints of the PL/9 language and the many different problem mesh sizes

the PE array can deal with. The provision of up to date data could not be guaranteed in any of the other systems which did not use shared memory, which were considered at the design stage. One such system included DMA techniques and the use of 'local' memory only within a PE. Suggestions for reducing the overhead of the get_address procedure, and thus improving the performance of the system in relation to competition from other architectures and systems, are made in the discussion chapter of this thesis.

The resulting Laplace program for use on the multiprocessor can be seen in program listing 3. The program is general in that, even though the Laplace problem may be required to run in 1, 4, 9 or 16 PEs, in each of the PEs required for the problem it is the same program which runs. This is made possible by information sent at run time by the master processor (in a fashion described in a later section), to initialize the problem within each PE. The information relates the PE to the relative position of the PE within the solution mesh, the size of the mesh problem and the value to which boundary nodes should be initialized. This data is transferred from shared memory to local variables during initialization of the array, to avoid corruption of the values which would occur during the solution of the Laplace problem since this data and that of some shared nodes of the problem would share the same memory. The actual program sequence run on any PE will thus be different, although the total program is identical on all of the PEs. Other than these sections of code, and the use of a get_address procedure, the function of the program is much the same as for the other Laplace solvers detailed so far. The one notable exception is the inclusion of a 'copy' procedure, which is used to copy the "corner node" data held in local memory to the two areas appropriate areas of shared memory, after a corner node has been updated to enable nearest neighbour access of the corner nodes to take place.

3.4 Programming points

In an asynchronous multiprocessor system such as this it is important to avoid the use of uninitialised data. This can occur when running a program in a number PEs which effectively initialises values in shared memory for subsequent access by a neighbouring PE; there must be two phases of the solution, firstly the boundary value initialization and secondly the repeated iteration until some criterion is met. If one PE completes its initialisation phase and then proceeds to the iteration phase, it is possible that invalid data can be read if a neighbouring PE has not yet finished its initialisation. To avoid this problem each PE must be synchronized by some means with its neighbours. In the case of the bench mark Laplace solver, after each of the PEs has finished its initialisation it is made to enter a "synchronize" state which halts the execution of the program. Only after all the PEs have entered this state can they be told to continue, a function provided by the overseer program.

Field equation solving programs to be run on the multiprocessor, including the Laplace program, should be written so that program execution starts at location $B000 in a PEs memory. This location corresponds to the start of program memory in a PE and is the default start location of a field equation solving program used by the multiprocessor operating system. The get_address procedure should begin at $C000, to ensure the same procedure can be used by the multiprocessor operating system to extract the solution from the mesh.

3.5 Laplace programs used on the multiprocessor system

The bench mark software outlined above was used with some variations to obtain the results of the tests in chapter 5. The Laplace program of listing 3 is used in the initial processing array tests, thereafter a faster version of the get_address procedure is used (this is listed as program 4). The problem convergence tests of chapter 5 are carried out by the program listed as program 5; the INCLUDED files are also listed immediately following the main program.

4.0 THE SYSTEM SOFTWARE

The system software consists of two major parts; the multiprocessor operating system and the master processor overseer program. The means of operation and the functions provided by this software are reliant on the packet switching technique which the system uses to pass messages between the PEs. A description of the packet switching technique is given in section 4.1 and the means by which it is implemented successfully is described in section 4.2, before the system software is described.

4.1 Packet switching within the multiprocessor system

Packet switching is a technique whereby information in the form of packets is 'switched' through a system. These packets comprise not only of a message but also data associated with the message, giving the source and destination of the message within the packet, enabling it to be switched either by hardware or software through the system to the correct destination.

Packet switching in the multiprocessor is used in the global transfer of information, each packet is switched through the system by means of pigeon holes in shared memory and the use of interrupts. All global communication passes along East-West rows through the PE array, to or from the master processor. The pigeon holes in shared memory, which make up a packet of information contain the source of information, destination of the information, and a message. If the packet has data associated with it other pigeon holes will indicate how much data there is, and where it is located. Because the packets are switched using software by means of interrupts, it is also necessary to include a software INTERRUPT IDentifier flag for the reason made clear below. The pigeon hole locations are reserved at the top of the associated shared memory block (East and West), such that any locations that a field solving program may use will not be corrupted.

In order to switch packets of information through the multiprocessor by means of interrupts, the packets are first loaded into the appropriate shared memory pigeon holes. An interrupt request is then sent to the nearest neighbour in the direction of the destination PE. The interrupted PE then checks the pigeon holes in the East and West shared memories for a TRUE software INTERRUPT IDentifier flag, since the hardware interrupt could have originated from either direction[1]. The packet of information is then read, a reset of the hardware interrupt is performed and the software INTERRUPT IDentifier flag is then reset to FALSE. If the packet was not destined for the PE which has just processed the packet, the packet (and any associated data) is copied to the appropriate shared memory locations

---

[1] The MC6809E does not support multiple vectored interrupts which, if available, could simplify this function.

and passed, in a similar fashion, further down the same row of the processing array until the correct destination is reached.

A packet of data has associated with it the following pigeon hole locations:

(a) An **INTERRUPT_ID** byte;

(b) A message **FROM** identifier;

(c) A message **TO** identifier;

(d) The **MESSAGE** byte;

(e) A **DATA_POINTER** to associated data memory;

(f) A **DATA_SIZE** pointer.

Each of these is discussed briefly in the following:

(a) The **INTERRUPT_ID** byte contains either TRUE OR FALSE to enable a PE to identify from the software of the multiprocessor operating system the direction from which the interrupt driven packet has originated.

(b) The **FROM** byte contains the unique number of the PE (held in the ROM containing the multiprocessor operating system within the sending PE) identifying the source from which the message has originated.

(c) The **TO** byte contains the unique number of the PE to which the message is destined.

(d) The **MESSAGE** byte contains one of the following codes, describing the type of packet:

message	corresponding hex code
MASTER	$00
PROGRAM_TO_LOAD	$01
PROGRAM_LOADED_OK	$02
RUN_PROGRAM	$03
PROGRAM_RUNNING_OK	$04
STOP_PROGRAM	$05
PROGRAM_STOPPED_OK	$06
GOT_SECTION_OF_PROGRAM_OK	$07
PROGRAM_LOAD_ERROR	$08
UNEXPECTED_INTERRUPT	$09
TRANSMISSION_ERROR_E2W	$0A
TRANSMISSION_ERROR_W2E	$0B
DUMP_DATA	$0C
DUMPING	$0D
HALT_PROCESSOR	$0E
PROCESSOR_HALTED	$0F
CONTINUED_RUN	$10
CARRY_ON	$11
RUN_LAPLACE	$12
WRONG_READ	$14
NULL	$00

It is these message bytes which are picked up by the interrupt routine (described in section 4.3.1) which initiate all the action taken by the multiprocessor.

(e) The **DATA_POINTER** (a word which) contains a pointer to where in memory any associated data is to be found.

(f) The **SIZE_OF_DATA** (a word which) contains the number of BYTEs of data which can be found at location pointed to by the contents of DATA_POINTER.

4.2 The mapping of data packets in pigeon holes in shared memory

In order to achieve bi-directional data transmission along a given East-West row of the PE array, without the destruction of information taking place as messages pass through a PE, a different set of pigeon holes is used for each direction of data transfer. To uniquely identify the pigeon holes for each direction of data transfer in each of the shared memory blocks, the pigeon hole locations are annotated with the direction in which the data will travel, or has travelled, to or from the PE. The annotated pigeon holes in the directions West and East are thus as follows:

```
WEST SM BLOCK

$9FFF byte      INTERRUPT_ID_TO_WEST
$9FFE byte              FROM_TO_WEST
$9FFD byte                TO_TO_WEST
$9FFC byte           MESSAGE_TO_WEST
$9FFA integer   DATA_POINTER_TO_WEST
$9FF8 integer   SIZE_OF_DATA_TO_WEST
$9FF7 byte    INTERRUPT_ID_FROM_WEST
$9FF6 byte            FROM_FROM_WEST
$9FF5 byte              TO_FROM_WEST
$9FF4 byte         MESSAGE_FROM_WEST
$9FF2 integer DATA_POINTER_FROM_WEST
$9FF0 integer SIZE_OF_DATA_FROM_WEST

EAST SM BLOCK

$7FFF byte    INTERRUPT_ID_FROM_EAST
$7FFE byte            FROM_FROM_EAST
$7FFD byte              TO_FROM_EAST
$7FFC byte         MESSAGE_FROM_EAST
$7FFA integer DATA_POINTER_FROM_EAST
$7FF8 integer SIZE_OF_DATA_FROM_EAST
$7FF7 byte      INTERRUPT_ID_TO_EAST
$7FF6 byte              FROM_TO_EAST
$7FF5 byte                TO_TO_EAST
$7FF4 byte           MESSAGE_TO_EAST
$7FF2 integer   DATA_POINTER_TO_EAST
$7FF0 integer   SIZE_OF_DATA_TO_EAST
```

It can be seen that output to the West of one PE is the input from the East of another PE. When programming the system care must be taken to get the

direction of information packets correct, or the data will be lost in the system.

4.3 The synchronize instruction

Within the multiprocessor system all messages which are packet switched through the system are acknowledged. The protocol of message passing in the multiprocessor system is simplified by restricting global communication in the PE array to one PE at a time, not only to simplify the message protocol but to ensure the integrity of messages being sent and received in the multiprocessor system. Global communication is restricted to one PE at a time by means of the synchronize instruction of the MC6809E microprocessor.

The state of interrupts (enabled or disabled) determines the action taken by the processor when the synchronize instruction is terminated by an interrupt signal. Regardless of the state of interrupts when the synchronize instruction is executed, the processor is halted until an interrupt occurs - unless an interrupt request is already present. On receipt of an interrupt, with interrupts enabled, the interrupt request is serviced, thereafter terminating the halted state and continuing program execution. If the interrupt occurs with interrupts disabled then the interrupt is not serviced, but the halted state of the processor is terminated and program execution is continued. In the system software, listings of which are referenced later, two applications of the synchronize instruction can be seen, both with previously disabled interrupts:

In the first application, the synchronize instruction is immediately followed by an enabling of interrupts. This enables an acknowledge packet of a previously sent message to be serviced without hanging any of the PEs

in the system. It can be seen that if the interrupt associated with the acknowledging packet was allowed to be serviced before the execution of the synchronize instruction (designed to wait for this acknowledging interrupt), communication with the PE would hang, causing the system to crash.

In the other application of the synchronize instruction, the incoming interrupt is used purely as a synchronizing signal to restart the execution of the program, and has no message implication. The interrupt request is not serviced and thus not reset by the interrupt service routine, a consquence of which it is reset immediately after the synchronize instruction, to enable future such events to occur.

4.4 The multiprocessor operating system

The functions of the multiprocessor operating system are initiated by the receipt of interrupt driven packets of information, originating in the master processor. This section describes the interrupt procedure around which the functions of the PE array are based, and then describes the functions offered by the operating system. The program listing of the multiprocessor operating system can be seen as program listing 6.

4.4.1 The interrupt procedure

All processing within the multiprocessor array is interrupt driven from packets of information originating in the master processor, and relayed through the system by component PEs. The interrupt procedure of the multiprocessor operating system is the point from which the functions provided by the system are initiated.

The interrupt procedure of the multiprocessor operating system is structured as a hierarchical tree, constructed using conditional statements. The procedure can be seen as part of the multiprocessor operating system, listed as program 6. On receipt of an interrupt the pigeon holes of the shared memory blocks are interrogated to find the direction from which the packet came, the ultimate destination of the packet and, if necessary, the message the packet contains. Knowing the direction from which the packet came enables a hardware reset of the latch causing the interrupt to take place, and a reset in the pigeon holes of the INTERRUPT_ID flag to indicate that direction is no longer interrupting the PE. If the destination of the packet does not match the PEs unique identification number the packet can be passed on, in the direction of the destination PE, with any associated data. If the destination of the packet has been reached then the message is decoded and one of the four basic functions of the multiprocessor operating system are performed; these are:

(a) To **LOAD** a program;
(b) To **RUN** a program;
(c) To **HALT** program execution in the PE array;
(d) To **DUMP** the results of a problem.

4.4.2 Functions of the multiprocessor operating system

(a) Load a program

Programs in Motorola hex format (see Windrush [92]) are passed as blocks of information, each of which is passed as the data associated with a message packet, through the PEs in the lower bytes of the shared memory. At the destination PE the information is decoded and loaded, a block at a time from shared memory, to the address within the PE indicated by the incoming coded program. This process continues until an end of transmission block is

detected, at which point the PE waits for the next interrupt driven packet. All packets of data entering the PE are acknowledged, with any errors being reported to the master processor.

(b) Run a program

The message packet initiating the running of a program can be one of two types (detailed in the overseer program a description of which follows), it can either execute a field equation instruction or simply run a program. In the case of a field equation run the information in the lower bytes of shared memory is the initial data required by the field equation solver (i.e. the size of the problem mesh and the boundary conditions). The program start location is assumed to be the location $B000 (the base location of the PEs program memory). In the case of a simple run it is the program start location which is passed through the shared memory, this enables a program to be run anywhere in PE memory.

(c) Halt a program

This message causes a global_halt_flag to be set to logical TRUE. At the end of the interrupt procedure this flag is tested and, if true, a synchronize instruction is executed. This stops the program returning to the main program and continuing execution of the problem, until an interrupt message is received which will reset the global_halt_flag. The instruction which resumes program execution is the continue option issued in the master processor. The halt instruction causes the state of the solution to be frozen, enabling the dump instruction to extract the solution at that time.

(d) Dump the results of a program

This instruction causes the PE (already in the halted state) to dump the state of the solution as it stands to the master processor. Data from the mesh is sent in blocks, each being the data associated with a message packet to the master processor. The number of blocks which are sent depends on the size of the problem mesh. The continue instruction issued in the master processor enables program execution to be continued in the PEs from the halted state.

Dumping of the data is made possible by the multiprocessor operating system having access to the get_address procedure of the solution program. The multiprocessor operating system has access to the size of the problem mesh by means of a global variable, set during the initialization, which takes place when a field equation is run. It is therefore possible for the multiprocessor operating system to provide a correct i and j vector for a call to the get_address procedure within the PE, and for the data to be extracted from the mesh. To enable the multiprocessor operating system to access the solution program's get_address procedure, the operating system is compiled with a ghost get_address procedure at the same location as the get_address procedure of the field equation solving program (in this case $C000).

4.5 THE OVERSEER

4.5.1 Introduction

The overseer program is a user-driven controller for the functions that the multiprocessor operating system can provide, and can be seen as program

listing 7. The overseer program provides the utilities needed to load, run and retrieve both results and status messages from the array of processing elements. The user currently has to issue these commands, from a position where proceedings within the array can be monitored. It is envisaged that any further updates of this program will have less interaction with the user as a result of greater overseer program control.

The user port of the MICROBOX II allows access to 32 bytes of memory. The addition of the interface board enables the MICROBOX II to access 16K bytes of memory, 4K bytes in each of the four PEs to which it is connected. The means by which this is achieved has been described in the previous chapter, with the shared memory of the PE array accessed by the MICROBOX II by setting a 32 byte window over the available 16K bytes of memory. The window in the shared memory is set by writing the base address of the required piece of memory to the variable SM_POINTER at $FF28, this enables 32 bytes to be accessed in the area $FF40-$FF5F. In the overseer program these offsets are made available through the use of a pointer variable (.sm). Details of the use and opreration of such variables can be found by reference to Windrush [92], [93] and [94].

The overseer program displays paged options which will load:

(a) A field equation solving program to a user requested number of PEs (1,4,9 or 16);

(b) A field equation solving program to a user specified, PE;

(c) Any program to any PE.

In order to run a program it is possible to choose options which will:

(a) Execute a field solving program, whose start location for execution is $B000, on the default architecture set by the load option 'L' described in detail below.

(b) Run a program in any PE with any start location.

Only one option for extracting the solution from the mesh exists, that of the DUMP instruction (detailed below). This extracts all the data from the local mesh of a PE and passes it to the master processor in blocks.

The screen of the master processor V.D.U. is split into two sections when the overseer program is run. The bottom of the screen provides paged menus from which the various options may be selected. The top of the screen is sectioned such that a one line status indication message can be displayed for each of the component PEs of the 16PE array. In all options which cause action to be taken in the PE array the relevant status message is displayed on the V.D.U. of the master processor, unless this option itself has been switched off. Similarly all incoming messages, be they data, acknowledge or error message packets, are displayed for the appropriate action of the user.

4.5.2 Available options of the overseer

<u>TEST</u> (option T)

This option sends a quick test signal to all 16PEs which is acknowledged. The signal which is sent is the same as the last two bytes of a program load

message and tests the ability of the PE to talk with the Master Processor using the software packet switching.

LOAD (option L)

This command prompts the user for a name of an already compiled program the user wishes to run on one of the architecture types (1PE, 4PE, 9PE or 16PE) the system currently offers. Input is required as to the number of PEs on which the program will run. The appropriate (compiled) file is then converted into Motorola hex format and transferred a line at a time through the system with each line occupying the memory locations at the base of the appropriate shared memory. Transferring a line of Motorola hex coded program, which consists of a maximum of 32 bytes, can be achieved without alteration of the shared memory pointer latches.

EXECUTE LAPLACE (option E)

The user is shown a small square grid of size nxn points with named boundaries. The user is prompted for the values of the boundaries and of 'n'. The overseer then sends this data through the lower bytes of East-West shared memory, to the default number of processors (set when using the L option).

HALT (option H)

Issuing a halt instruction causes a halt in the execution of the program executing on the default number of PE (set using the 'L' option). This enables a snapshot of the results at a given time to be obtained, when the dump option ('D') is issued. The halt state is maintained by a variable

called the global_halt_flag, which is set on receipt of the halt message. In the halt state all interrupt requests are still dealt with, but as long as the global_halt_flag remains set calculations are frozen.

DUMP (option D)

On issuing a dump request the (already halted) array elements have the data at the nodes of the processing mesh extracted using the get_address procedure of the Laplace program, from within the multiprocessor operating system. The array results extracted are sent to the master processor as blocks of data (the number of blocks depending on the size of the mesh) which are then displayed on the screen; an option also exists to store the results on a floppy disk. Further modifications of the program may enable communication to be established with a mainframe computer enabling results to be post processed and graphical output obtained.

CONTINUE (option C)

The continue instruction instructs all PEs previously loaded with the 'L' option, and currently in the halt state, to continue.

OPEN OUTPUT FILE (option V)

This option opens a file on disc (called R.PL9) to which any dumped results will be output, as well as echoed to the screen. The disc must not already contain this file as the overseer will not overwrite an existing file.

CLOSE OUTPUT FILE (option Z)

This option should be issued after the results have all been dumped to the results file, to ensure the correct closure of the file.

SINGLE LOAD (option W)

This instruction is a load to a specific PE. It is similar to the LOAD option (L) except that the user is prompted not for the number of PEs to be loaded, but the unique number of the PE to be loaded.

RUN (option R)

This option will run a previously loaded program, whose execution may start at any valid start location within a PEs RAM. The user is first prompted for the number of PEs on which a program is to be run. The user is then prompted for the start address within that processing element at which execution will begin.

SINGLE RUN (option S)

This option allows the execution of a program in a single PE, where the start address for execution is assumed to be $B000. The user is prompted for the number (in hex) of the processing element on which the user wishes to run a program.

INDIVIDUAL CONTINUE (option B)

This allows a single PE to be told to continue from a halted state. The user is prompted for the unique number of the processor in which the option is to be effected.

COMPASS (option P)

A small procedure showing the directions North, East, South and West orientations of the PEs of the array, as they relate to the master processor.

QUIT (option J)

This option jumps out of the overseer program and back to the point where the MICROBOX II had just loaded FLEX.

REFRESH (option F)

This option clears the screen of any messages or information which are no longer needed.

SILENT RUNNING (option Q)

This option speeds up the time taken to load large programs or when the results of a large field problem are being dumped. This is achieved by turning off the status messages displayed on the V.D.U., the same option can also turn back on the status message display.

CHAPTER 5

CHAPTER 5

RESULTS

1.0 INTRODUCTION

The results of this chapter appear under the following headings:

2.0 Processing array data tests

3.0 Problem convergence tests

4.0 Relative performance of the multiprocessor array.

Each section describes the results it contains, and the salient features of the results are identified. Section 2.0 is designed to show the speedup of the 16PE system over a single processor system with respect to the number of calculations that can be performed within a given time. Section 3.0 is designed to show the speedup of the 16PE system with respect to the rate at which problem solution occurs. Finally, section 4.0 compares the performance of the 16PE system to that of some commercially available machines.

The method by which speedup figures are calculated are detailed for the initial set of results, this method is then followed in subsequent sections where stated.

2.0 PROCESSING ARRAY DATA TESTS

The aim of this section is to display the speedup of the

multiprocessor system, with regard to its potential processing power over a single PE, by analysis of its ability to perform a set number of iterative passes over the data of a given problem.

This section is sub-divided as follows:

2.1 Displays the initial results for speedups;
2.2 Validation of the method used to obtain speedup figures;
2.3 Displays the speedups obtained from an optimized program;
2.4 Displays the optimized results compared to those of an independent system (still based on the same microprocessor).

Speedup has been defined to be the increase in processing power of a number of processors over that of a single processor. In terms of the data of this chapter this can be interpreted as the benefit gained by a multiprocessor system over that of a single PE; or simply the ratio of the time taken to obtain a solution (or to arrive at a predefined state of solution) on the multiprocessor system to the time taken on one PE.

2.1 Initial results

The program of listing 4 was used to obtain the times for five passes of the solution algorithm over various problem sizes, enabling a speedup figure based on the ratio of time taken (as defined above) to be calculated. The times for five passes over various problem sizes can be seen in table 5.1.

data size	1 PE	4 PE	9 PE	16PE
12x12	3.55	1.37	1.13	0.74
24x24	15.8	5.65	3.77	2.28
36x36	37.36	13.15	8.25	4.82
48x48	68.3	23.9	14.5	8.5
60x60	114.1	37.1	21.2	12.2
72x72	167.0	55.1	32.2	18.2
84x84		75.7	43.7	24.7
96x96		99.1	56.8	32.3
108x108		126.6	71.1	40.8
120x120		156.0	88.9	50.2
132x132		190.5	107.4	60.6
144x144		228.0	127.7	72.0
156x156			149.7	84.5
168x168			173.0	96.3
180x180			200.0	112.3
192x192			226.5	128.3
204x204			256.0	144.1
216x216			287.0	161.6
228x228				180.2
240x240				199.0
252x252				220.4
264x264				241.4
276x276				263.4
288x288				286.4

TABLE 5.1 Five pass time (s) of initial program

Rather than use these results directly it is useful to create a figure which can be used for comparison purposes between different sets of results. The average time for one pass of the data is therefore taken. Table 5.2 shows the average pass times corresponding to the data of table 5.1.

array size	1 PE	4 PE	9 PE	16PE
12x12	0.71	0.27	0.23	0.15
24x24	3.16	1.13	0.75	0.46
36x36	7.47	2.63	1.65	0.96
48x48	12.1	4.8	2.9	1.7
60x60	22.8	7.42	4.2	2.4
72x72	33.3	11.0	6.4	3.6
84x84		15.1	8.7	4.9
96x96		19.8	11.3	6.5
108x108		25.2	14.2	8.1
120x120		31.2	17.8	10.0
132x132		38.1	21.4	12.1
144x144		45.6	25.4	14.4
156x156			29.8	16.8
168x168			34.6	19.2
180x180			40.0	22.4
192x192			45.3	25.6
204x204			51.2	28.8
216x216			57.4	32.2
228x228				36.0
240x240				39.8
252x252				44.1
264x264				48.3
276x276				52.7
288x288				57.3

TABLE 5.2 Average pass time (s)

Because the 16PE system has sixteen times the amount of memory as one of the component PE, it is not always possible to obtain speedup figures directly for large problem sizes - using the ratio of the time taken for a solution on a number of PEs to the time taken on 1PE. Memory restrictions preclude the solution of large problems on a relatively small number of PEs, these restrictions can be clearly seen as gaps in the results displayed in table 5.1. In order for speedup figures (based on the above ratio) to be calculated, some means of extrapolating the results for the memory restricted cases above must be used.

Logarithmic plots of the average pass time against problem size for a given architecture, show a high degree of correlation. Figures 5.1, 5.2 and 5.3 display this relationship for 1PE, 4PEs and 9PEs respectively. The equations relating the average pass time (t) to the number of points (p), along one edge of the problem mesh, derived from these relationships are found to be:

1PE $$t = \frac{p^{2.15}}{295.12}$$

4PE $$t = \frac{p^{2.06}}{612.35}$$

9PE $$t = \frac{p^{2.00}}{812.83}$$

The high degree of correlation between t and p justifies the use of the above equations in the extrapolation of the values for t, for which the available memory within the architecture prevents solution.

The logarithmic relationship of t against p for the data of the 16PE case in table 5.2 can be seen in appendix E. This verifies that the linearity observed also occurs in the 16PE system and reinforces the justification of the above equations for use in the extrapolation of data.

Table 5.3 shows the average pass time for the range of architectures and problem sizes (with extrapolated values indicated).

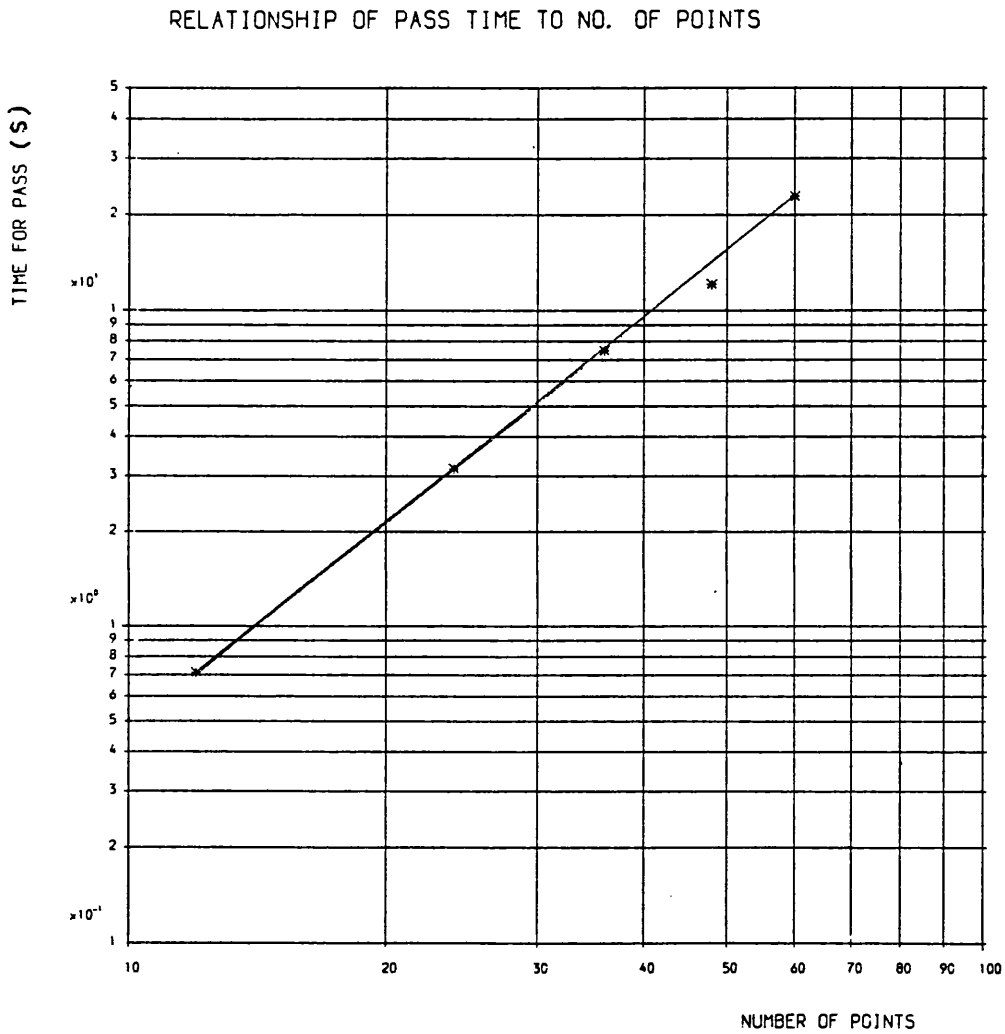


FIGURE 5.1 AVERAGE PASS TIME AGAINST THE NUMBER OF MESH POINTS FOR 1PE

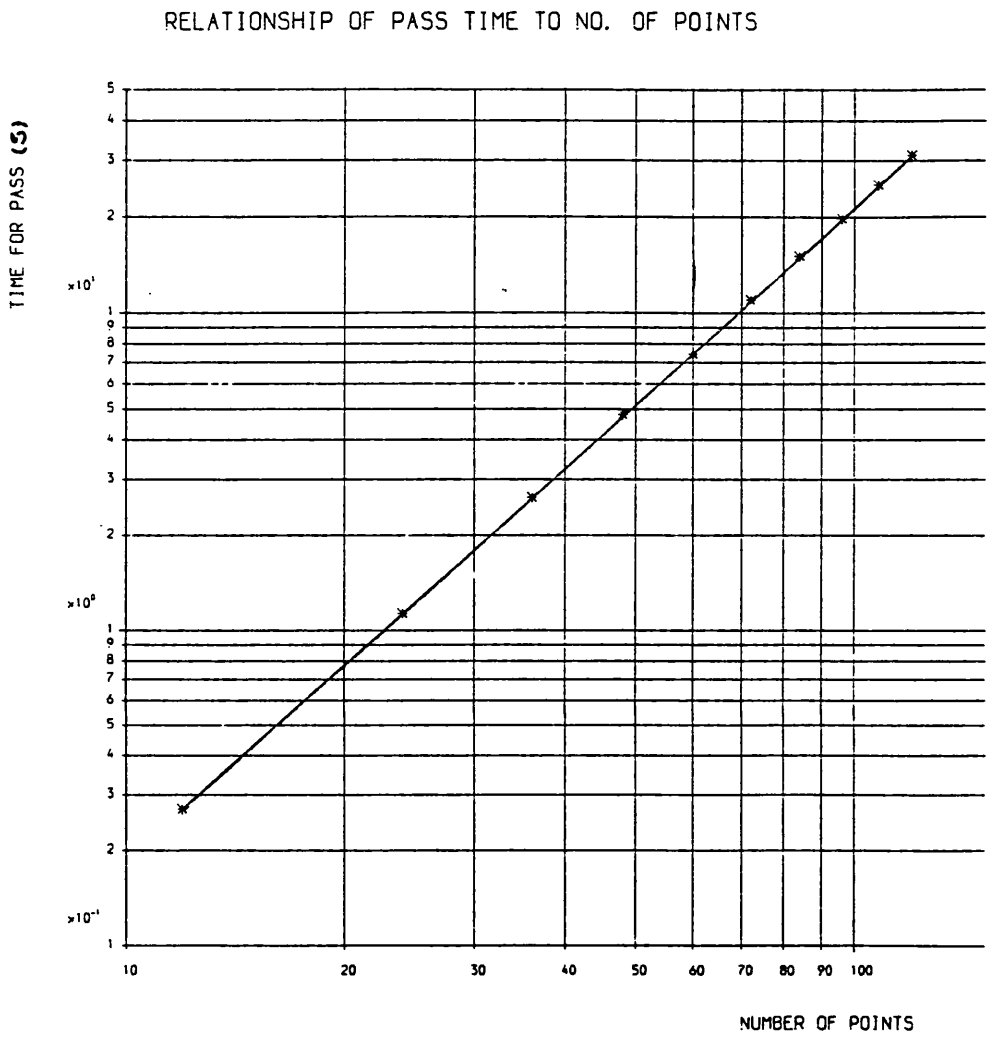


FIGURE 5.2 AVERAGE PASS TIME AGAINST THE NUMBER OF MESH POINTS FOR 4PES

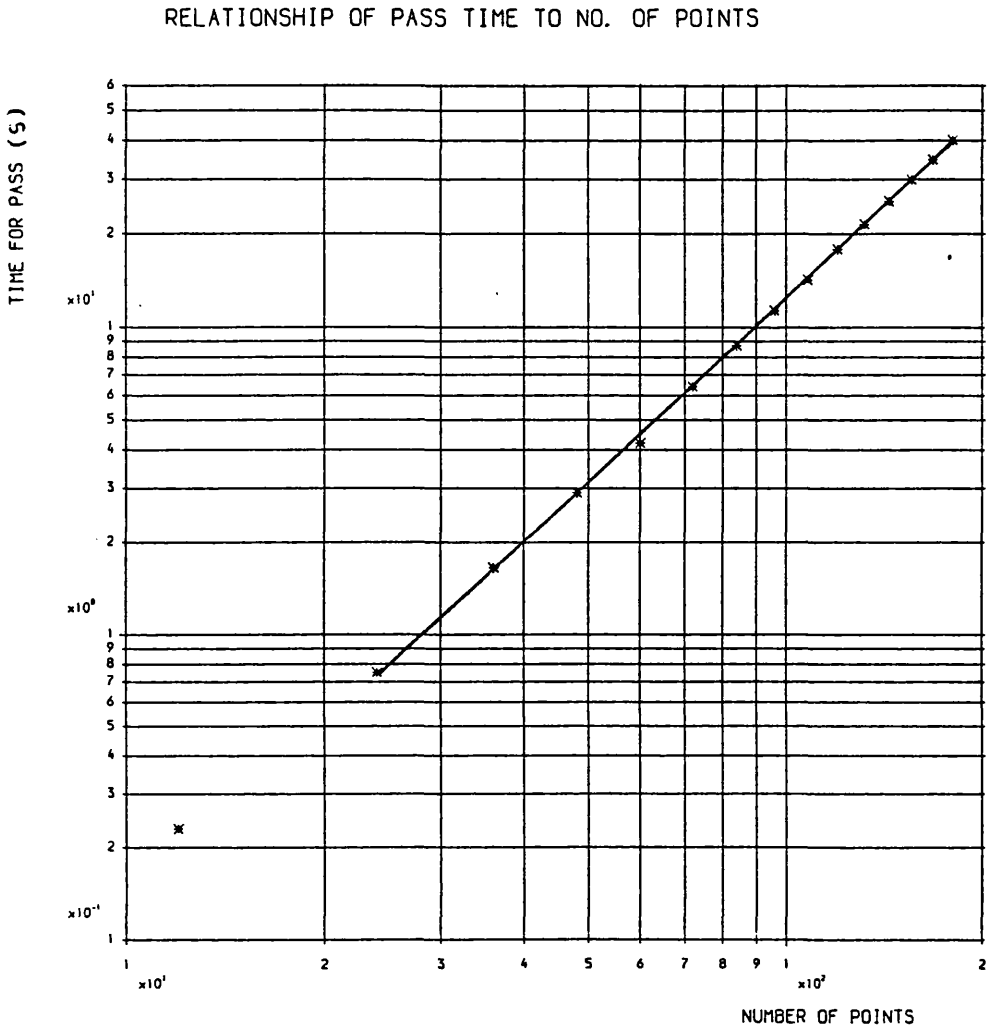


FIGURE 5.3 AVERAGE PASS TIME AGAINST THE NUMBER OF MESH POINTS FOR 9PES

array size	1 PE	4 PE	9 PE	16PE
12x12	0.71	0.27	0.23	0.15
24x24	3.16	1.13	0.75	0.46
36x36	7.47	2.63	1.65	0.96
48x48	13.6	4.8	2.9	1.7
60x60	22.8	7.42	4.2	2.4
72x72	33.3	11.0	6.4	3.6
84x84	46.4	15.1	8.7	4.9
96x96	61.9	19.8	11.3	6.5
108x108	79.7	25.2	14.2	8.1
120x120	100.0	31.2	17.8	10.0
132x132	122.8	38.1	21.4	12.1
144x144	148.1	45.6	25.4	14.4
156x156	175.8	53.8	29.8	16.8
168x168	206.3	62.7	34.6	19.2
180x180	392.2	72.2	40.0	22.4
192x192	274.8	82.5	45.3	25.6
204x204	313.1	93.5	51.2	28.8
216x216	354.1	105.2	57.4	32.2
228x228	397.7	117.6	63.9	36.0
240x240	444.0	130.7	70.9	39.8
252x252	493.2	144.5	78.1	44.1
264x264	545.0	159	85.7	48.3
276x276	599	174.3	93.7	52.7
288x288	657	190.3	102.0	57.3

e x t r a p o l a t e d
a r e a

TABLE 5.3 Average pass time (s)

From the data of Table 5.3, Table 5.4 shows the corresponding speedup figures based on the time taken for the same problem size by one PE.

array size	1 PE	4 PE	9 PE	16PE
12x12	1	2.63	3.08	4.73
24x24	1	2.79	4.21	6.87
36x36	1	2.84	4.53	7.78
48x48	1	2.83	4.68	8.0
60x60	1	3.07	5.43	9.5
72x72	1	3.03	5.2	9.25
84x84	1	3.07	5.33	9.46
96x96	1	3.13	5.47	9.52
108x108	1	3.16	5.61	9.83
120x120	1	3.2	5.62	10.0
132x132	1	3.22	5.74	10.15
144x144	1	3.25	5.83	10.3
156x156	1	3.26	5.89	10.46
168x168	1	3.29	5.96	10.74
180x180	1	3.31	5.98	10.7
192x192	1	3.33	6.06	10.73
204x204	1	3.35	6.11	10.87
216x216	1	3.36	6.17	11.0
228x228	1	3.38	6.22	11.04
240x240	1	3.39	6.27	11.15
252x252	1	3.41	6.31	11.18
264x264	1	3.42	6.36	11.2
276x276	1	3.43	6.39	11.4
288x288	1	3.45	6.44	11.46

TABLE 5.4 Speedups (relative to performance of 1PE)

This data is displayed three-dimensionally in figure 5.4, which shows speedup as a function of architecture size and the size of the problem. The speedup figures can be seen to fall of dramatically, as expected, for problem sizes where contentions are more frequent (i.e. in the region where $p^2<20000$). The speedup can be seen to be an almost linear function of the number of PE for values of p^2 out of the severely degraded zone.

The data may also be viewed in relation to the ideal, in figure 5.5. The relationship of speedup with the number of PEs can again be seen to be linear (for $p^2>20000$).

The best performance of the 16PE system is 28% less than the ideal and 24% less than the value predicted in chapter 2. The reason for this

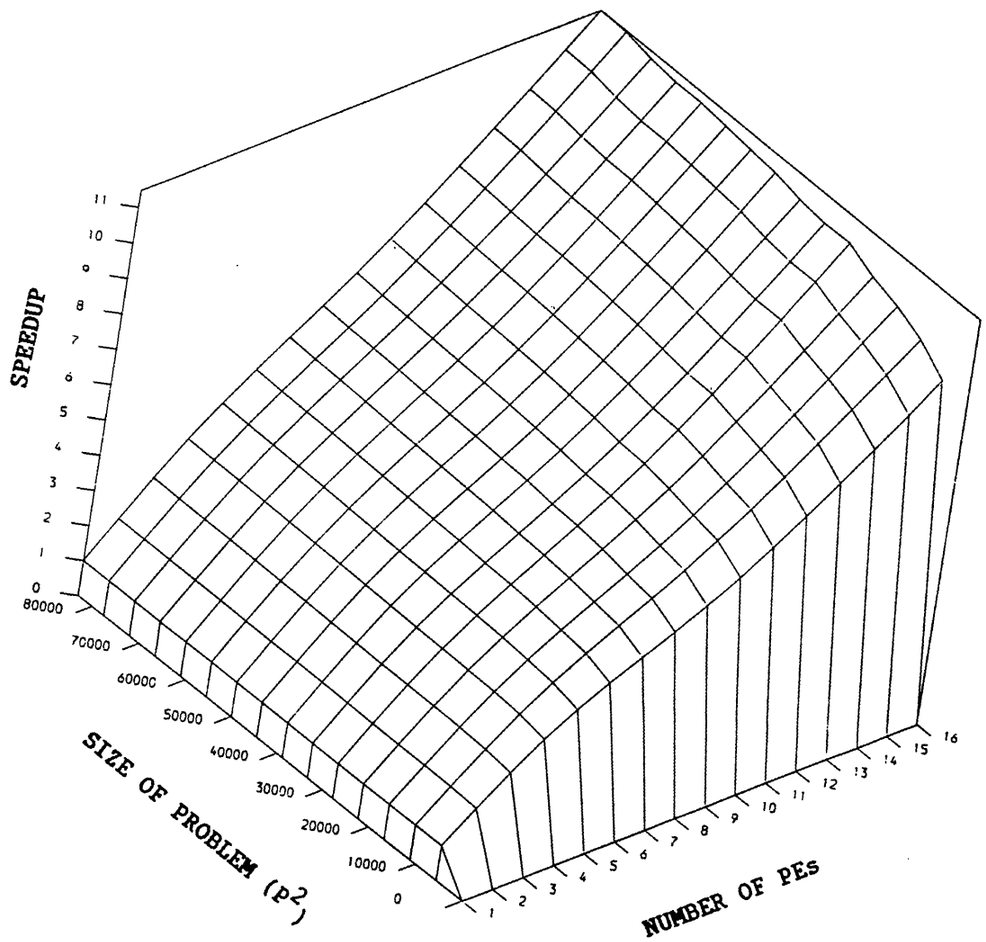


FIGURE 5.4 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES AND THE SIZE OF THE PROBLEM MESH

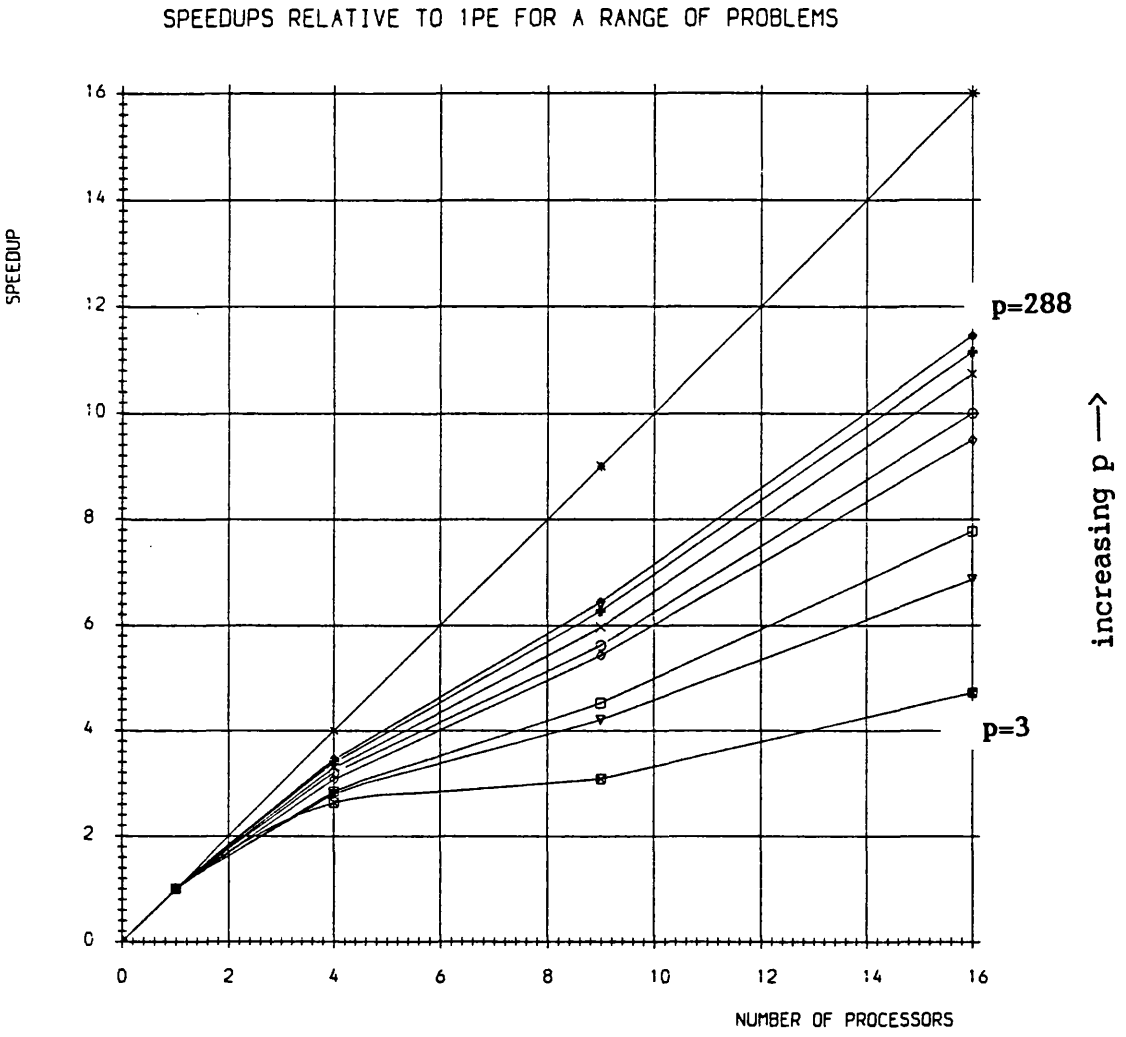


FIGURE 5.5 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES, FOR A RANGE OF PROBLEM SIZES

unexpectedly severe degradation in performance and the method by which an improvement was made is discussed in chapter 6. Improved results were obtained using an optimized program the results of which may be seen in section 2.3.

2.2 Validation of the method used to obtain speedup figures

It could be argued that taking the time for 5 passes over the data may not be truly representative of the shared memory switching, which occurs when running programs with run times of several hours. In order to check that taking the times for 5 passes of the data is truly representative of the shared memory switching which takes place for longer run times, a further experiment was undertaken.

The time taken to pass 16 times over the data for a problem mesh size where p=60 was taken. This data and the corresponding results of the 5 pass time case can be seen in table 5.5, where the average pass time can be seen enclosed in parenthesis.

grid size 60x60	16 pass time (s)	5 pass time (s)
1PE	353.6 (22.1)	114.1 (22.8)
4PE	123.4 (7.77)	37.1 (7.42)
9PE	69.9 (4.37)	21.2 (4.24)
16PE	39.7 (2.48)	12.2 (2.44)

TABLE 5.5 16 and 5 Pass times

Figure 5.6 displays the average pass times of the data for the 5 and 16 pass case. The average pass times of the 5 pass case are all within 4% of the corresponding time for the 16 pass case. The error in obtaining the pass time data is estimated at 2%, since readings were obtained from a hand operated stop watch activated by changes in processors status, as indicated on the status monitor board. Within the 2% tolerance the pass times of the 5 and 16 pass data agree with each other, which implies that any error is zero or negligible. Since any error should be consistent in all cases and as speedup is calculated as a ratio of two average pass times, any errors in the average pass times that do exist should be cancelled out during the calculation of the speedup. The method of obtaining speedup figures can therefore be said to be valid.

2.3 Optimized results

The speedup figures of section 2.1 were less than had been predicted in chapter 2. The larger than expected degradation in the speedup figures was due to the get_address procedure (described in chapter 4), the details of why this was the case are discussed in chapter 6. A more efficient version of this procedure was written, the details of which are again left until chapter 6, and the tests of section 2.1 re-run.

Appendix F contains the results of the re-run using the optimized program, following the method of section 2.1 to enable a table of average pass times to be found, from which the speedup figures of table 5.6 are calculated.

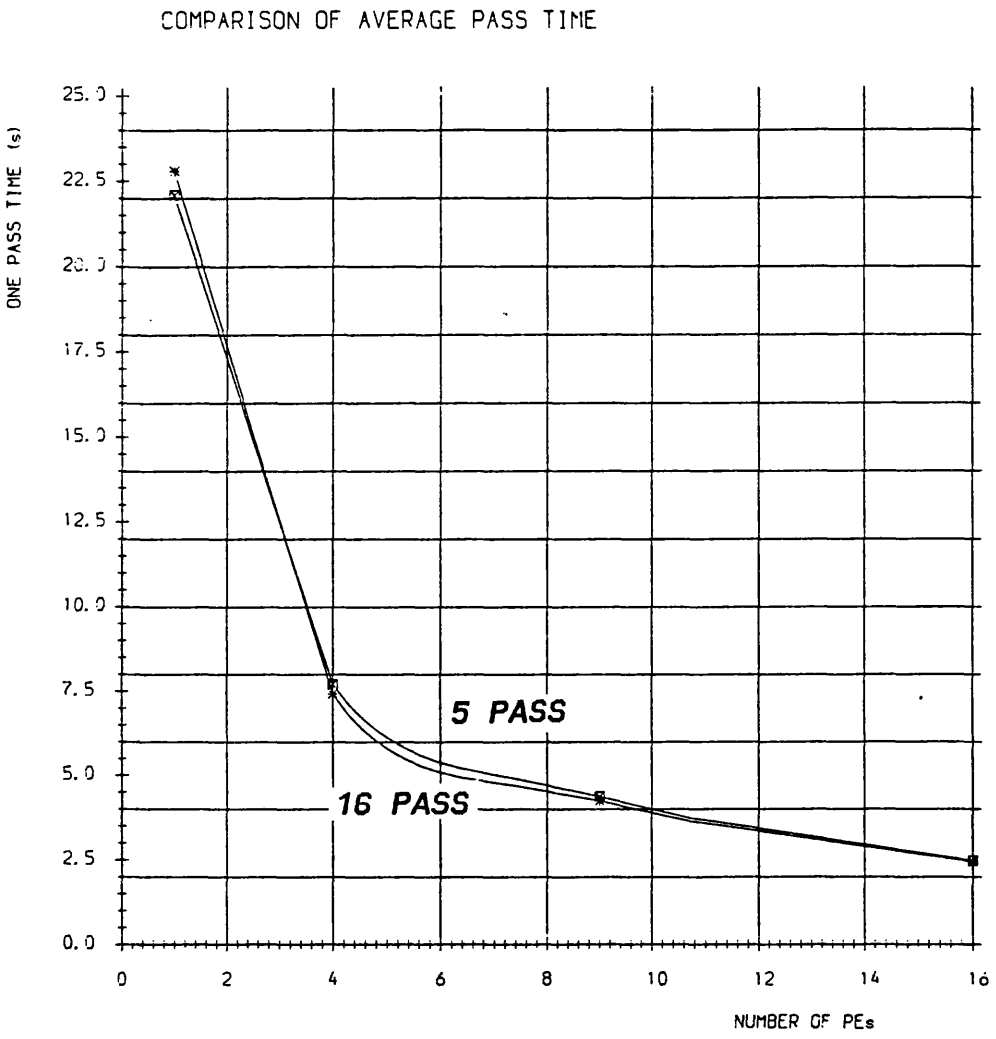


FIGURE 5.6 COMPARISON OF AVERAGE PASS TIMES FOR 16 AND 5 PASSES OVER THE DATA

size	1PE	4PE	9PE	16PE
12x12	1	2.76	4.24	5.54
24x24	1	3.18	5.50	10.58
36x36	1	3.26	6.61	11.30
48x48	1	3.29	6.65	11.90
60x60	1	3.27	6.95	12.30
72x72	1	3.28	7.03	12.42
84x84	1	3.31	7.10	12.69
96x96	1	3.33	7.24	12.68
108x108	1	3.34	7.33	13.03
120x120	1	3.36	7.40	13.11
132x132	1	3.38	7.46	13.20
144x144	1	3.39	7.50	13.30
156x156	1	3.41	7.56	13.43
168x168	1	3.41	7.62	13.50
180x180	1	3.42	7.65	13.60
192x192	1	3.41	7.69	13.68
204x204	1	3.43	7.73	13.74
216x216	1	3.44	7.75	13.78
228x228	1	3.45	7.79	13.87
240x240	1	3.45	7.77	13.93
252x252	1	3.46	7.85	13.98
264x264	1	3.47	8.02	14.04
276x276	1	3.47	7.91	14.09
288x288	1	3.47	7.95	14.10

Table 5.6 Speedup figures (optimized program)

This data can be viewed three-dimensionally in figure 5.7, which shows speedup as a function of architecture (i.e. the number of processing elements) and problem size (p^2). Degradation can be seen in the region where $p^2<10000$, again due to the increased number of contentions which are seen at PE boundaries. The actual shape of the surface is similar to that of figure 5.4, but the magnitudes of the speedups are much improved (seen more clearly in figure 5.8).

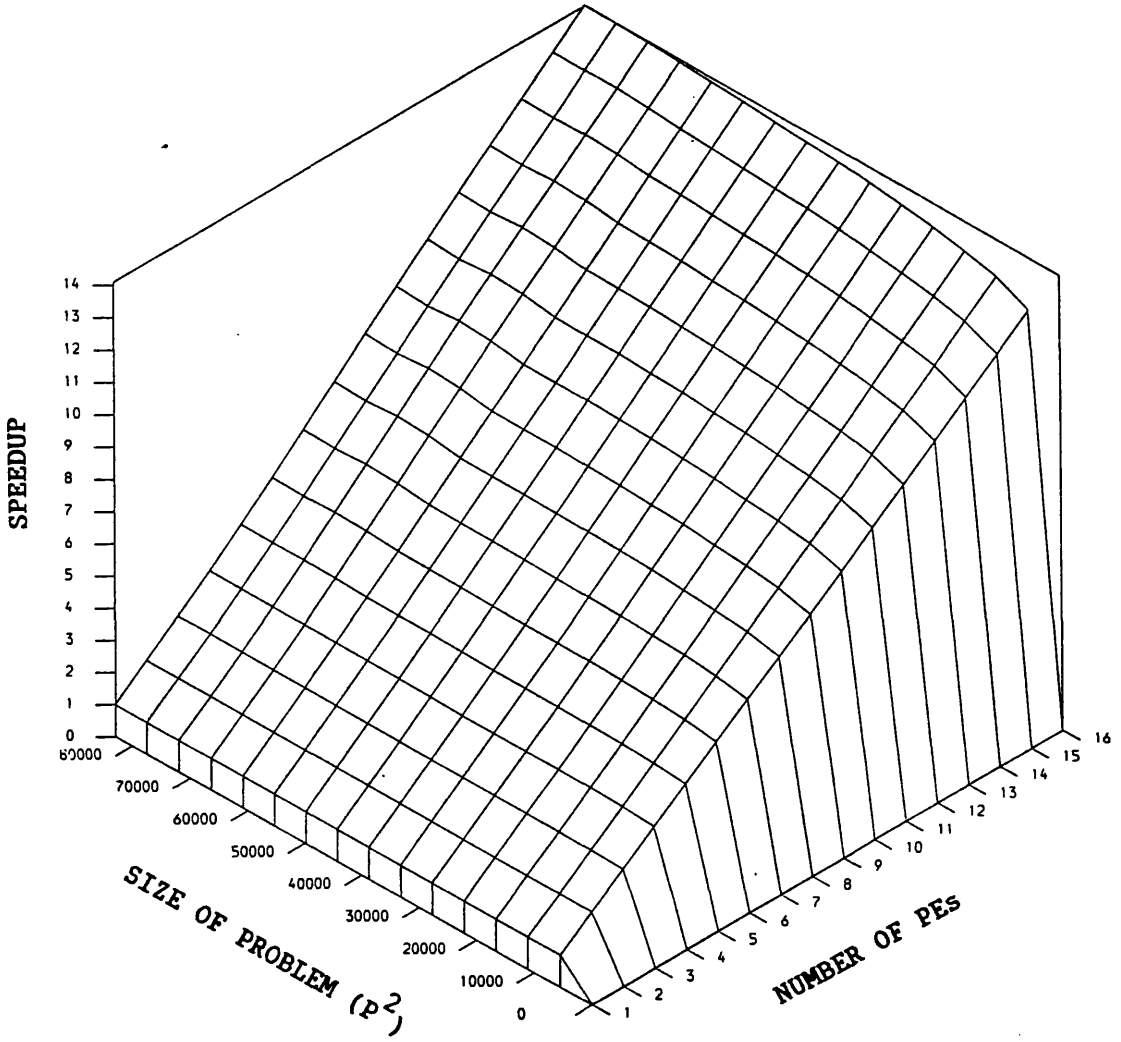


FIGURE 5.7 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES AND THE SIZE OF THE PROBLEM MESH

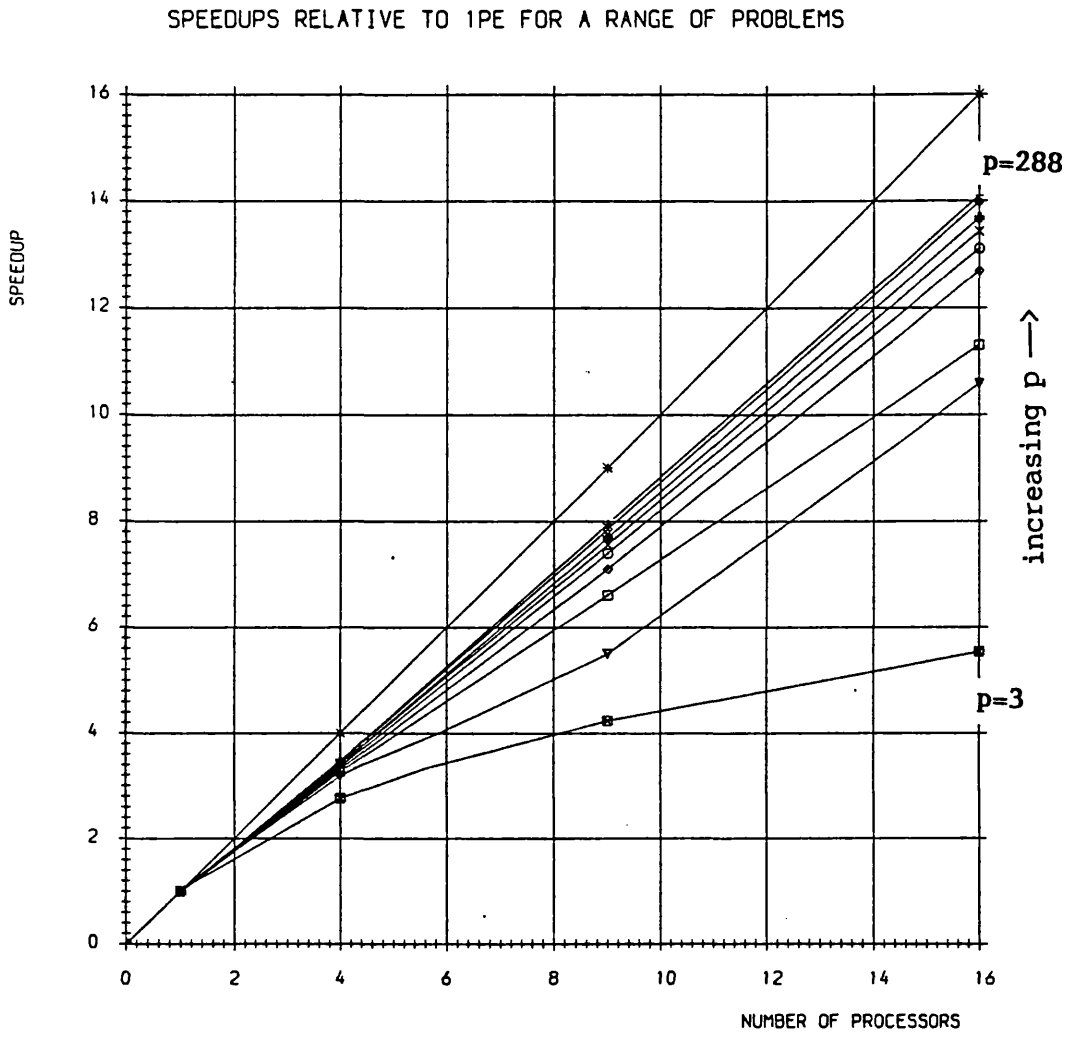


FIGURE 5.8 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES, FOR A RANGE OF PROBLEM SIZES

Figure 5.8 shows the speedup for selected problem sizes, with the ideal speedup shown for reference. The performance of the array can be seen to be linear, for values of p which take the multiprocessor out of the area of severe degradation. This linear trend displays no sign of falling off as the number of PEs increases, and achieves a speedup 88% that of the ideal, and 94% of the value predicted in chapter 2.

2.4 Comparison with independent computer based on the same processor

In order that the performance of the multiprocessor system can be judged with reference to an independent computer (still based on the same processor) and not just to a component PE of the machine, the same tasks performed by the multiprocessor were performed by a MICROBOX II (a MC6809E based microcomputer).

The storage of data within the MICROBOX II does not need a 'get_address' procedure, since there is no shared memory and data does not need to pass to any other system. The tasks of the previous section were performed on the MICROBOX II with the data held in a two-dimensional array constructed as suggested by WINDRUSH [94]. Without the overhead of a 'get_address' procedure the results for the MICROBOX II are expected to be faster than those of a component PE of the multiprocessor, as a result of which the speedup figures of the multiprocessor are expected to be lower.

Appendix G contains the results of running the same problems on the MICROBOX II, and the processing of these results needed to produce a table of average pass times (extrapolated where necessary). The speedup figures are calculated, as before, as the ratio of the average pass time of a single PE to that of the multiprocessor. The average pass times of the

multiprocessor are those of the previous section (using the optimized program), the single PE in this case is taken to be the MICROBOX II (whose average pass times are those in appendix G), from which the speedup figures of table 5.7 are obtained.

size	1PE	4PE	9PE	16PE
12x12	0.78	2.16	3.31	4.32
24x24	0.81	2.59	4.52	8.61
36x36	0.82	2.66	5.4	9.29
48x48	0.81	2.69	5.44	9.79
60x60	0.82	2.71	5.7	10.13
72x72	0.83	2.72	5.81	10.26
84x84	0.82	2.75	5.94	10.46
96x96	0.82	2.76	5.98	10.47
108x108	0.83	2.79	6.07	10.78
120x120	0.82	2.81	6.13	10.86
132x132	0.83	2.82	6.19	10.96
144x144	0.83	2.83	6.23	11.09
156x156	0.83	2.84	6.29	11.17
168x168	0.83	2.85	6.34	11.25
180x180	0.83	2.84	6.37	11.35
192x192	0.83	2.87	6.4	11.42
204x204	0.83	2.87	6.45	11.47
216x216	0.84	2.89	6.48	11.53
228x228	0.84	2.89	6.51	11.61
240x240	0.84	2.90	6.5	11.66
252x252	0.84	2.90	6.59	11.73
264x264	0.84	2.91	6.61	11.77
276x276	0.84	2.92	6.64	11.82
288x288	0.84	2.92	6.67	11.83

extrapolated area

Table 5.7 Speedup of the multiprocessor based on the performance of an independent microcomputer

This data may be viewed three-dimensionally in figure 5.9, which shows speedup as a function of architecture (the number of processing elements) and problem size (p^2). Again for values of $p^2>10000$ linearity of speedup with the number of PEs is observed, and severe degradation for $p^2<10000$ as in the previous section.

The effect of the MICROBOX II not having the overhead of a 'get_address'

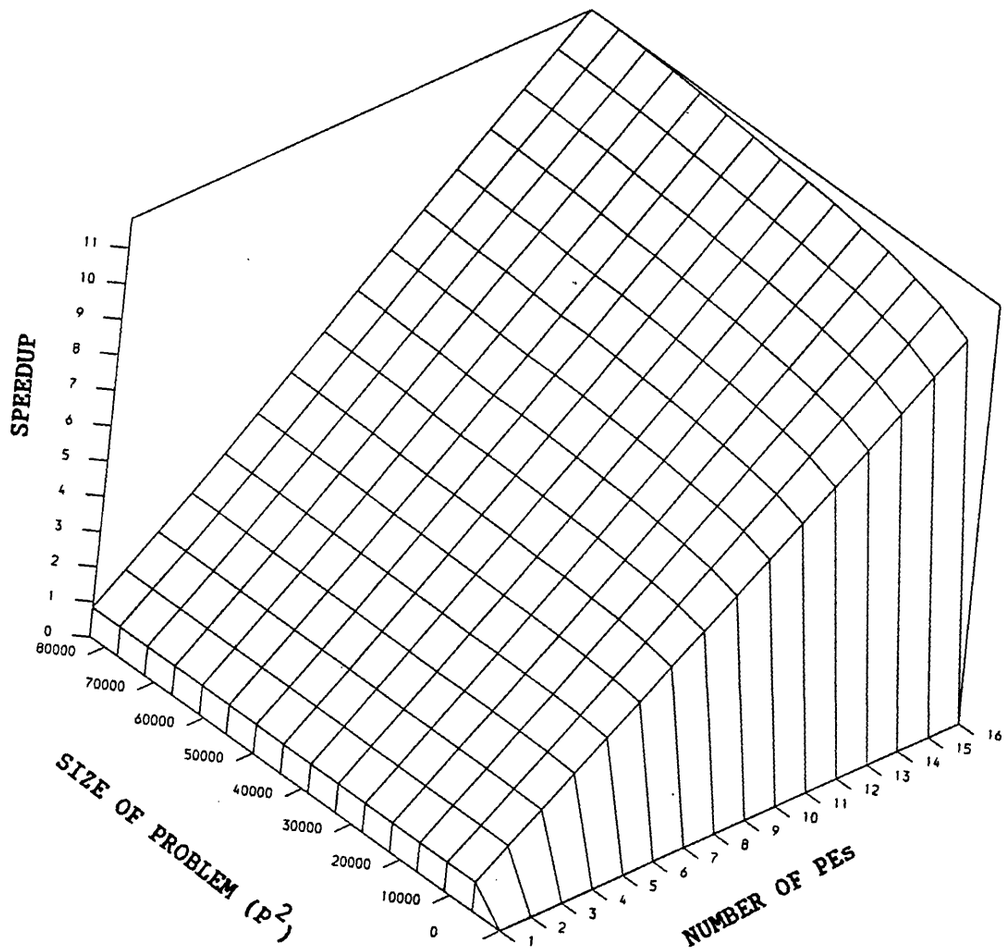


FIGURE 5.9 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES AND THE SIZE OF THE PROBLEM MESH

procedure can be seen in table 5.7, where 1PE is an average of 17% slower than the MICROBOX. This is also reflected in figure 5.10 where the maximum speedup is degraded to 74% of the ideal (compared to the 88% figure when based on a component PE). However, the linear speedup with the number of PE (for large p) can be seen clearly, and as in all cases to date shows no sign of falling off with an increased number of PE.

3.0 PROBLEM CONVERGENCE TESTS

Chapter 6 discusses the possibility of a multiprocessor increasing the stability of a field equation solving algorithm, and links this with the rate at which values propagate across a problem mesh and the time taken to reach a solution. Program 5 is designed to test the multiprocessors ability to speedup the time to a solution. The program is described in chapter 4 and runs the same Laplace problem as before, but the times taken in this case are those of the multiprocessor reaching a predefined state of solution (in this case when all nodal data points are above 22.2% of the prescribed boundary value).

The speedup figures of this section therefore reflect the speedup of the multiprocessor system based on problem solution time, rather than in the previous section where speedup was that of the increase in pure processing power.

This section is further divided into two sections:

3.1 Examines the speedup based on problem solution times of the multiprocessor, when compared to the performance of a component PE.

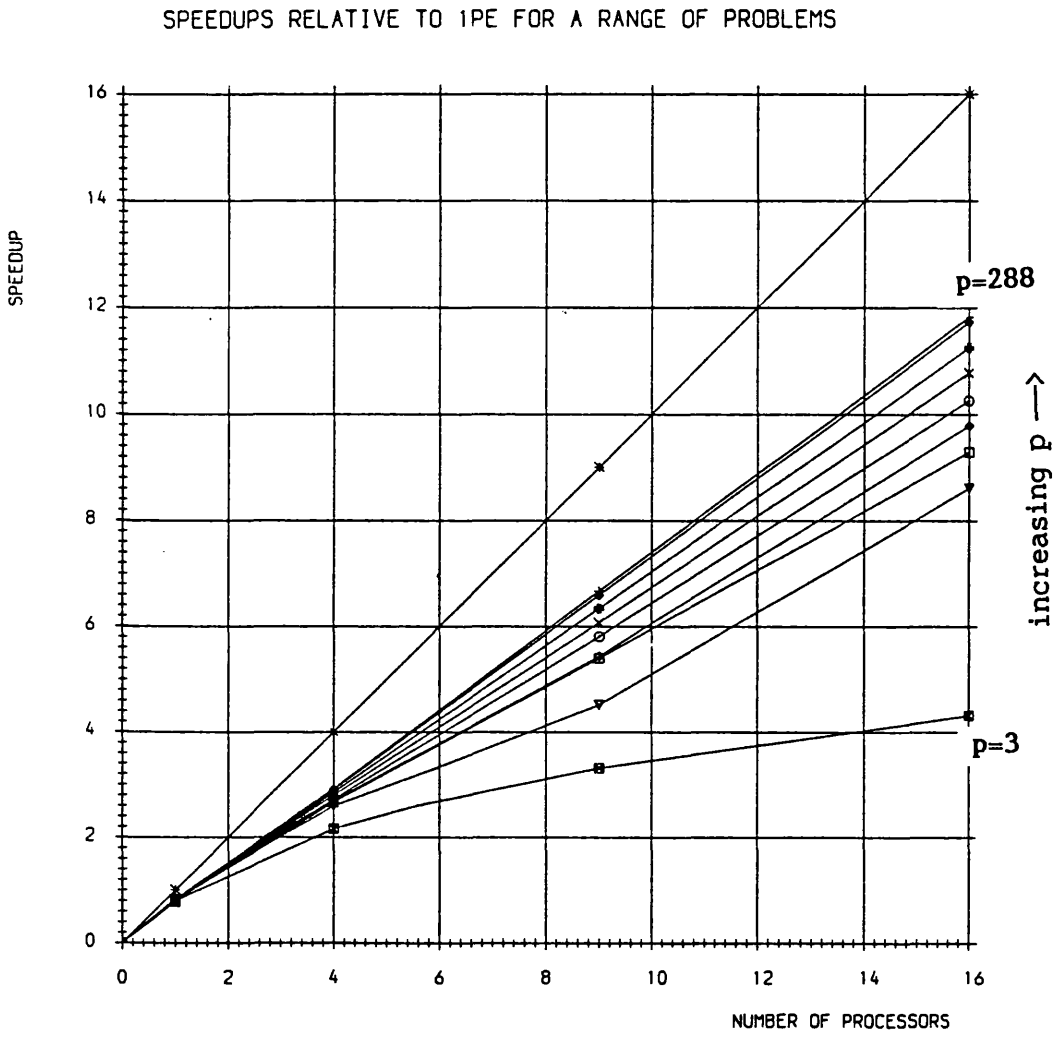


FIGURE 5.10 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES, FOR A RANGE OF PROBLEM SIZES

3.2 Examines the speedup based on problem solution times of the multiprocessor, when compared to the performance of an independent microcomputer - the MICROBOX II.

3.1 Speedups based on problem solution times with reference to a component PE of the multiprocessor array

The data of this section is obtained when running program 5 (described in chapter 4) on the multiprocessor array. The times are obtained, in different architectures for varying problem sizes, to reach a state of solution where all nodes have reached an arbitrary value (in this case 22.2% of the boundary value). These times will be representative of the actual time taken to completely solve the problem. Appendix H contains the results and the subsequent processing of the results using the method of section 2.1 (extrapolating where necessary) for the times taken to reach this state of solution on the multiprocessor. This enables the speedup figures of table 5.8 to be calculated.

Once the extrapolating equations are found they are checked by using them to predict solution times for problem sizes, which are then validated by running the program. In the case of the 16PE system validation of the equation was achieved in the prediction of a run time of 50 hours and 32 minutes with an accuracy of 0.7%.

size	1PE	4PE	9PE	16PE
24x24	1	3.29	6.75	11.41
36x36	1	3.27	7.15	12.32
48x48	1	3.30	7.21	12.74
72x72	1	3.31	7.42	13.24
144x144	1	3.33	7.88	14.54
216x216	1	3.33	8.16	15.36
288x288	1	3.34	8.36	15.97

Table 5.8 Problem convergence speedups based on the performance of a component PE

This data can be seen in figure 5.11, where speedup can be seen as a function of the number of processing elements and the size of the problem mesh. Degradation can again be seen in the region where where the value of p is small, but the speedup figures obtained (for larger numbers of PE and large p) are almost that of the ideal linear speedup (where the ideal is that of the increase in pure processing power). In the case where p=288 the speedup obtained for 9PE is within 7.2% of this ideal and for 16PE within 0.2%.

A more important result becomes apparent, when analysing the shape of the relationship seen between speedup and the number of PEs. It can be seen that the speedup is not linear but appears such that;

$$\frac{\partial S}{\partial N} \alpha N^c$$

for large p where c is an unknown constant>1
S is the speedup
N is the number of PEs.

There is no sign of the relationship of speedup to the number of PEs falling off for larger values of PE. In fact if this equation holds true it is conceivable that speedups greater than that expected by the increase in pure processing power alone may be achieved, in respect of the time taken

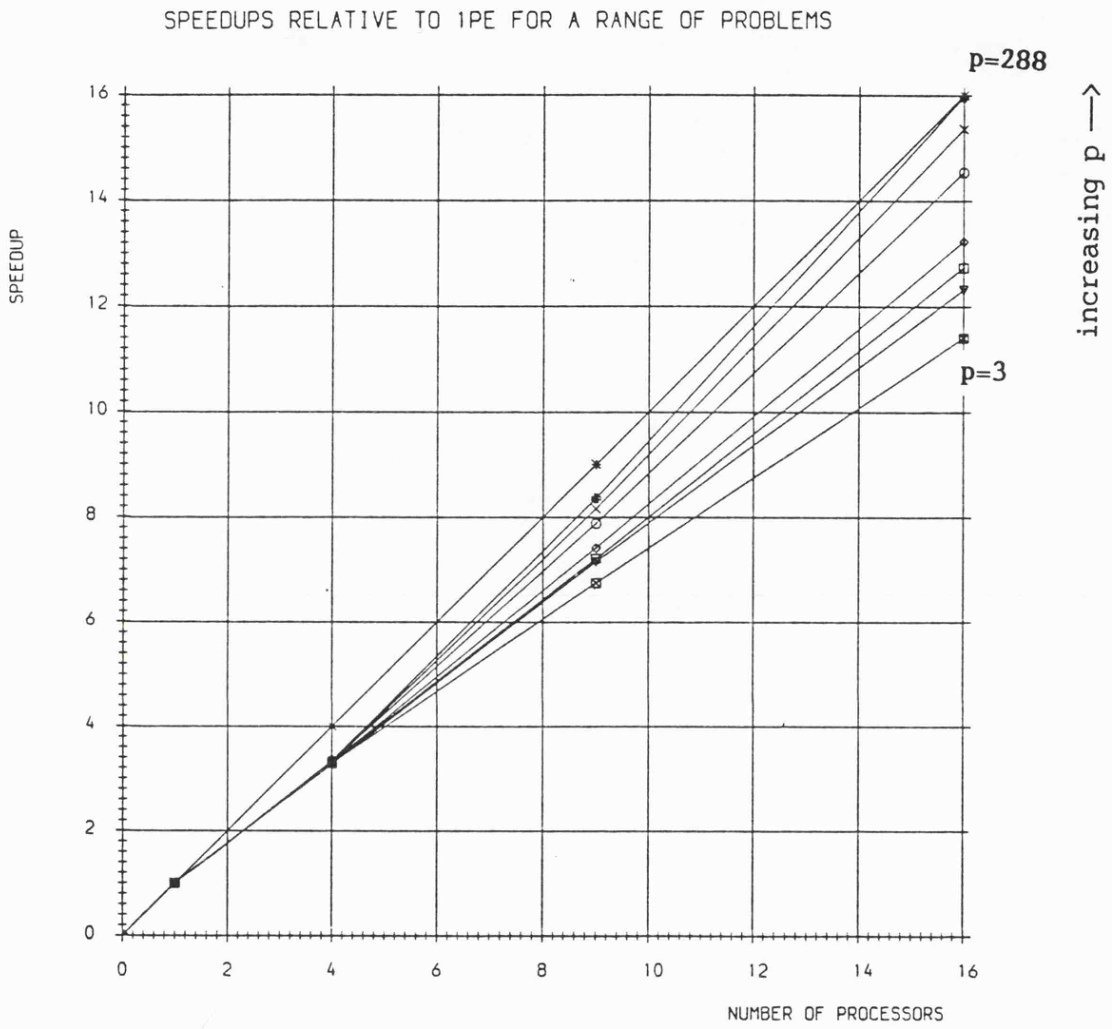


FIGURE 5.11 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES, FOR A RANGE OF PROBLEM SIZES

to solve a problem. This can be shown if the actual time for the 16PE system running the p=288 problem is used, rather than the result obtained from the line of best fit. In this case a speedup figure of 16.08 was achieved (see appendix H), which is slightly above this 'ideal'. At this stage the accuracy of the extrapolation for the time taken by 1PE may be questioned (not the 16PE case as this has been verified), but there is no evidence of these relationships breaking down in the work which has been conducted, and even if a slight error did exist the shape of the relationships would still be similar.

The above relationship of the slope of the curve implies that the more PEs there are, the greater the benefit of the multiprocessor system. The relationship quoted is only true when the system is out of the area of performance where severe degradation takes place, which would indicate that for a set size of problem an optimum size of architecture could be found to speed up the solution.

The processing power of such a multiprocessor architecture with speedup not falling off with the number of PEs is, in theory, infinite! On the basis of problem solution time, an ideal relationship for speedup (where the ideal is based on the increase in pure processing power of the system) is the minimum speedup obtainable for problems where p is large. With such an architecture a system aiming for infinite performance would not be shelved for the usual reason of diminishing returns, as every extra £ buys at least the same performance as the last.

3.2 Problem convergence of the multiprocessor system compared with an independent system

In order to compare the performance of the multiprocessor with an independent system, the times were taken for the MICROBOX II when running a PL9 suggested construct of the previous task, for similar problem sizes. These can be seen in appendix I, along with the processing of results necessary to obtain a table of solution times (extrapolated where necessary) for the problem sizes of the previous section.

The multiprocessor solution times are obtained in section 3.1. Table 5.9 shows the speedup figures of the multiprocessor array based on the performance of the MICROBOX II (from appendix I).

array size	1PE	4PE	9PE	16PE
24x24	0.84	2.78	5.71	9.64
36x36	0.85	2.79	6.08	10.49
48x48	0.87	2.87	6.23	11.08
72x72	0.87	2.88	6.52	11.84
144x144	0.91	3.01	7.24	13.76
216x216	0.92	3.06	7.63	14.90
288x288	0.93	3.10	7.93	15.70

extrapolated area

Table 5.9 Speedups of the multiprocessor array based on the performance of the MICROBOX II

Plotting the data as before, figure 5.12 shows the speedup of the array relative to the MICROBOX II, for a range of problem sizes and architectures. The same characteristics as figure 5.11 can be seen. At high values of p the performance of the 16PE array is only slightly degraded from the values of the previous section, even though the speedups are based on the performance of the MICROBOX II (running without the overhead of a 'get_address' procedure). Speedup appears to be more dependent on the

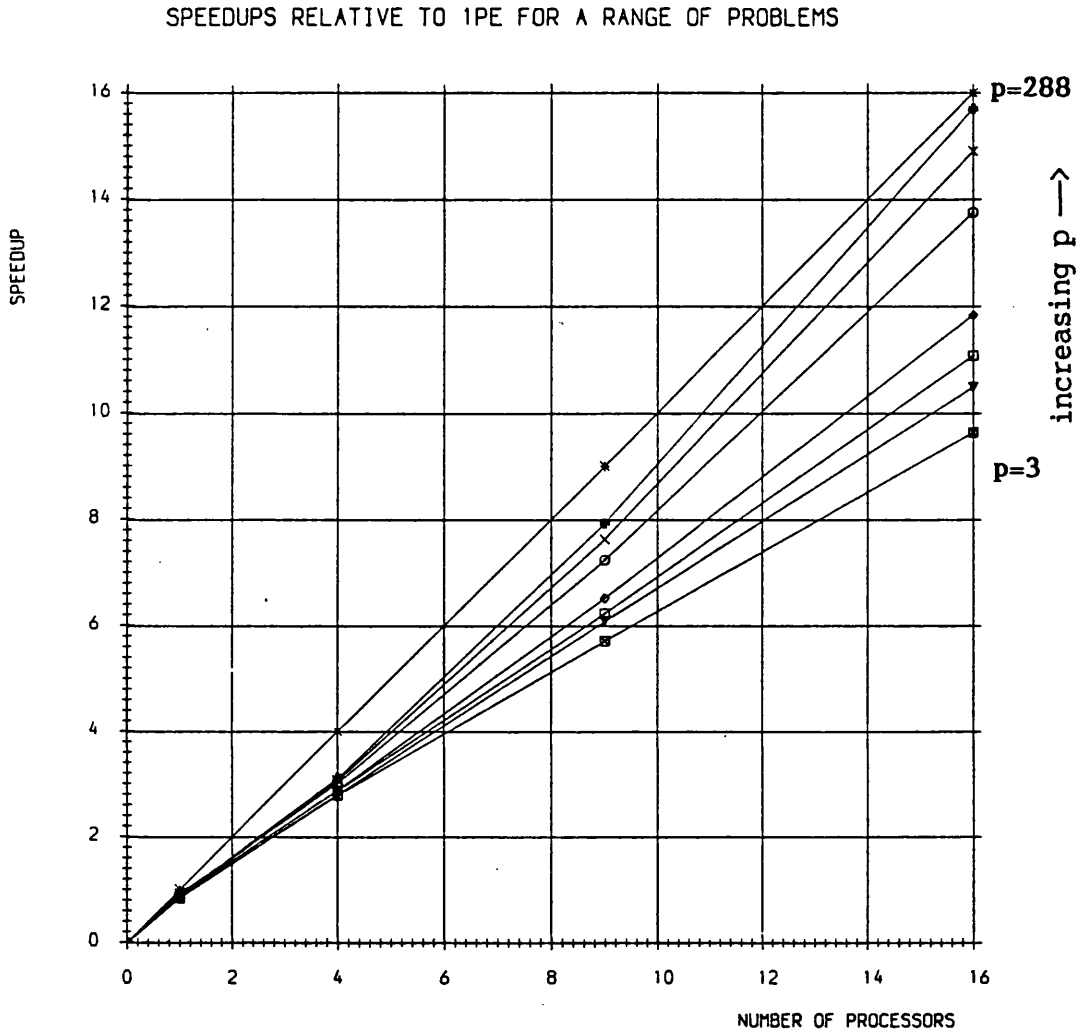


FIGURE 5.12 SPEEDUP AS A FUNCTION OF THE NUMBER OF PES. FOR A RANGE OF PROBLEM SIZES

number of PEs and less on performance of the base system. This, and the fact that the shape of the relationship between speedup and the number of PEs is similar to the results of the previous section, would seem to support the hypothesis that:

$$\frac{\partial S}{\partial N} \propto N^{c}$$

(symbols previously defined) .

4.0 RELATIVE PERFORMANCE OF THE MULTIPROCESSOR ARRAY

This section is designed to display the performance of the multiprocessor system in relation to commercially available machines, with regards to performance criterion. Two criterion are investigated; firstly a straight forward comparison of performance and secondly a comparison is made on the cost of that performance. The bench mark being run on all machines in this section is the Laplace solver, with a problem size of 288x288 nodes (or an estimate, where indicated, of the performance at this size of problem), since the best performance of the multiprocessor array occurred for this problem size.

Programs were developed in the FORTRAN language as specified in chapter 4, all using the same algorithm for the solution of the field equation[1]. The solution times on these systems can be seen in table 5.10.

---

[1] Where possible compilation was carried out without any optimization to compensate for the deficiencies of the one pass PL9 compiler.

MACHINE	TIME TO SOLUTION
MICROBOX	2720982 s
*OLIVETTI M24	276307 s
16PE	183143 s
PRIME	27720 s
** VAX 780	10470 s
VAX 785	5983 s
VAX 8600	2356 s

Table 5.10 Time to solution

* estimated from the data in appendix J.

** estimated from a Digital Equipment Corporation (DEC) bench mark.

The corresponding performance of the above systems relative to the 16PE system, and the cost of the machine, can be seen in table 5.11.

MACHINE	PERFORMANCE RELATIVE TO THE 16PE MACHINE	COST (1000 £)	PERFORMANCE/ COST (x0.001 £)
* BBC (B+)	0.025	0.3	0.0833
MICROBOX II	0.0625	0.35	0.1785
** OLIVETTI M24	0.6628	2.5	0.265
16PE SYSTEM	1.0000	1.95	0.5128
PRIME	6.61	70.00	0.0944
VAX 780	17.6	100.00	0.1760
VAX 785	30.9	150.00	0.206
VAX 8600	77.2	250.00	0.308

Table 5.11 Performance of commercially available machines compared to that of the multiprocessor

* an estimated performance based on comparisons made at low values of p using interpreted BASIC (not compiled) code.

** estimate based on the work in appendix J, which follows a similar method to that used throughout the chapter to estimate the time taken for the OLIVETTI M24.

Performance data may be seen as a function of cost in figure 5.13, where the performance bandwith of commercially available machines is indicated by hatching. The performance of the multiprocessor lies outside this band at a level where the graph indicates that a more expensive machine would normally be needed to achieve this performance. Chapter 6 discusses the possibility of using more powerful processors, creating a new area of performance in which multiprocessor systems would operate at levels higher than would usually be available for this price.

The performance/cost as a function of cost can be seen graphically in figure 5.14, which indicates the benefits that a correctly chosen architecture can have for a multiprocessor system. The multiprocessor system offers a much higher performance per unit cost (more than 1.5 times that of its nearest rival) than the other systems displayed. Chapter 6 discusses how the performance/cost ratio is affected by increasing the power of the PEs and the possible competition from relatively cheap Restricted Instruction Set Computers (RISCs).

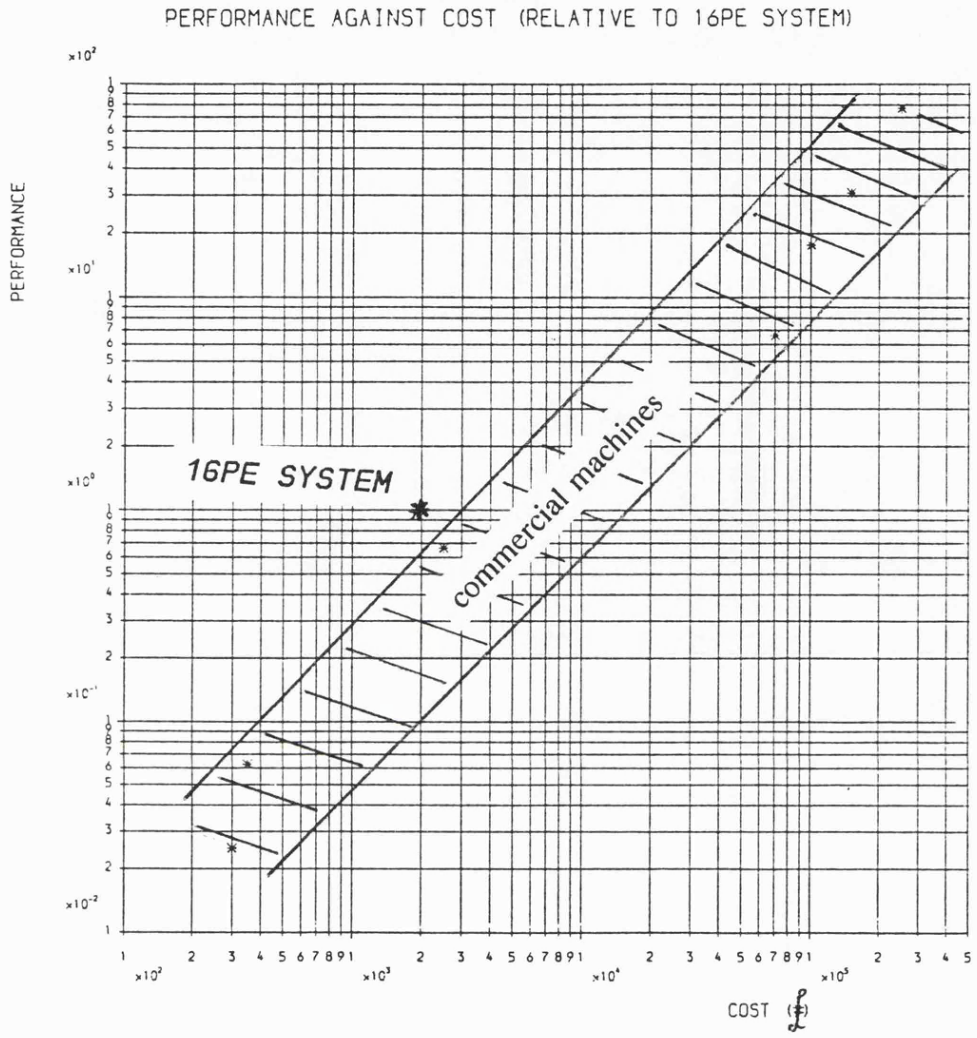


FIGURE 5.13 PERFORMANCE AGAINST COST

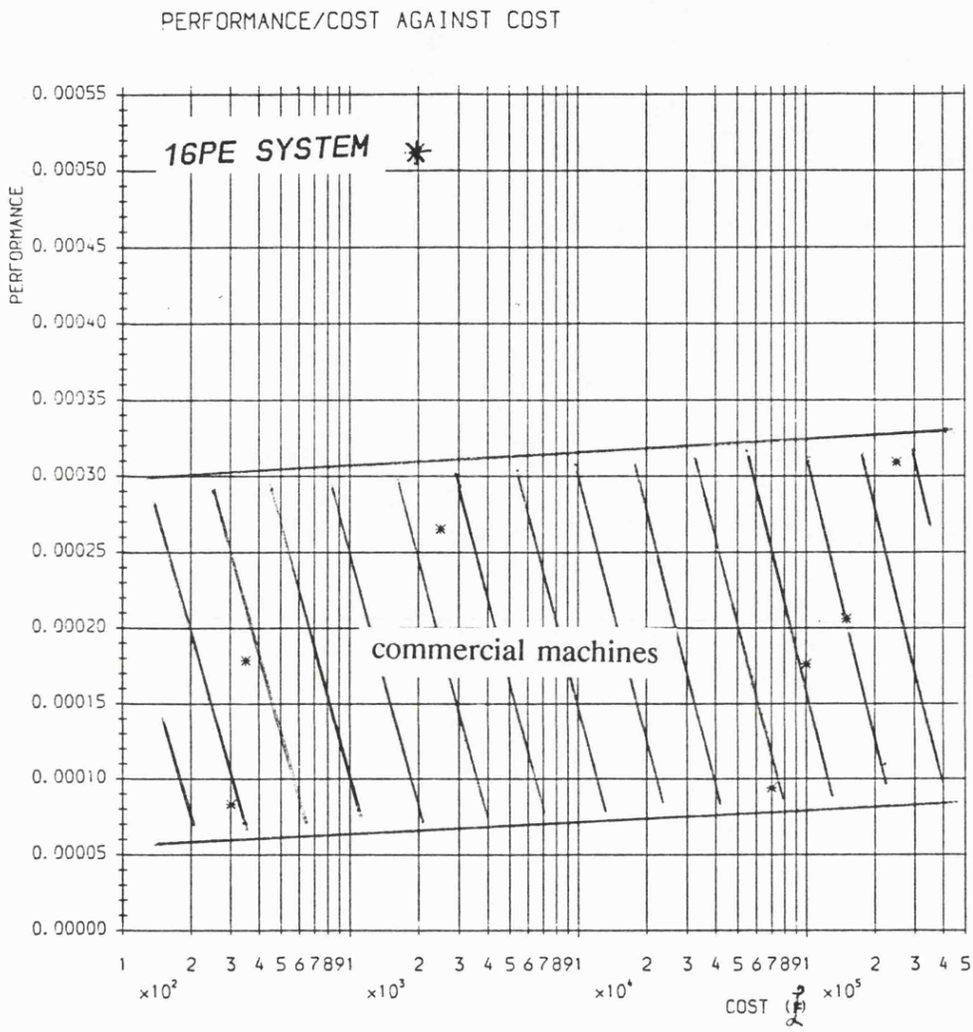


FIGURE 5.14 PERFORMANCE/COST AGAINST COST

CHAPTER 6

CHAPTER 6

DISCUSSION

1.0 BACKGROUND

The reason for constructing the multiprocessor system was to demonstrate that the possibility of obtaining a linear relationship between speedup and the number of processors in the system, predicted in chapter 2, is in fact obtainable in practise.

In order to verify this prediction a suitable bench mark had to be chosen. As the machine had been purposely built for field equations it was logical to use a field equation to test it with.

A Laplace solver was chosen as the bench mark, this would solve a square mesh problem which although seemingly trivial is nevertheless representative of other field equation types. The solution involves boundary values being defined on the edges of this square mesh and the values at nodes within the mesh being found by means of repeated iteration (as described in chapter 4) until Laplace's equation is satisfied. Various algorithms exist for the solution of Laplace's equation, some of which show more rapid convergence on a solution than the method which has been coded. In this case it is the relationship between speedup and the number of processors which is important, and not particularly the performance of the actual algorithm, in this respect the chosen algorithm is more than adequate.

The discussion of this chapter is sectioned as follows:

2.0 The processing array tests

3.0 The possibility of a system model

4.0 The Convergence tests

5.0 A comparison with the other machines

6.0 Alternative architectures offered by a PE of the multiprocessor

7.0 Possible future work.

Each of these sections will be discussed separately.

2.0 THE PROCESSING ARRAY TESTS

2.1 Initial results

The investigation of chapter 2 made clear that to avoid the multiprocessor system being severely degraded, it should be operated using the largest size of problem mesh possible. This would reduce the percentage of nodes at the boundary of a PE in relation to the total number of nodes within the PE and in so doing reduce the degradation caused by shared memory accesses, since these boundary nodes are stored in shared memory. The tests performed include problem sizes where the effects of such degradation can be seen clearly, in the initial tests for a problem mesh of 12x12 nodal points the 16PE array can be seen to be degraded a level of performance achievable with less than 5PEs (a 70% degradation). This degradation seen in figure 5.2 of the previous chapter is reproduced in figure 6.1, with this specific case of degradation highlighted.

This area of operation of the system in which degradations occur, can be seen in the three-dimensional plots of the results in chapter 5. For a

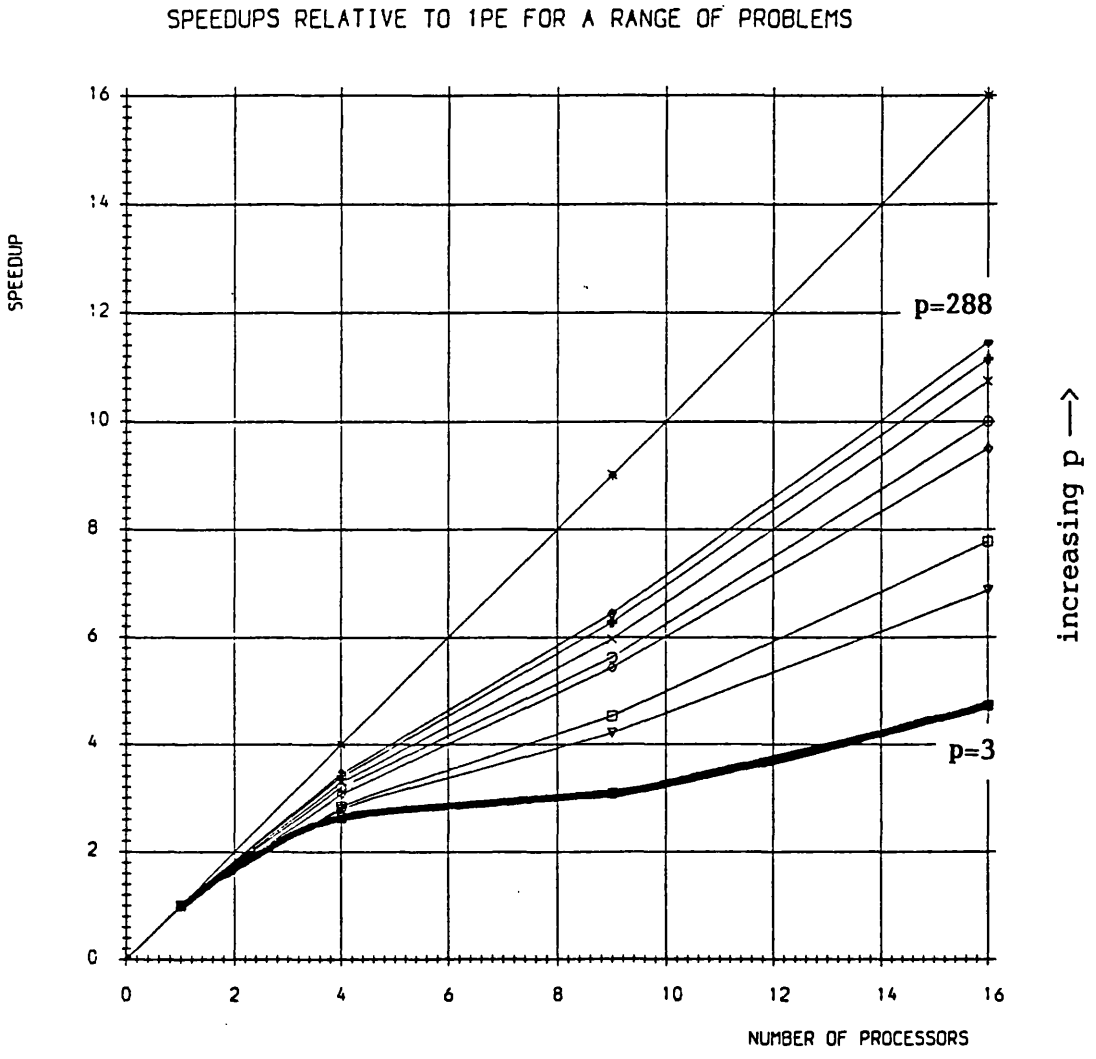


FIGURE 6.1 HIGHLIGHTED DEGRADATION

total problem size of p^2 nodes, this region of degradation can be seen to lie in the region where $p^2 < 20000$ (for 16 PEs). The severe degradations occur in the region where $p^2 < 5000$. In general this area of severe degradation can be avoided by ensuring that the number points on one edge of a PE boundary (p_L) is less than 6% of the total number of nodes which that PE contains. For a problem containing a total of p^2 points, then for a number of PEs (n):

$$\frac{n^{\frac{1}{2}}}{p} < 6\% \quad ; \qquad \text{since } p_L = p/(n^{\frac{1}{2}}) \ .$$

Thus, a criterion has been found which should help in deciding the number of PEs on which a problem will be run for the greatest benefit (i.e. choosing the number of PEs such that the region of severe degradation is avioded).

The area of degradation occurs in all the tests of the previous chapter, but is not itself discussed again as more important findings are presented.

Out of the region where severe degradation takes place the initial results for speedup show that the predicted linear speedup can indeed be achieved. However, the initial slope of the line of the speedup did not match that predicted in chapter 2, where for example in the case of 16PE a speedup figure of 15 was predicted, and a figure of 11.45 was being achieved (this can be seen in figure 6.2 which zooms in on figure 2.4(a) of chapter 2, in relation to the ideal and expected speedup figures). The cause of the degradation responsible for reducing the speedup figure to 72% of the ideal and 76% of the expected speeedup could only have been due to one or more of the following factors:

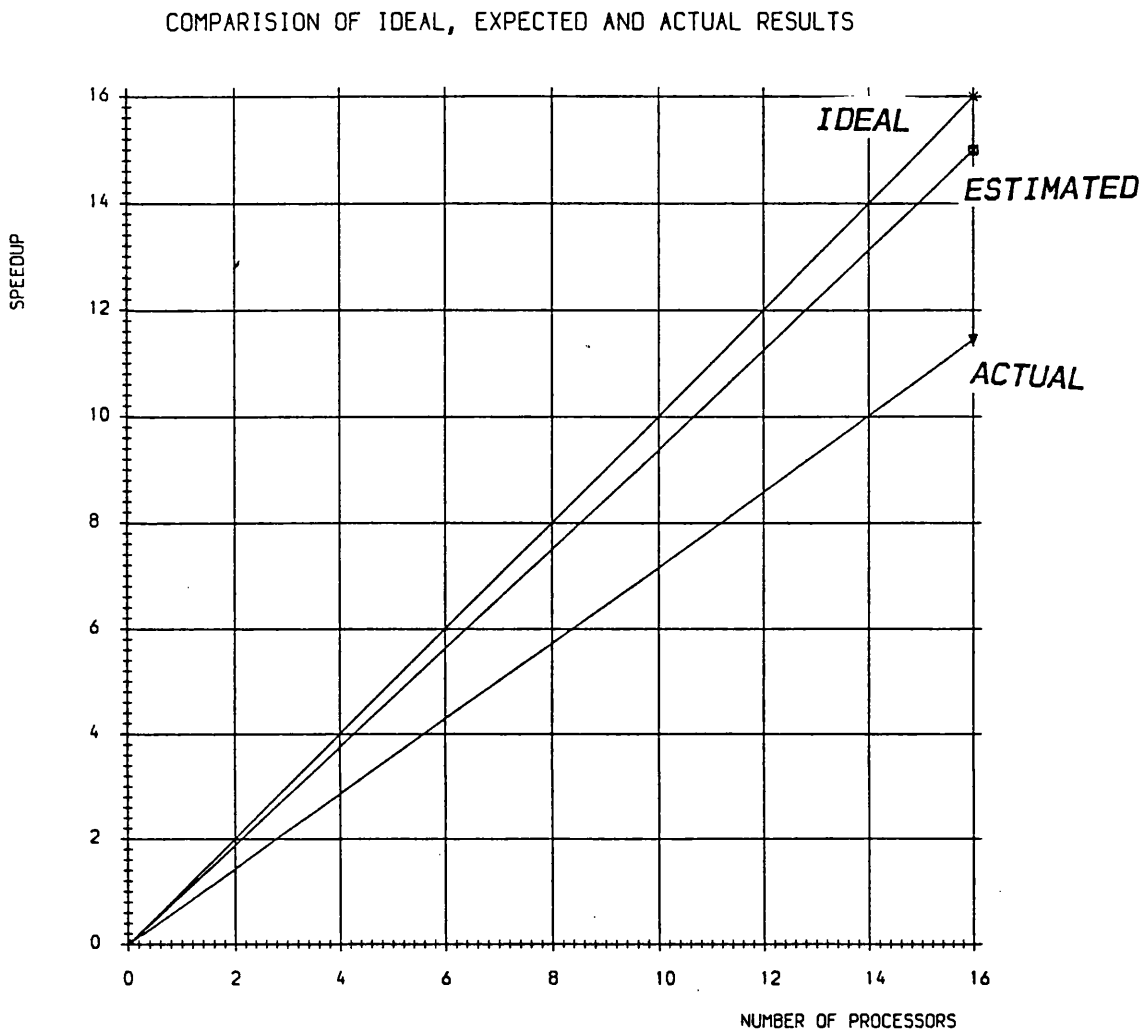


FIGURE 6.2 IDEAL, ESTIMATED AND ACTUAL PERFORMAMCE OF THE 16PE ARRAY

(a) More contention effects than expected, possibly due to the extra overhead of synchronization before a memory switch occurs;

(b) The way a problem is partitioned onto the array;

(c) Hidden overhead in the software being run;

(d) The initial prediction being wrong.

Each of these factors was assessed as to the likelihood of it being a major contribution to the degradation.

(a) Synchronization and contention problems

In order to check whether synchronization and contention effects were the (or a major) cause of the degradation witnessed, the worst case of synchronization and contention effects was considered for a given problem: When running a 288x288 mesh problem on the 16PE array each PE deals with a 72x72 array of nodes. Analysis of the algorithm reveals that, for such a problem size, a PE in the middle of the 4x4 array of PEs would make 1148 shared memory accesses in each pass of the mesh. If the maximum amount of contention possible occurred for every shared memory access (in this case $4x10^{-6}$s), then for five passes of the data 0.023s ($4x10^{-6}$x1148x5) would be lost due to contention. Since the 5 pass time over the nodes of a mesh this size is of the order of 286s, implying that the time lost due to contention and synchronization would be 0.008% of the time for 5 passes it was unlikely that synchronization and contention effects were the cause the problem.

To confirm the hypothesis that contention and synchronization effects were not the cause of the degradation witnessed, a 288x288 mesh of a Laplace

problem was mapped onto the 16PE array (72x72 nodes in each PE). A PE in the middle of the 16PE array took 4 min 45.95 seconds to do 5 passes of the data without any contentions at its boundary (the ability to run a program on one PE alone is a special function of the overseer program [described in chapter 4]), while the same problem took 4 min 46.69 seconds with active neighbours. The difference of 0.74s between the five pass times with and without contention and synchronization effects implies that the degradation seen in the system should be 0.26% (0.74/286.69), much less than the 28% witnessed in the results taken. The discrepancy (0.74s compared with 0.023s) between the theoretical (maximum) and the actual effects of contention is almost certainly due to inaccuracies in timing, being about 0.25% of the run time measured. Thus, these contention and synchronization effects are not seen to be a major cause of the degradation seen.

(b) Mapping of the problem onto the array

An ideal mapping of a problem mesh onto an array of PE would result in each PE taking the same time to process the nodes it had been allocated, this would imply that each PE was doing the same amount of processing work. Partitioning of the problem between PEs is achieved by allocating sections of the mesh (each with an equal number of nodes), to a PE whose position in the PE array is analogous to the position of the section of the problem mesh with respect to the whole problem mesh. The processing time of the nodes within each PE depends on the PEs position within the PE array, since PEs at the boundary of the PE array contain a percentage of the problem boundary nodes which do not need processing. The time taken to process the other (active) nodes it contains is faster than for PEs which do not contain any (or as many) boundary nodes of the problem. It can be seen therefore that mid placed PEs take longer to pass over their respective data mesh than the outermost PEs.

The speedup figures recorded are based on the average pass times of a given size of problem on 1PE compared with that of the same problem run on a number of PEs. However, because all the timings are based on "the last PE to finish" the time recorded for a problem size where a PE in the centre of the PE array was dealing with 72x72 nodes, would effectively be that of a problem with a total of 82944 (72x72x16) active nodes i.e. a total problem size of 290x290 nodes, and not the 280x280 problem from which 72x72 active nodes are mapped onto such a PE. The severe degradation witnessed in the initial results may not actually exist, but may be a consequence of the way the problem is mapped onto the PE array and the way speedup figures are calculated. If this was the case then the average pass times for a 74x74 mesh problem running on 1PE should be comparable with that of a 288x288 mesh problem running on 16PEs, since the 1PE system and one component PE of the 16PE array have the same number (72x72) of 'active' nodes.

The average pass time for the 1PE case (with a 74x74 node mesh) is 35.4s (extrapolated from the equation given in chapter 5) and 57.3s in the 16PE case. These times are so different they eliminate mapping as the main cause of the witnessed degradation, while indicating that overheads in the software may be responsible for the degradation.

(c) Hidden overheads in the software

The average pass times compared in the previous section for two different cases, each of which ran the same software, showed a great discrepancy in times where in theory none should exist. The software being run has a procedure (called the 'get_address' procedure) used to determine the correct place in memory for the data to be accessed. The get_address procedure (seen in program 4) is frequently accessed (each time data from

the problem mesh is required), so any problem which exists here would be significant. The differences in the times seen in the previous section must have been the result of different times taken to execute the PE dependent sections of this procedure. In order for the degradation to be avoided the code had to be improved such that the time taken to run problems with the same number of active nodes, was made similar for each number of PEs the program may run on, making sure that the time taken to execute the program on a given architecture was not reduced. The updating of the get_address procedure involved :

1. Byte sizing of variables as they enter the procedure to cut down the time taken for any comparisons;
2. Improved structuring the procedure, so as to minimize the time taken for an address calculation for the majority case of a node located within a PEs local memory;
3. Optimizing the source code such that the one pass PL9 compiler will produce efficient object code.

The result of these modifications was the get_address procedure listed as program 4, and led to the execution time for a 288x288 problem mesh on 16PE being reduced from 57.3s to 41.6s (approximately a 25% reduction in run time), and to the difference between the pass times of 1PE dealing with a 74x74 mesh and a component PE of the 16PE array dealing with a 72x72 mesh (i.e. the same number of active - non boundary - nodes) being reduced from 62% to 17%.

This was obviously the cause of the degradation from the predicted value. The test pin-pointed the cause of the problem enabling very significant improvements to be made.

(d) Initial prediction being wrong

Although the cause of the degradation has been established it is useful to note that the prediction of the speedup was based on contention effects, the amount of global communication and the amount of time spent in, or preparing to use, shared memory. The results of the prediction are the results of a simple probability model but are thought nevertheless to be correct, and have been shown to be close to the actual performance of the multiprocessor with the faster get_address procedure.

2.2 Final results

The improvement provided by the faster get_address procedure are substantial. Linear speedups are still obtained but with a gradient against the number of processors approaching that of the ideal. With the 16PE system tackling a 288x288 problem mesh the system runs at 88% of the theoretical maximum (compared with 71% in the initial case and basing speedups on one PE of the array). If the speedup figures are based on those of an independent processor (the MICROBOX II) the system performs at 73.9% (11.83/16x100) of the theoretical maximum. This performance is still not ideal and could possibly be improved by further optimization of the code of the get_address procedure, which may involve writing the routine in assembler.

The fact that linear speedup has been achieved is enough to justify a multiprocessor system, the cost effectiveness of which is discussed later. The linearity of speedup with the number of PEs displays shows no sign at all of falling of with an increased number of PEs, as many of the multiprocessor architectures seen in chapter 1 would. With ever increasing performance

for increasing numbers of PEs there is no limit to the performance that multiprocessor systems can offer with the correctly chosen architecture for the problem.

3.0 SYSTEM MODEL

A system model capable of predicting the speedup figures for a given number of processors handling a given problem would be a very useful design tool, in that predictions of speedups could be made for numbers of processor larger than in the 16PE system here and thus help to determine the feasibility of a potential systems.

Since such a model is based on a particular type of architecture, as long as this architecture is adhered to, the speedup figure predicted would apply to a similar system based on any processor. If the speedup being predicted, for a given size of problem on a given number of PEs, could be made relative to a base machine, a relative performance model could then be created. It would then be possible for a given performance requirement and fixed number of PEs to find an appropriate processor for use as the heart of a PE for that requirement or alternatively, for a fixed performance requirement and processor (on which the system is based) for the number of PEs required to achieve that requirement to be found.

It is useful to investigate whether it is possible to construct a system model that would enable speedup to be calculated, knowing the number of processors in the system and the size of the problem mesh. In the previous chapter for each type of the allowable number of PEs upon which a problem may be run a unique equation capable of predicting this time, based on the size of the problem mesh has been found. The general form of these equations is:

$$t = \frac{p^a}{b}$$

where t = time taken

p = number of points on one side of problem mesh

a = constant for a given architecture

b = constant for a given architecture

To obtain a system model the values of 'a' and 'b' must become functions of the number of processors in the system. Collating the data available (from the initial results of chapter 5) gives:

number of PE	a	b
1	2.08	296.4
4	2.05	658.8
9	2.00	1123.6
16	1.97	1763.3

TABLE 6.1 Constants for prediction

Following the success of the method by which the equations used in the extrapolation where found (through chapter 5) the method is repeated and, the data of table 6.1 can be seen plotted logarithmically in figures 6.3 and 6.4. The speedup (SU) figures within a system model can now be expressed as:

$$SU = \frac{p^{power}}{constant}$$

where $power = \frac{no\_of\_processing\_elements^{-0.02}}{0.48}$

and $constant = no\_of\_processing\_elements^{0.64} x 229.21$

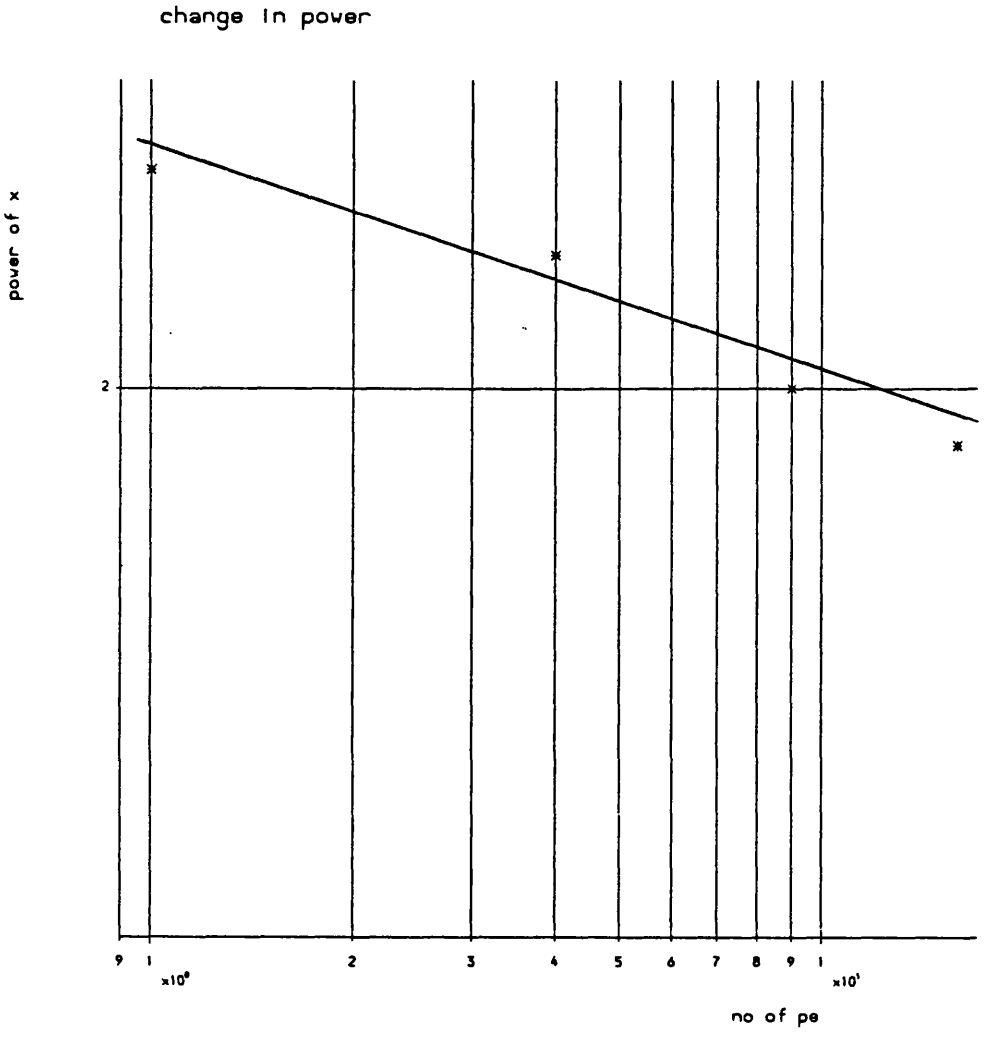


FIGURE 6.3

LOGARITHMIC RELATIONSHIP OF THE POWER (A) TO THE NUMBER OF PES

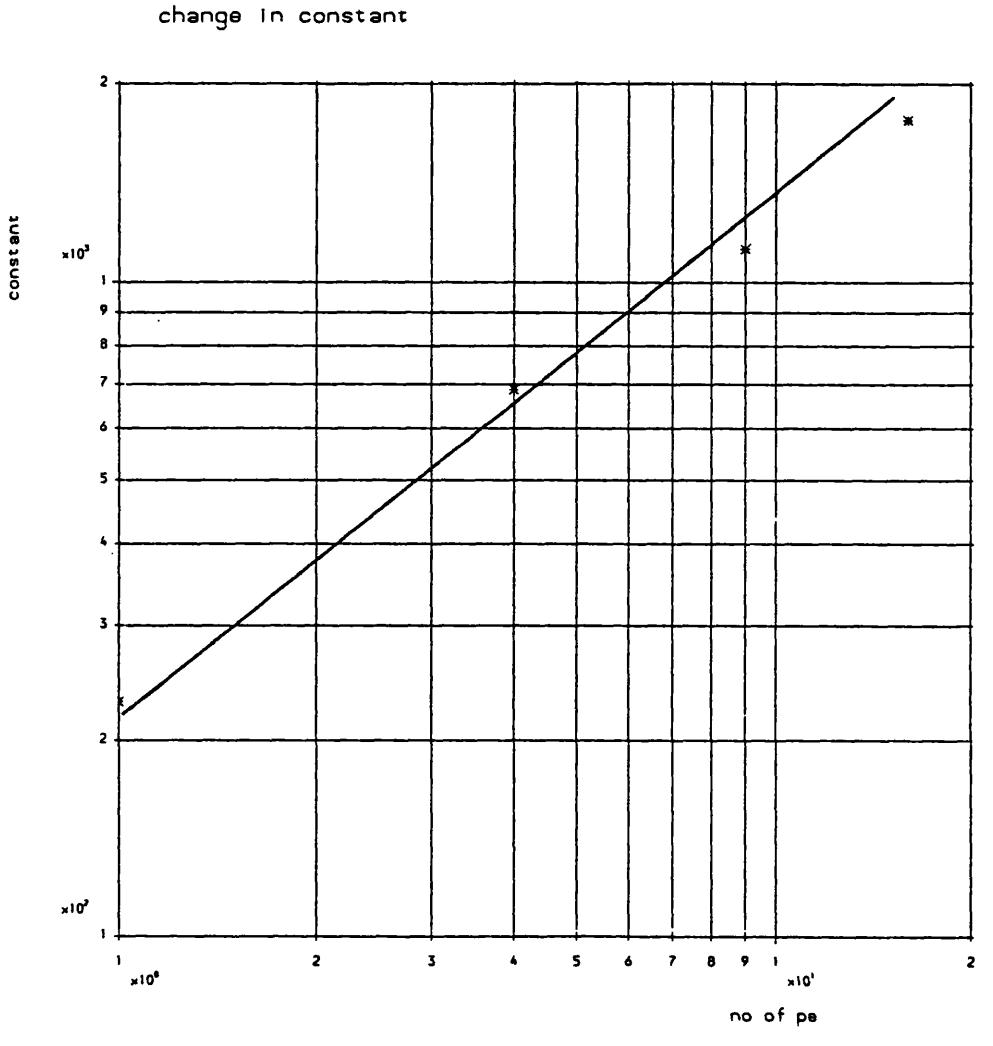


FIGURE 6.4

LOGARITHMIC RELATIONSHIP OF THE CONSTANT (B) TO THE NUMBER OF PES

The tabulated output for speedups, and the values of 'a' and 'b' being used for this set of equations can be seen as the output of the program "PRED" (microfiche listing 3). The three-dimensional graph of speedup as a function of the number of processors and problem size predicted by the system model can be seen in figure 6.5 to have a shape similar to that of the actual system. When comparing the actual speedups of the initial results in chapter 5 with those predicted in appendix K, the magnitude of the speedup figures being predicted can be seen for small problem sizes to be up to 64% in error (in the case where p=12 on 16PE). For larger problem sizes the error is reduced to -5%, making it possible for the present system model to be used to estimate a lower bound of speedup for any given system, but with questionable accuracy.

The error is due to the inaccuracy of the assumed linear relationship between the value of 'a' and the number of processors, this can clearly be seen in figure 6.3 to be non linear, but in the absence of a better relationship a linear approximation had to be made. If a better relationship could be found then it would be possible to enhance the predictive nature of the system model to that of the relative performance model suggested previously, where in the ideal case:

$$RP = \frac{PPE}{PBS} \times NPE$$

where RP = The relative performance of the system to that of the base system
PPE = The performance of a processing element relative to a reference system
PBS = The performance of the base system relative to the reference system
NPE = The number of processing elements.

For the results obtained in chapter 5 for problems with a relatively large number of nodes (p) in the problem mesh, the results were near enough to

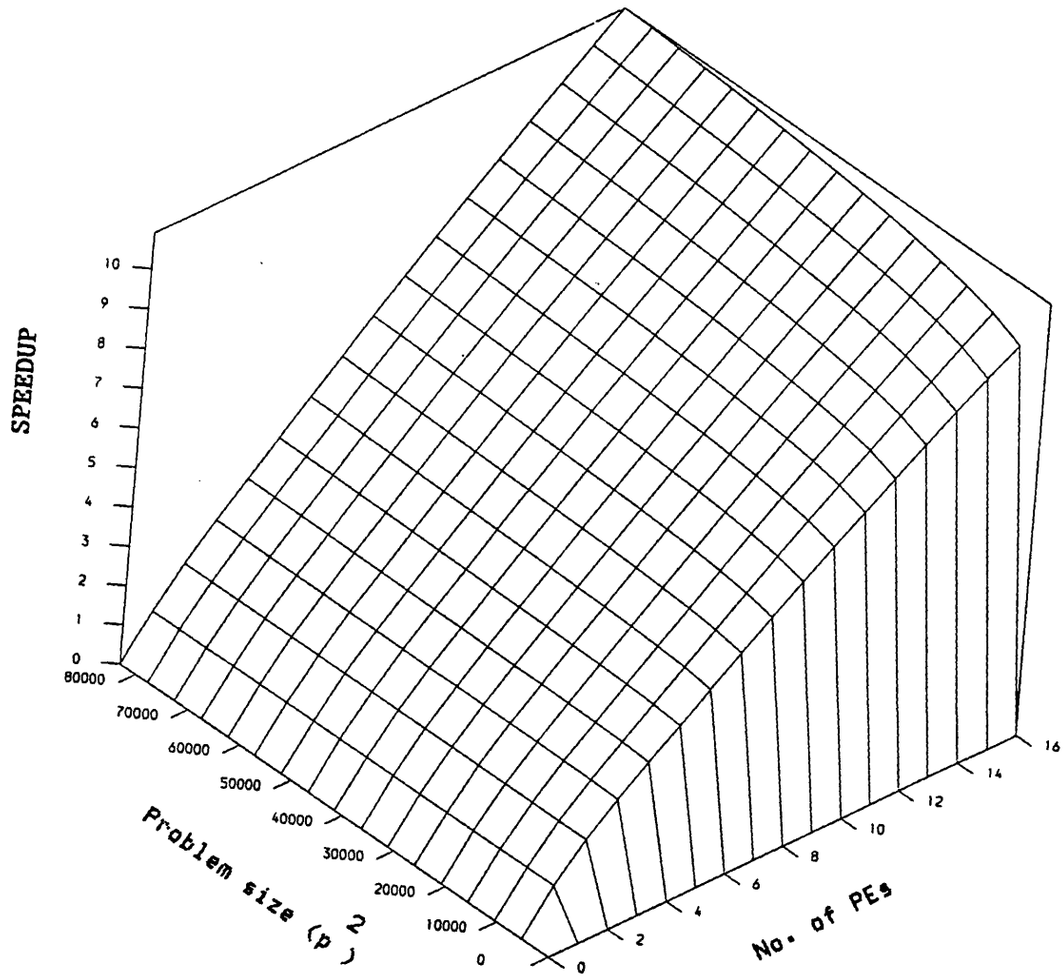


FIGURE 6.5 PREDICTION BY THE SYSTEM MODEL SHOWING SPEEDUP AS A FUNCTION OF PROBLEM SIZE AN NUMBER OF PES

the ideal for the above relationship to be used in predictions for other systems. It was stated that, for the machine to operate without severe degradation the problem sizes should be kept as large as possible. For small mesh problems a system model more accurate than the one presented here must be found; this would enable the number of PEs(NPE) in the above relative performance model to be adjusted accordingly, to the degradation which can result from this. This is, however, not likely to be of much interest; in practise the machines are only designed for problems with large numbers of processing elements and the solution of large problems.

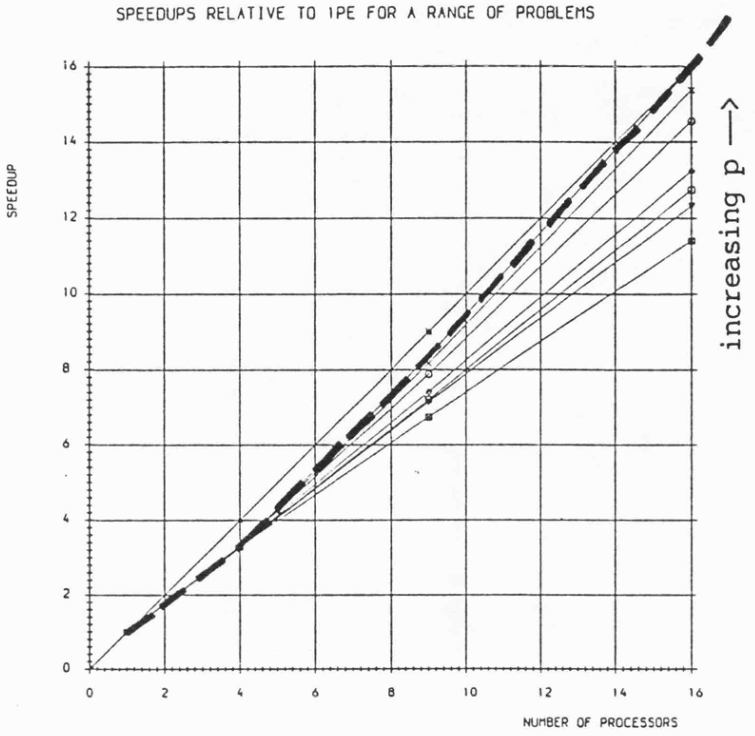
4.0 THE CONVERGENCE TESTS

One of the objectives of the investigation was to examine the effects a multiprocessor solution of a problem would have on the stability of the equations being solved. Stability is a problem in more complex field equations such as the Navier-Stokes equations of fluid flow, and if it could be improved it would inevitably lead to faster solution times for a given problem or more accurate results. The stability of field equations to a great extent governs the rate at which (the correct) values appear at the nodes of the particular problem mesh in question.

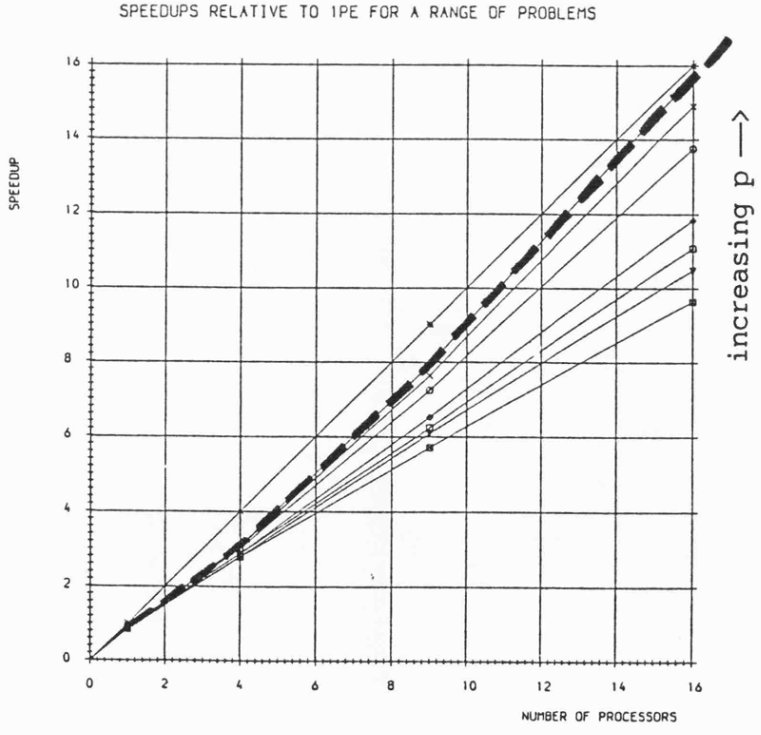
The speedup figures obtained in chapter 5 for the convergence tests were much improved over those of the set operations test. In the case of speedups based on the performance of a component PE of the multiprocessor, speedups within 0.1% of the ideal are achieved (in the case where p=288 on 16PEs) - where the ideal is that of the increase in the potential processing power of the system over that of a component PE. This is only slightly degraded (to within 2% of the ideal) when comparing with the performance of an independent 6809E system (the MICROBOX II) for the reason given in chapter 5 and highlighted again below.

The results of the problem convergence tests of chapter 5 suggest strongly that a speedup greater than that of the 'ideal' value can be achieved for large problem sizes, when basing the ideal on the increase in pure processing power over a component PE. Figure 6.6 reproduces figures 5.11 and 5.12 from chapter 5 highlighting this trend. Errors in results are not considered to be responsible for the findings since run times of up to 50 hours and 20 minutes duration were, where possible, used to verify the extrapolating equations (see appendix H). In an attempt to explain how results better than the ideal could conceivably occur consider the following:

In theory the time taken to pass over the two-dimensional data, of the bench mark Laplace solver, should vary linearly with the number of points (p^2). This has been seen to be true, in the logarithmic plots of the processing array tests of chapter 5, with small deviations due to the processing using the boundary nodes. If p^2 refers to the number of points in a problem on a single PE, then if the same problem is run on 16PE the time taken to pass over all the points on one PE would vary with $(p/4)^2$, giving a theoretical maximum speedup figure for the 16PE system of 16 ($p^2/(p/4)^2$). Similar thinking to the above when calculating the speedup when considering problem convergence would suggest that the rate at which values are propagated across the two-dimensional mesh in one PE, should vary not only with the number of points but also the time taken to pass over them i.e. to vary with p^4. The variation with p^4 can be seen in the logarithmic plots of the problem convergence tests of the previous chapter, again with slight deviations being caused by boundary node eccentricities. The calculation of the maximum theoretical speedup is not as simple as the above calculation since values propagating across the inner PEs must first have propagated through the outer PEs.



(A)



(B)

FIGURE 6.6 THE TRENDS IN SPEEDUP OF THE MULTIPROCESSOR

The maximum theoretical speedup in the processing power has been found above. When looking at the way in which the PE array performs the field calculations as opposed to the number of calculations it can perform in a given time, a new maximum emerges. The maximum speedup in the processing power must be increased to take into consideration the fact that the multiprocessor system provides a two-dimensional pipeline. The pipeline effectively consists of n/2 stages (where n^2 is the number of PE in the system) through which values propagate towards a focus at the center of the problem mesh (at the centre of the PE array). The following relationship for speedup (S) should then hold (for large p):

$$S \alpha \frac{n^3}{2}$$

(where $n^3/2 = n^2 \times n/2$).

The hypothesis in chapter 5 that:

$$\frac{\partial S}{\partial n} \alpha n^c ,$$

(where c is a constant) is born out by the above since:

$$\frac{\partial S}{\partial n} \alpha n^2 ,$$

which would fully describe the trends indicated in figure 6.6. The value of 'c' should not be taken directly from this relationship since boundary value eccentricities have not been taken into consideration.

Operation of the PE array indicates the existance of the trends seen in figure 6.6, but currently the system is unable to perform in the region which may be viewed as better than the 'ideal'. If the degradation in the array can be reduced (for example by further improvements in the get_address procedure) or the PE array extended to form a 5x5 array, then proof that performances better than the 'ideal' would almost certainly be

obtained. If this relationship can be proven, then such multiprocessor systems offer the remarkable prospect of providing a greater benefit than the architecture of the machine would at first suggest.

5.0 COMPARISON WITH OTHER SYSTEMS

5.1 Performance on the bench mark program

The bench mark Laplace algorithm was implemented on several systems, each of which was given the problem which gave the best performance in the 16PE array (that of the 288x288 mesh). Where it was not possible to implement the mesh directly an approximation (as indicated) was used. Systems other than the 16PE array did not need the get_address procedure and therefore did not have the associated overhead of this code. Where possible programs were compiled with a "no-optimize" switch (this applies to the PRIME and VAX systems) to compensate for the use of a one pass compiler in PL9 coded systems.

Running the bench mark in a range of machines (program 5 [for 6809E systems], program 2 for the MICROBOX II, program "Laplace" (on microfiche) [for VAX and PRIME systems] and program 1 [for the OLIVETTI]), the best improvement over the 16PE array was found to be the VAX 8600 which was a factor of 77.2 faster.

The relative performance figure for the VAX 780 was 17.6 times that of the 16PE system. A preliminary estimate of the performance of the 16PE array included in chapter 2 put this figure at 0.6. This discrepancy between the estimated figure and the actual figure shows the danger of accepting manufacturers bench marks for small microcomputers, and mixing them with what appear to be similar bench marks for mainframe systems. Consequently

rather than the 16PE system operating with a performance in excess of that of a VAX 780 and much higher than the band in which commercial machines can be seen (in chapter 2) to lie for the same price, the machine operates just above this band and with a performance 17.6 times less than that of a VAX 780 (figure 5.13 in chapter 5). Nevertheless it has been shown that a multiprocessor system can be made to operate outside the normal band of operation of commercially available machines in an area which should be of commercial interest.

5.2 Performance/costs of the above systems

It is only when comparing the performance of each machine with respect to its cost that the true worth of the 16PE multiprocessor system emerges. In terms of the performance/cost ratio the 16PE array, which can be seen in figure 5.14 in chapter 5, the multiprocessor is 2.5 times better than the VAX 8600, and 3 times better than the MICROBOX II.

The consequence of an architecture which displays (at worst) a constant performance/cost ratio is discussed below. In general, all systems which show increasing performance for increasing numbers of PEs can in theory achieve any desired performance simply by incorporating enough PEs. In practise however it is the rate of speedup (S) with the number of PE (n) which renders a system impractical if:

$$\frac{\partial^2 S}{\partial n^2} < 0 \quad ;$$

since this implies that the increase in performance gained by the addition of another PE is not as much as for the previous PE, which for large numbers of PE renders the system uneconomical. Due to the (at worst)

linear relationship between SU and n displayed throughout the previous chapter where it can be seen that for large problem sizes:

$$\frac{\partial^2 S}{\partial n^2} > 0 \quad ;$$

every added PE delivers (at least) the same increase in performance, in such a system the performance is not limited by diminishing returns.

The performance envelope of the MC6809E based system with cost can be seen in figure 6.7 where the bandwidth of performance for commercially available machines (shown by hatching). Figure 6.7 also shows the performance envelopes of similarly architectured machines based on more powerful PEs, which can achieve the same performance levels as the MC6809E based system but with far fewer PEs.

5.3 Possible competition to a multiprocessor system

Systems which display high performance for their cost such as the APOLLO and SUN workstations, the micro VAX systems (Malone [58]) all at around $30,000 and the promise of the ACORN Restricted Instruction Set Computer, may in the short term offer competition to multiprocessor systems comprising of few PEs. However, if a faster and possibly cheaper processor does appear there is no reason why these processors cannot become the heart of a multiprocessor system, similar in architecture to the one which has shown so much success for this particular application, and thus out perform the systems in which the processor was first launched. It should also be remembered that a multiprocessor system is modular and therefore its power can be increased by addition of more units (PEs), and since its performance is linear with the number of PEs its performance is in theory unlimited.

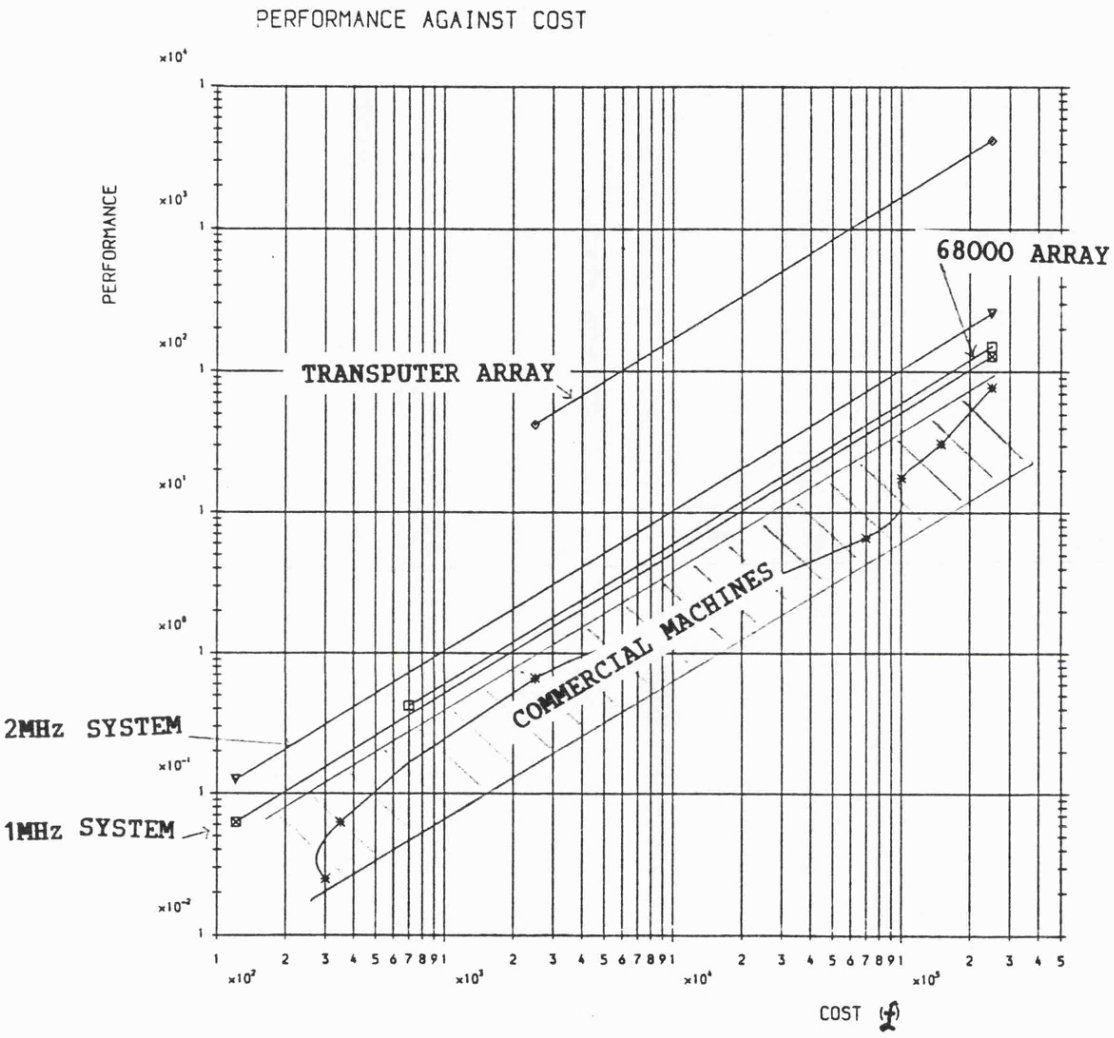


FIGURE 6.7 PERFORMANCE ENVELOPES OF MULTIPROCESSOR MACHINES

6.0 A MORE GENERAL SYSTEM

Although the architecture of the 16PE multiprocessor array has been specifically designed to optimize the solution of field equations, it is possible that other uses for the system may be found.

6.1 Alternative architectures offered by a PE of the multiprocessor

Although the PEs have been arranged as a 4x4 array there is no reason why they should not be arranged differently, since this can be achieved simply within the same rack which presently holds the system by altering the connections between PEs. New topologies may not involve all the connections a PE can offer, or even all the PEs, enabling architectures to be tailored to a specific problem. For example, in the case of a field equation problem where a rectangular mesh (rather than a square mesh) would be more useful, a PE array of 3x5, 2x8 or even 1x16 PEs could conceivably be formed.

7.0 FUTURE WORK

7.1 Further optimization of the 16PE array

Further optimization of the 16PE array routines may be carried out, as suggested earlier in this chapter, to bring the system nearer to the ideal performance than has been achieved to date. If this can be performed, proof of the hypothesis, that the can system perform better than would be expected by the increased processing power of the system, may be found, while improving the performance in general of the system with respect to other machines.

7.2 A 2MHZ version

A 2MHz version of the 16PE multiprocessor is directly available on the current system by simply updating the processor in the system to the MC68B09E and the XTAL package to 32MHz. The performance of this system can be seen in relation to those of chapter 5 and the 16PE (1MHz) version in figure 6.7. This shows that a 2MHz version would be out of the main stream of available machines (shown by hatching) with a performance twice that of the 1MHz system and not usually attainable at this cost. In practise however noise problems could cause difficulties and a 2MHz system may have to implement the changes of the following section, if noise free operation is to be guaranteed.

7.3 System redesign

The need for discrete components in the contention circuitry is no longer necessary since debugging of the 16PE system has been completed. Placing the contention circuitry and the interrupt mechanism (where possible) into fast Programmable Read Only Memory (PROM), would considerably reduce the number of components in a PE. The use of PROMs and multi-layer printed circuit board technology would reduce the size of a 16PE system to desk top proportions. The present method of inter-board connections would have to be dropped in favour of a mother board, which may also serve as the master processor for the system. The benefit of such a system would be seen to be high processing power, compactness, while offering noise suppression, ease of expansion and the possibility of commercial production.

7.4 More powerful processors

Retaining the same architecture a system with a more powerful PE would enable a given performance level to be achieved with fewer PEs. It is envisaged that a more powerful system would adopt the system redesign suggested above, to reduce the unit cost of a PE for a system with a large number of PEs; two examples of systems based on more powerful processors can be seen in the sections below.

(a) A Molorola 68000

This would need the least redesign of the system currently in operation due to the similarities between the processors. A 16PE machine based on the MC68000 processor, would show an expected speedup factor of 5 over the 16PE (MC6809E). With a floating point coprocessor this figure would be expected to increase to about 6.75 (based on the performance of a comparable 8086 system - the Olivetti M24). The extra cost of the processors, faster memory, redesigned components and the new printed circuit boards put the price of a PE in this array at around £700. The expected performance envelope of a system based on this processor can be seen in figure 6.7 to be almost that of the 1MHz 6809E system, but it must be remembered that the same performance levels are being achieved with less PEs. Using the formula of section 3.0 a 5x5 array of MC68000 based PEs would provide a performance similar to that of a VAX 780 (PPE=0.6628, PBS=17.6 and NPE=25) for a price only one third that of the VAX system.

(b) A Transputer (the T800)

The Transputer architecture which would ideally suit the field equation problem would be that of an n x n array, with local communication in shared

memory and global communication by means of transputer links. The INMOS transputer (the IMS T800) has floating point capability built into the chip. The expected improvement of a Transputer PE over a PE based on the MC6809E microprocessor (based on INMOS figures) is estimated to be be 675 times better. With an expected cost per PE of £2500 the range of operation of this system can be seen in figure 6.7, where the high performance and relatively low cost of the potential PE put the performance of the transputer array in an area of great potential commercial interest.

Again there is no reason why the Transputer system would not follow the model suggested in section 3.0. This would imply that one PE based on a Transputer would out perform the VAX 780 by a factor of 2.39 (PPE=675/16, PBS=17.6, NPE=1) on the given problem (it must however be remembered that the VAX is a more general machine). A 2x2 array of transputers would out perform a VAX 8600 by a factor of 2.18 (PPE=675/16, PBS=77.2, NPE=4) for a cost only 4% that of the VAX system. An array of 13x13 transputer based processing elements, arranged with an architecture similar to the prototype in this work, is expected to outperform the CRAY II and to do so at a cost of only 8½% that of the CRAY system[1].

---

[1] Estimated from figures obtained from Perrenod [65] and this work.

CHAPTER 7

CHAPTER 7

THE CONCLUSIONS

A probability model of a multiprocessor architecture for the solution of field equations revealed that if communication between processing elements within the architecture was restricted to nearest neighbour, then linear speedup of processing power with the number of processing elements would be seen. Specifically for an architecture comprising of 16 such processing elements a performance 15 times greater than that of a single processing element is predicted.

Linear speedup is a major achievement in a parallel processing system, as it implies the degradations usually seen in such a system have been overcome. Performance has been seen to vary linearly with the number of processing elements in a multiprocessor when running the field problems for which the architecture had been designed. The gradient of this obtained linearity was increased in the course of the work by 23% to a value 74% of the ideal when compared to an independent system. This corresponds to a performance 14.1 times greater than that of a component processing element (i.e. 94% of the value predicted). Suggestions for improving the linearity further are made in the discussion section of this thesis.

Analysing the performance of the multiprocessor architecture leads to the hypothesis that benefits greater than those expected from the increase in pure processing power of the multiprocessor are to be gained due to the pipelining effect which the architecture produces. Clear indications of the validity of this hypothesis can be seen in the results, the implications of

which are discussed in this thesis. It is expected that proof of the hypothesis would be found if the architecture was extended to form a 5x5 array of processing elements or the degradation from the ideal gradient of linear speedup improved, as suggested.

The success of the architecture in fulfilling the objectives of this work lead to the formation of relative performance model. From this model the performances of similarly architectured machines were found for varying numbers of processing elements based on different processors. This model is used to predict that an array of 13 x 13 processing elements based on the INMOS transputer, would out perform a CRAY II supercomputer and do so at a cost only 8½% that of the CRAY system.

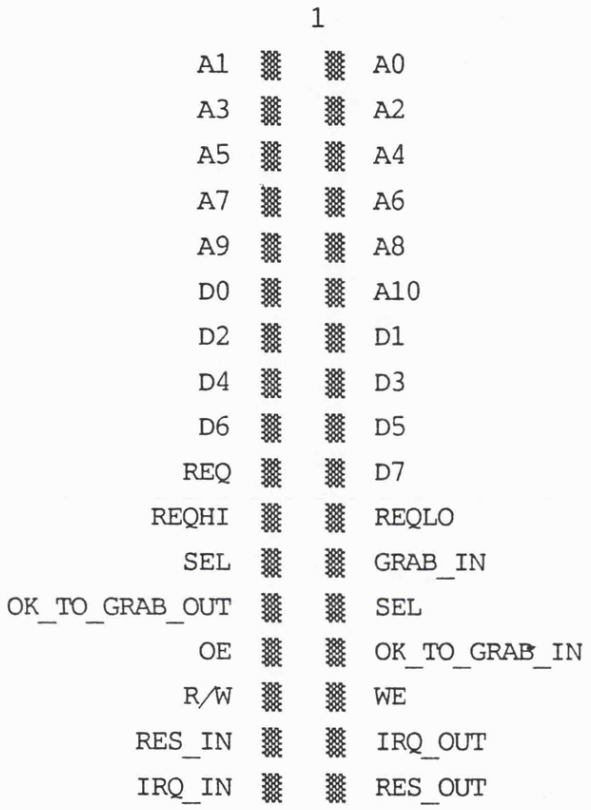
APPENDICES

APPENDIX A

PIN CONNECTIONS FOR THE SHARED MEMORY PORTS OF A PROCESSING ELEMENT

SHARED MEMORY CONNECTORS

SHARED MEMORY ON THE PCB



EAST AND SOUTH

SHARED MEMORY OFF THE PCB

1

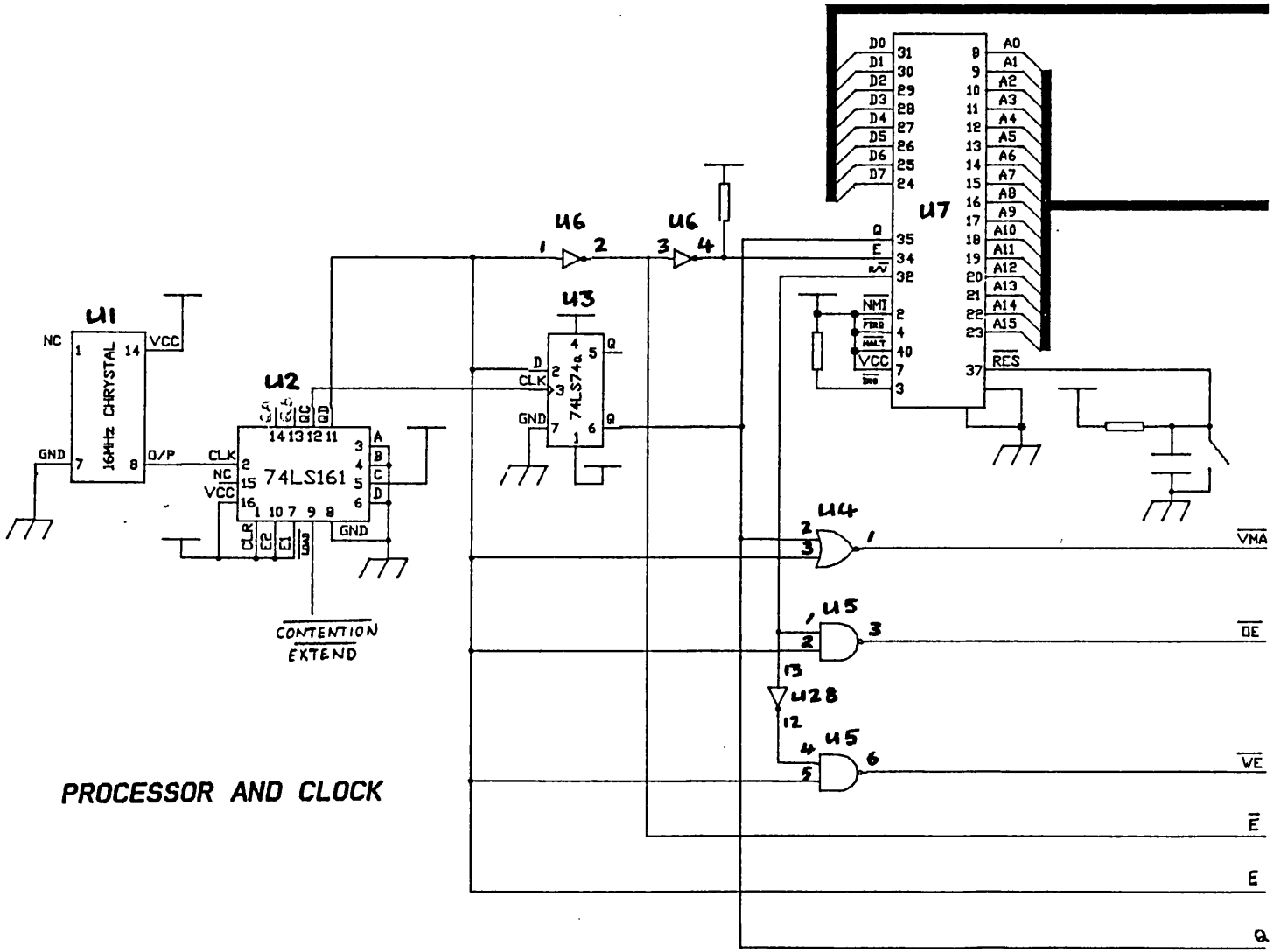
A1	▒	▒	A0
A3	▒	▒	A2
A5	▒	▒	A4
A7	▒	▒	A6
A9	▒	▒	A8
D0	▒	▒	A10
D2	▒	▒	D1
D4	▒	▒	D3
D6	▒	▒	D5
REQ	▒	▒	D7
REQHI	▒	▒	REQLO
SEL	▒	▒	GRAB_OUT
OK_TO_GRAB_IN	▒	▒	SEL
OE	▒	▒	OK_TO_GRAB_OUT
R/W	▒	▒	WE
RES_OUT	▒	▒	IRQ_IN
IRQ_OUT	▒	▒	RES_IN

NORTH AND WEST

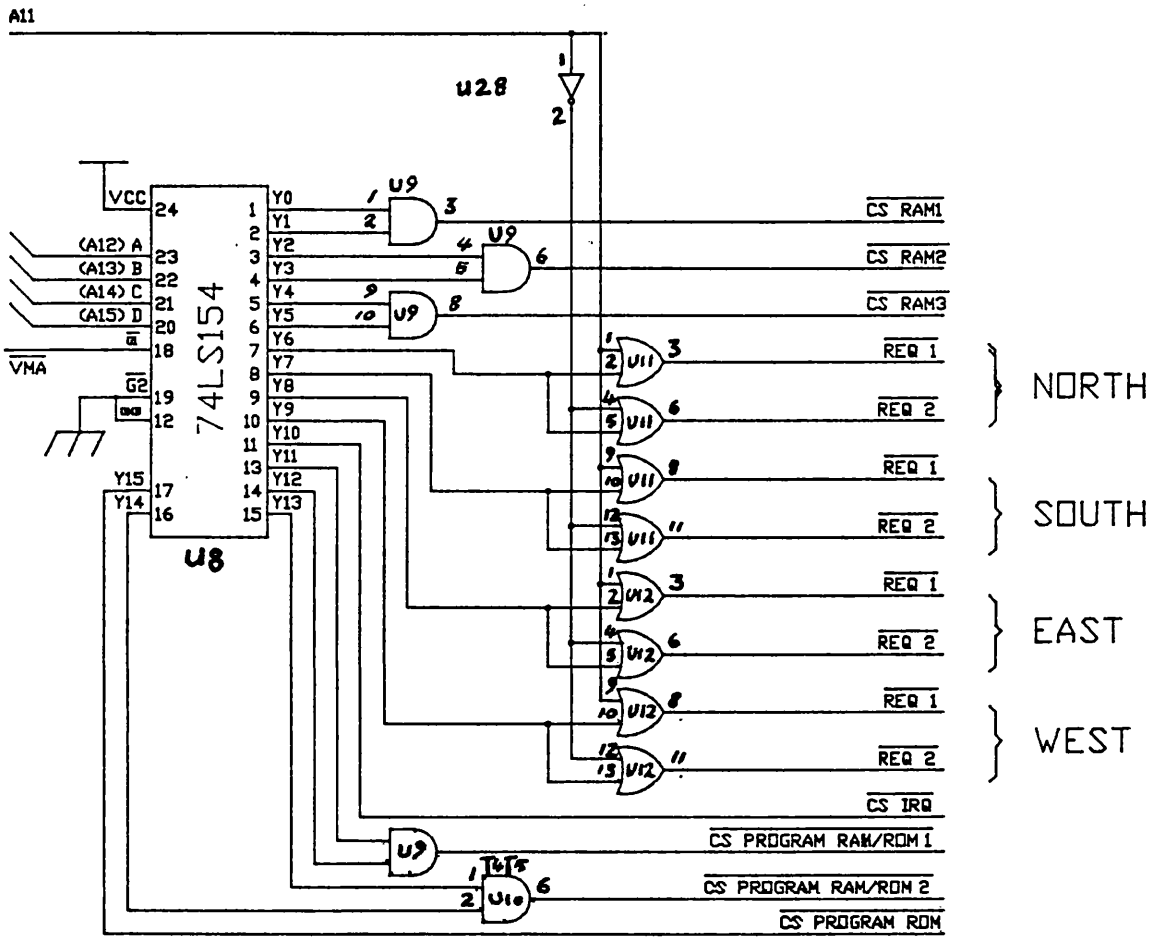
APPENDIX B

THE PROCESSING ELEMENT BOARD

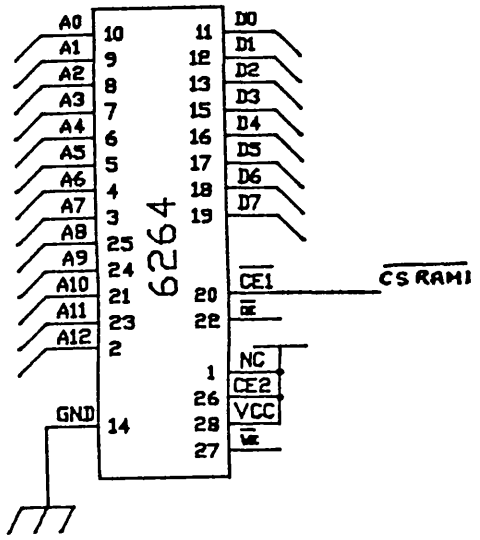
CIRCUIT DIAGRAMS

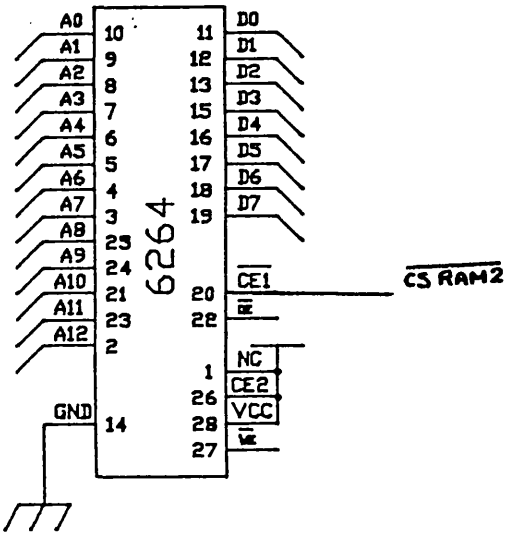


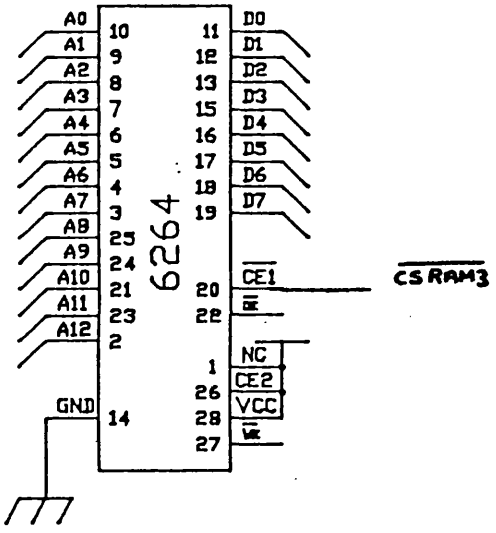
PROCESSOR AND CLOCK

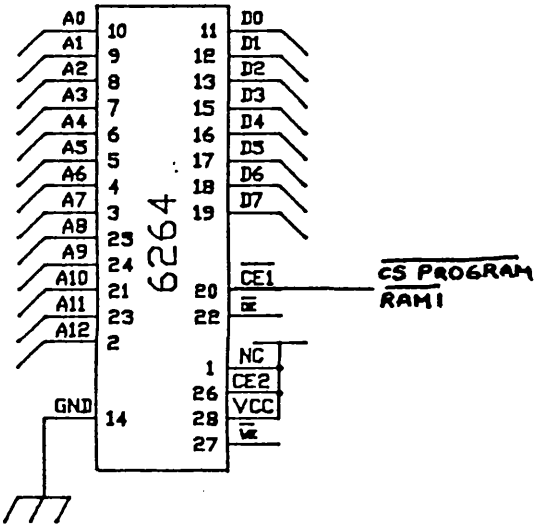

A11
U28
VCC
(A12) A
(A13) B
(A14) C
(A15) D
VMA
G2
Y15
Y14
74LS154
U8
U9
U11
U12
U10
CS RAM1
CS RAM2
CS RAM3
REQ 1
REQ 2
NORTH
SOUTH
EAST
WEST
CS IRQ
CS PROGRAM RAM/ROM 1
CS PROGRAM RAM/ROM 2
CS PROGRAM ROM

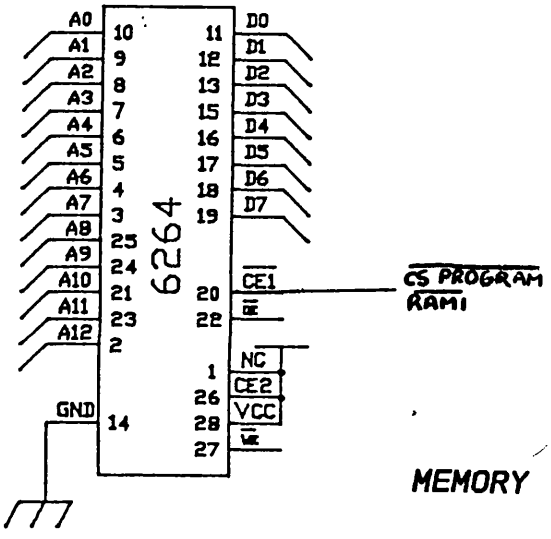
DECODING

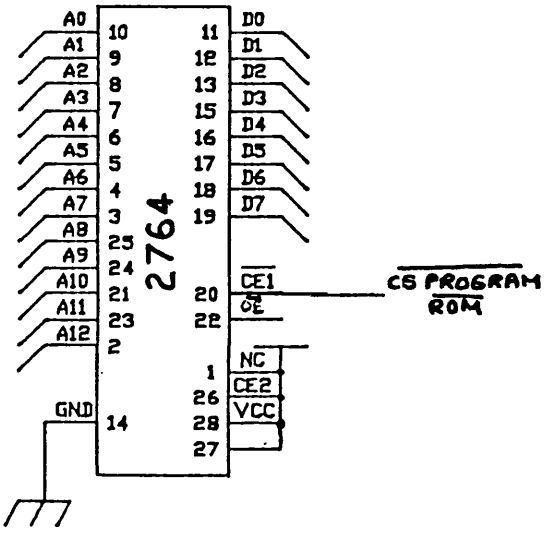

6264
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
D0 D1 D2 D3 D4 D5 D6 D7
CE1
OE
NC
CE2
VCC
WE
GND
CS RAM1


6264
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
D0 D1 D2 D3 D4 D5 D6 D7
CE1
OE
NC
CE2
VCC
WE
GND
CS RAM2

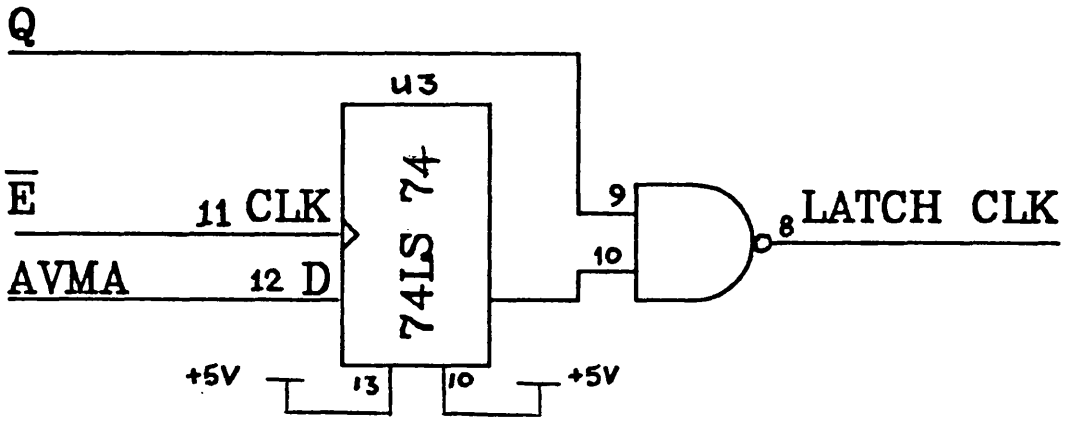

6264
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
D0 D1 D2 D3 D4 D5 D6 D7
CE1
OE
NC
CE2
VCC
WE
GND
CS RAM3


6264
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
D0 D1 D2 D3 D4 D5 D6 D7
CE1
OE
NC
CE2
VCC
WE
GND
CS PROGRAM RAM1

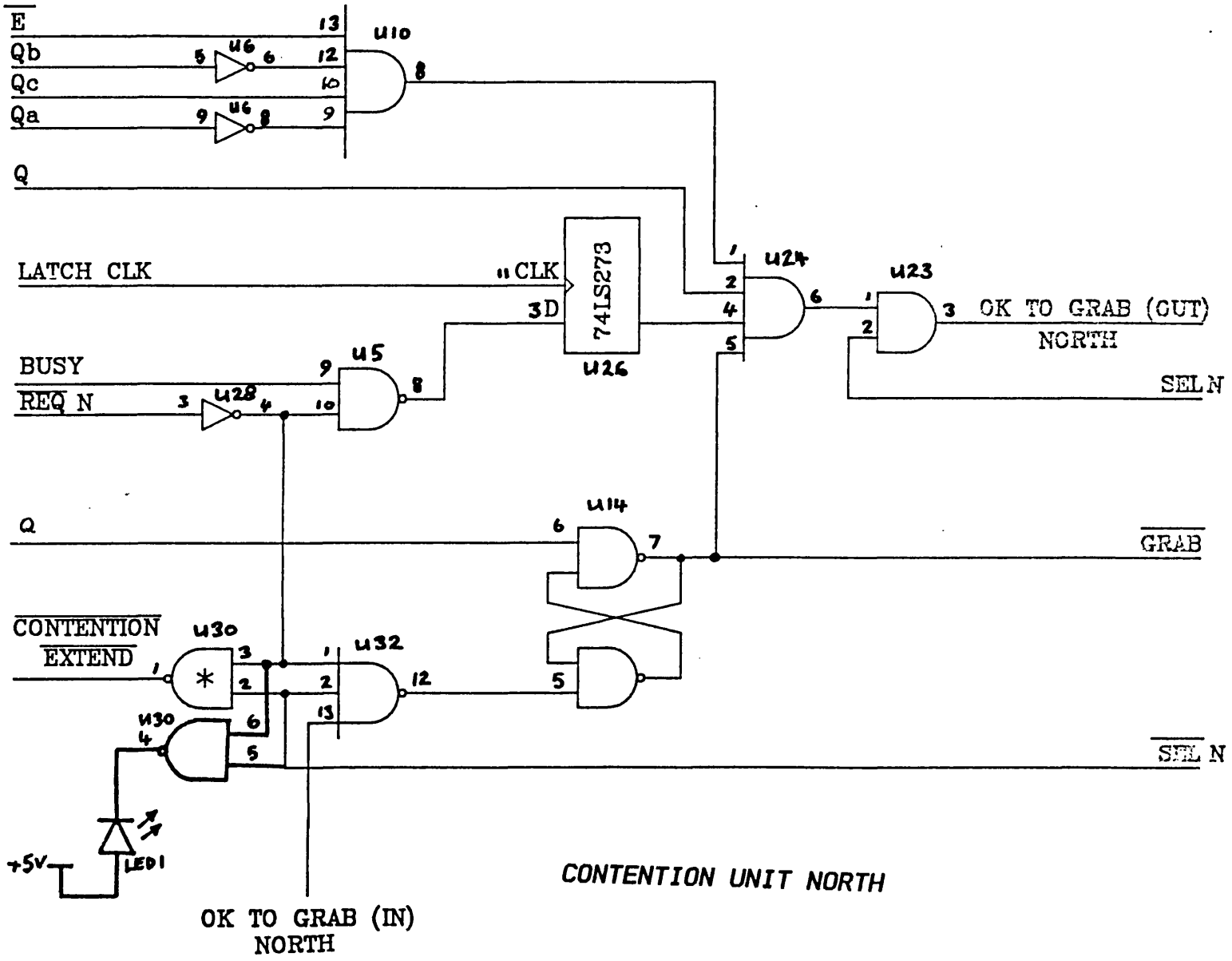

6264
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
D0 D1 D2 D3 D4 D5 D6 D7
CE1
OE
NC
CE2
VCC
WE
GND
CS PROGRAM RAM1

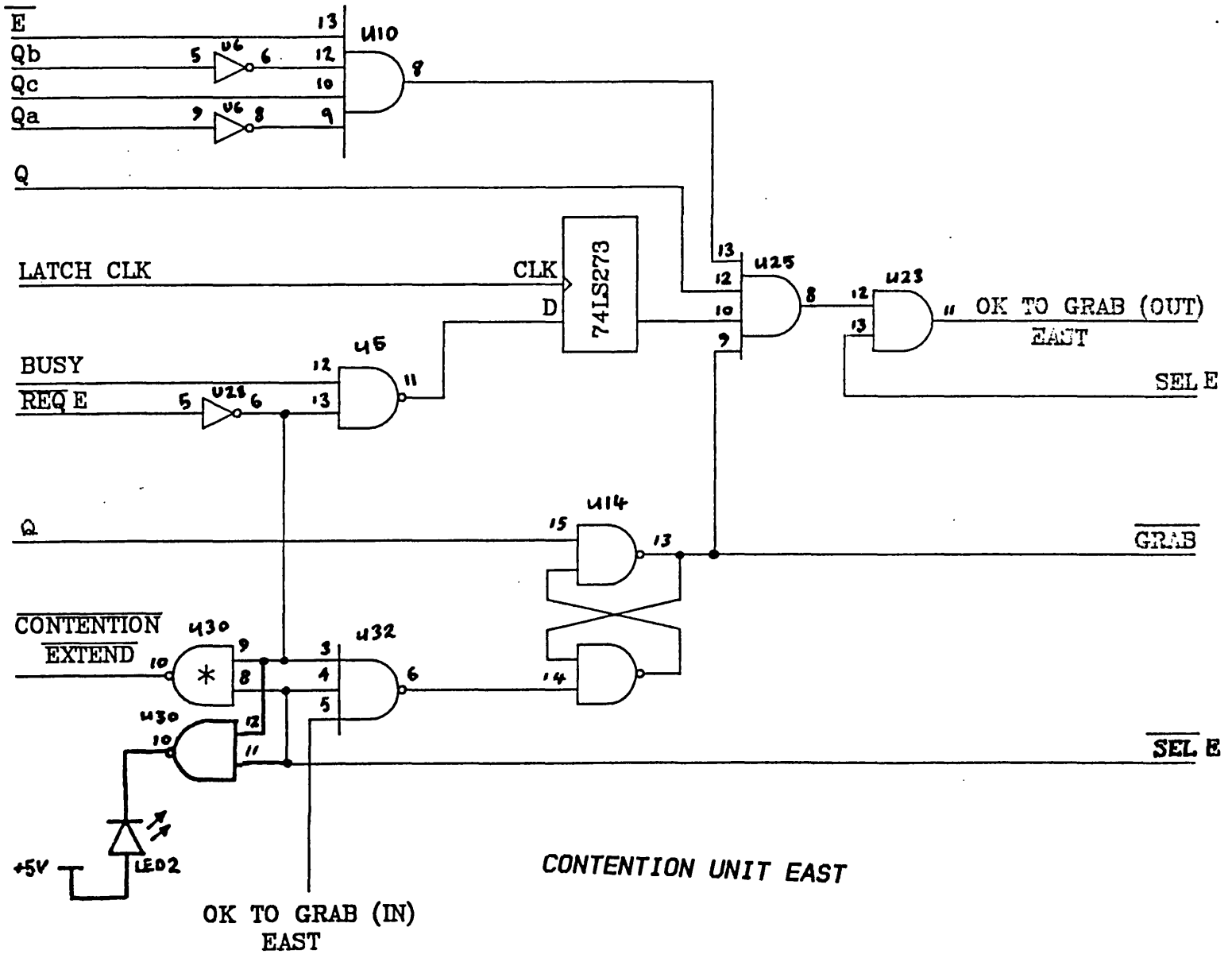

2764
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
D0 D1 D2 D3 D4 D5 D6 D7
CE1
OE
NC
CE2
VCC
GND
CS PROGRAM ROM

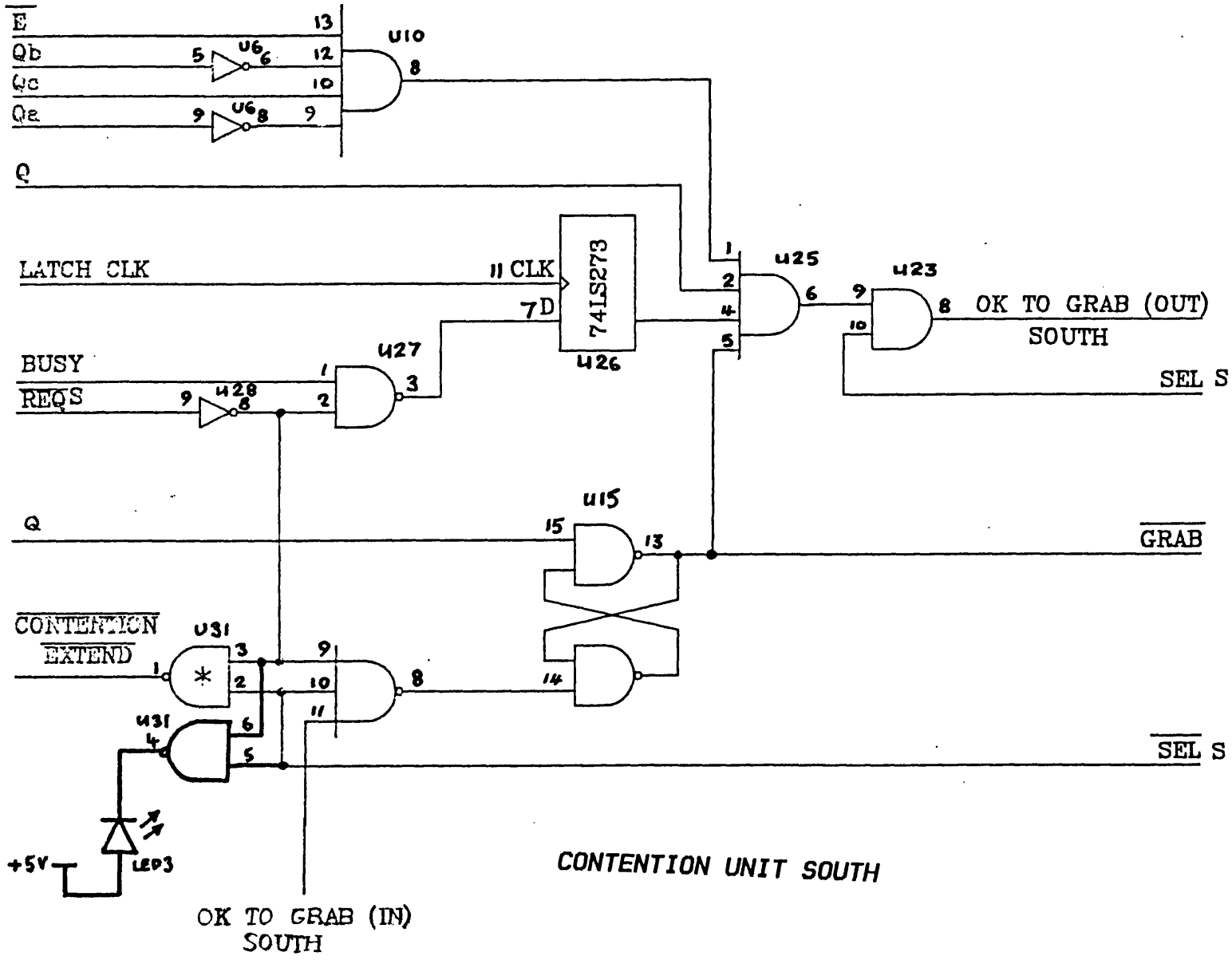
MEMORY



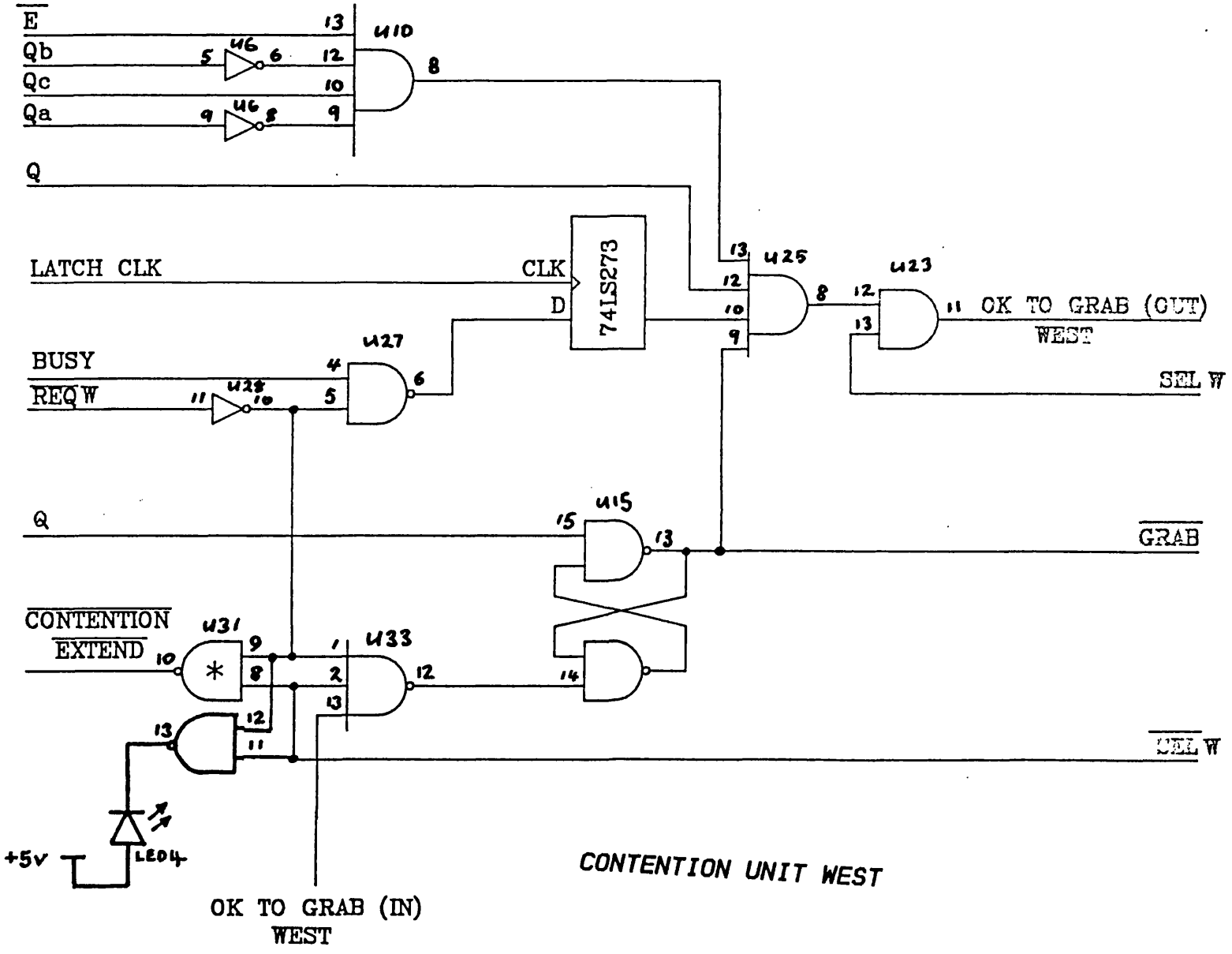
LATCH CLOCK


E
Qb
Qc
Qa
U6
U10
Q
LATCH CLK
CLK
3D
74LS273
U26
U24
U23
OK TO GRAB (OUT)
NORTH
SEL N
BUSY
REQ N
U28
U5
Q
U14
GRAB
CONTENTION
EXTEND
U30
U32
SEL N
+5V
LED1
OK TO GRAB (IN)
NORTH
CONTENTION UNIT NORTH

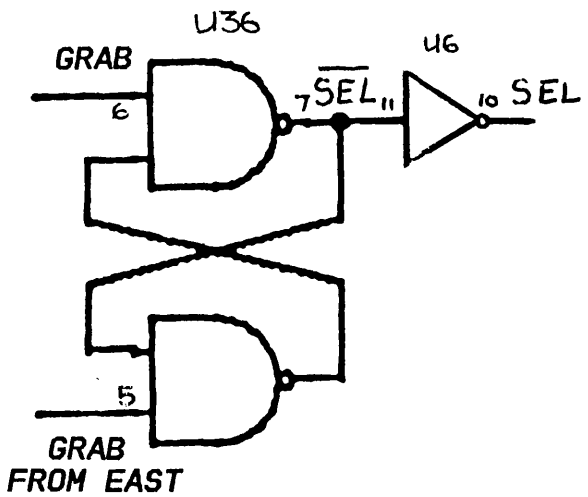

E
Qb
Qc
Qa
U10
U6
Q
LATCH CLK
CLK
D
74LS273
U25
U23
OK TO GRAB (OUT)
EAST
SEL E
BUSY
REQ E
U28
U5
Q
U14
GRAB
CONTENTION
EXTEND
U30
U32
SEL E
+5V
LED2
OK TO GRAB (IN)
EAST
CONTENTION UNIT EAST



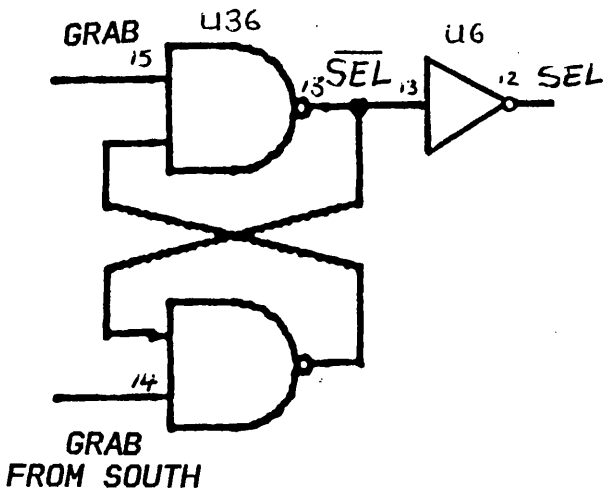
CONTENTION UNIT SOUTH



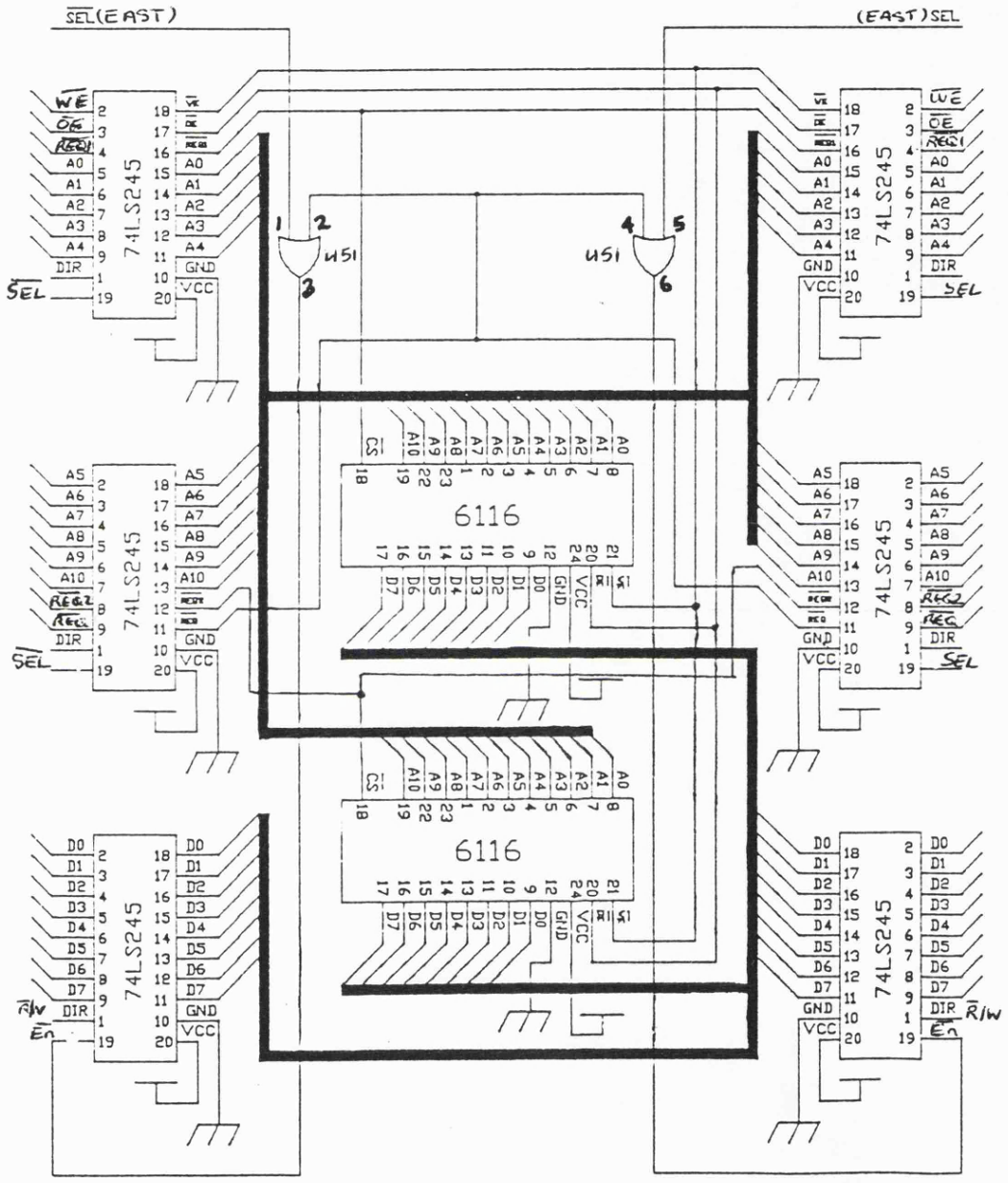
CONTENTION UNIT WEST



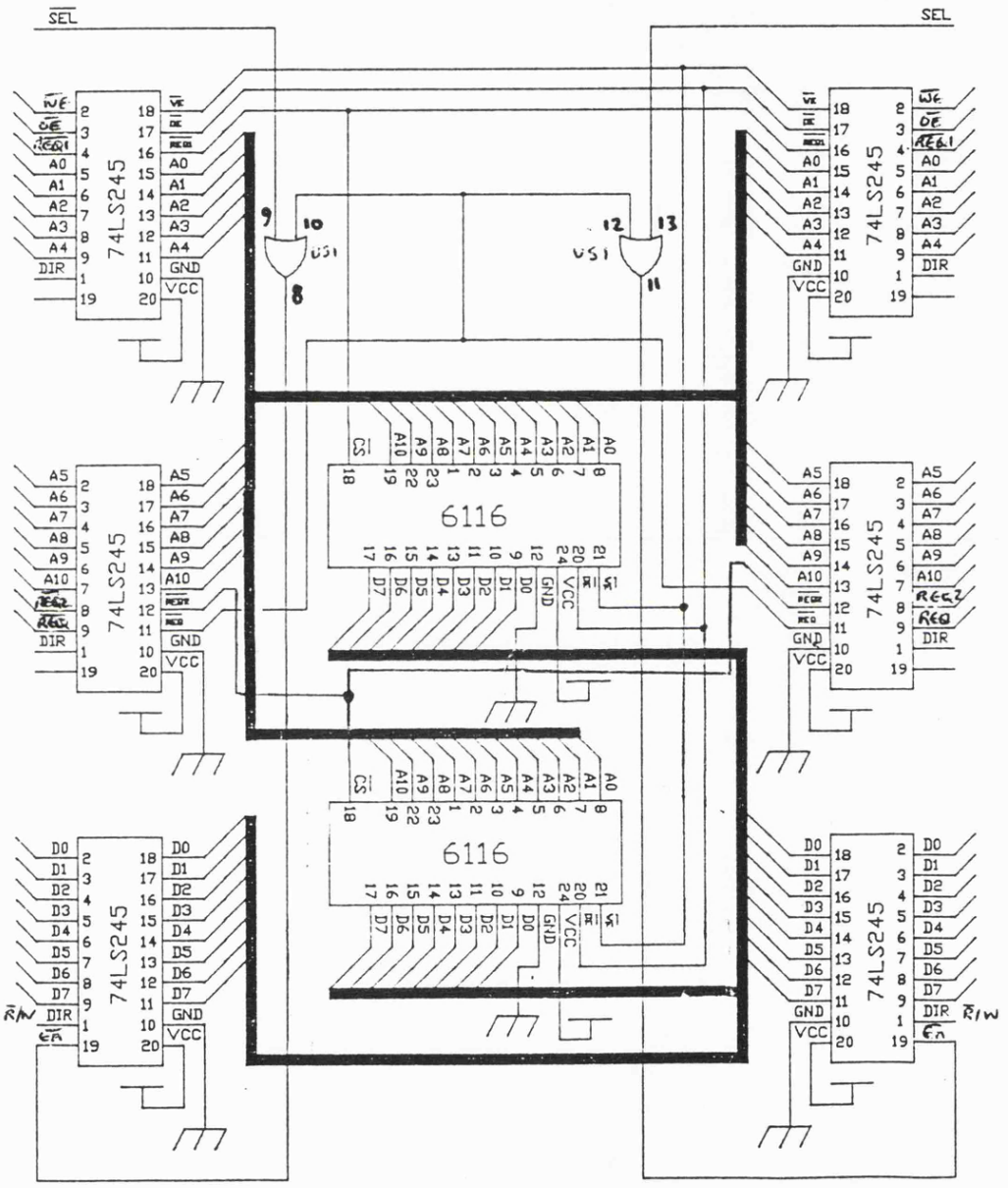
EAST SHARED MEMORY SELECT



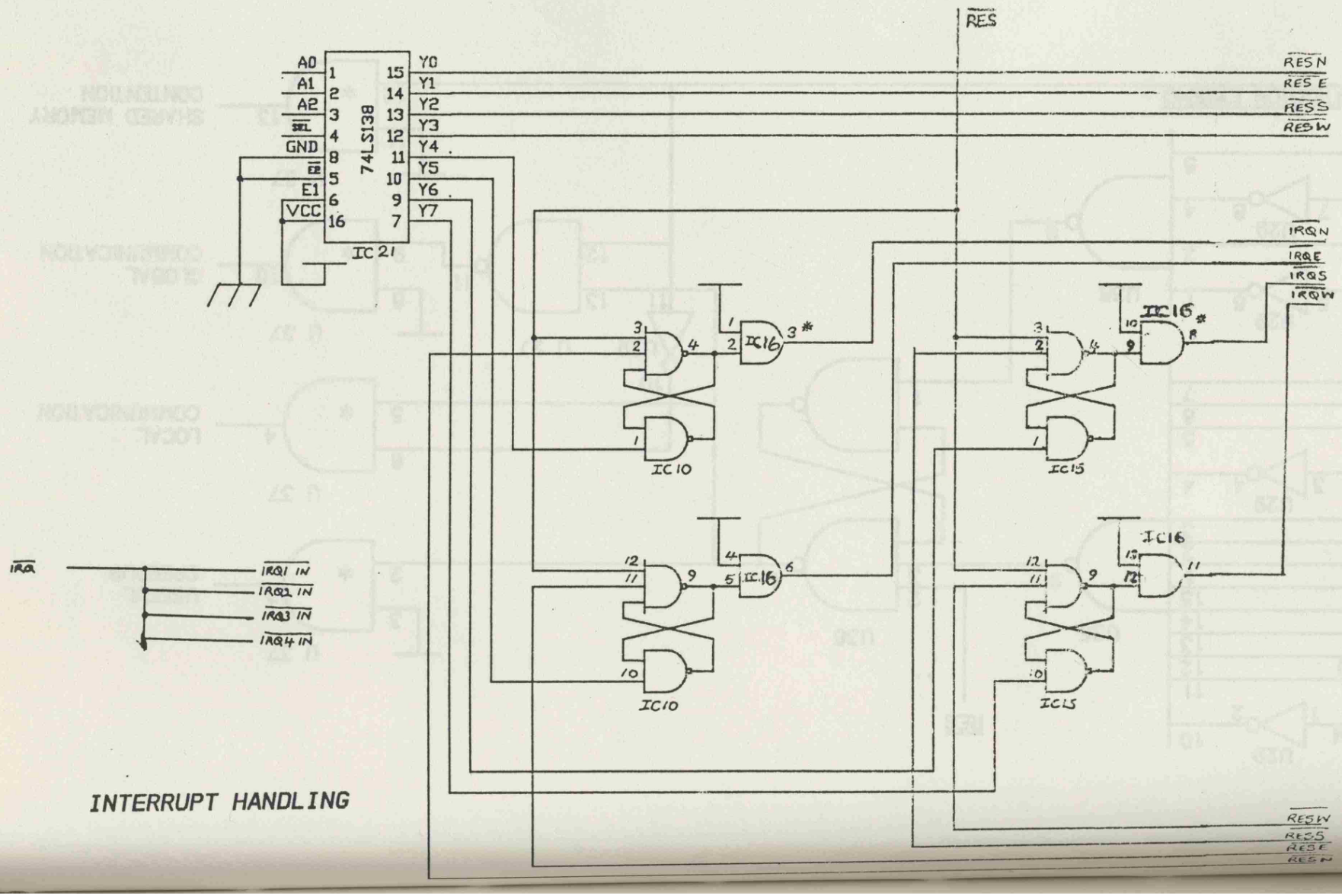
SOUTH SHARED MEMORY SELECT



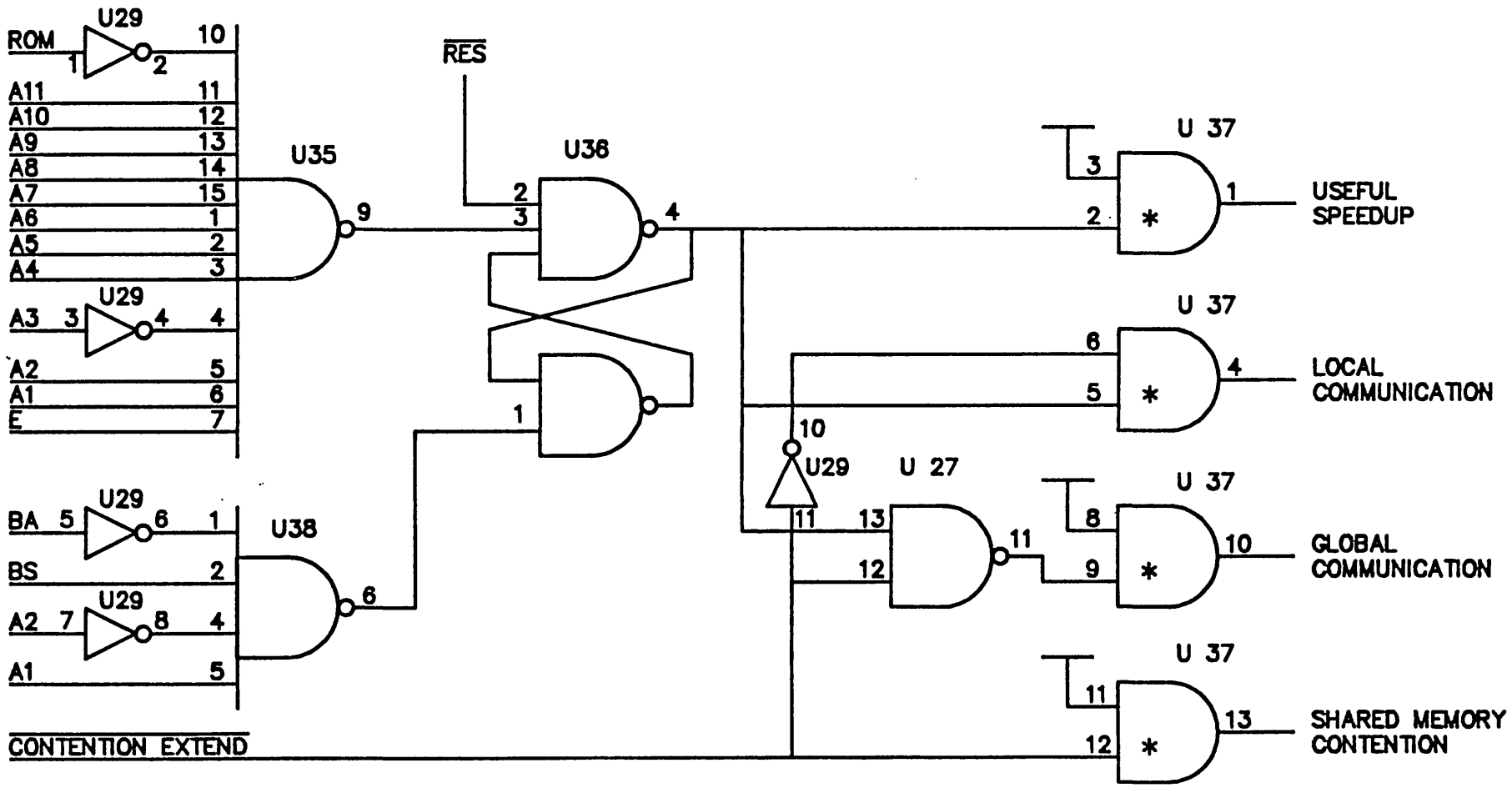
EAST SHARED MEMORY BLOCK



SOUTH SHARED MEMORY BLOCK



INTERRUPT HANDLING



STATUS MONITOR

COMPONENT LISTINGS

COMPONENTS OF A PROCESSING ELEMENT

IC no.	DEVICE
1	XTAL
2	74LS161
3	74LS74
4	74LS02
5	74LS00
6	74LS04
7	MC6809E
8	74LS154
9	74LS08
10	74LS21
11	74LS32
12	74LS32
13	74LS138
14	74LS279
15	74LS279
16	74LS09
17	2764
18	6264
19	6264
20	6264
21	6264
22	6264
23	74LS08
24	74LS21
25	74LS21
26	74LS273
27	74LS00
28	74LS04
29	74LS04
30	74LS01

continued ...

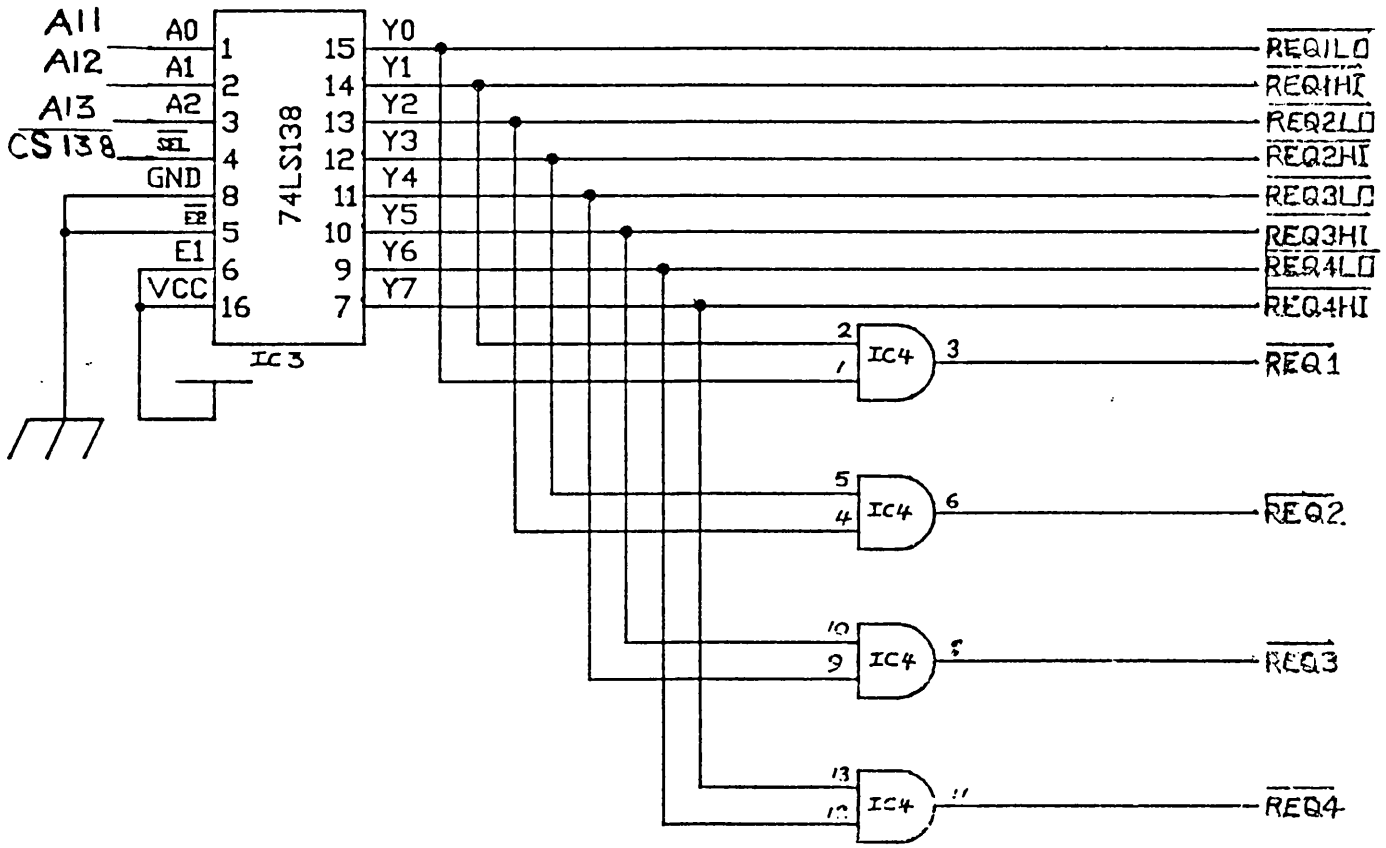
continued ...

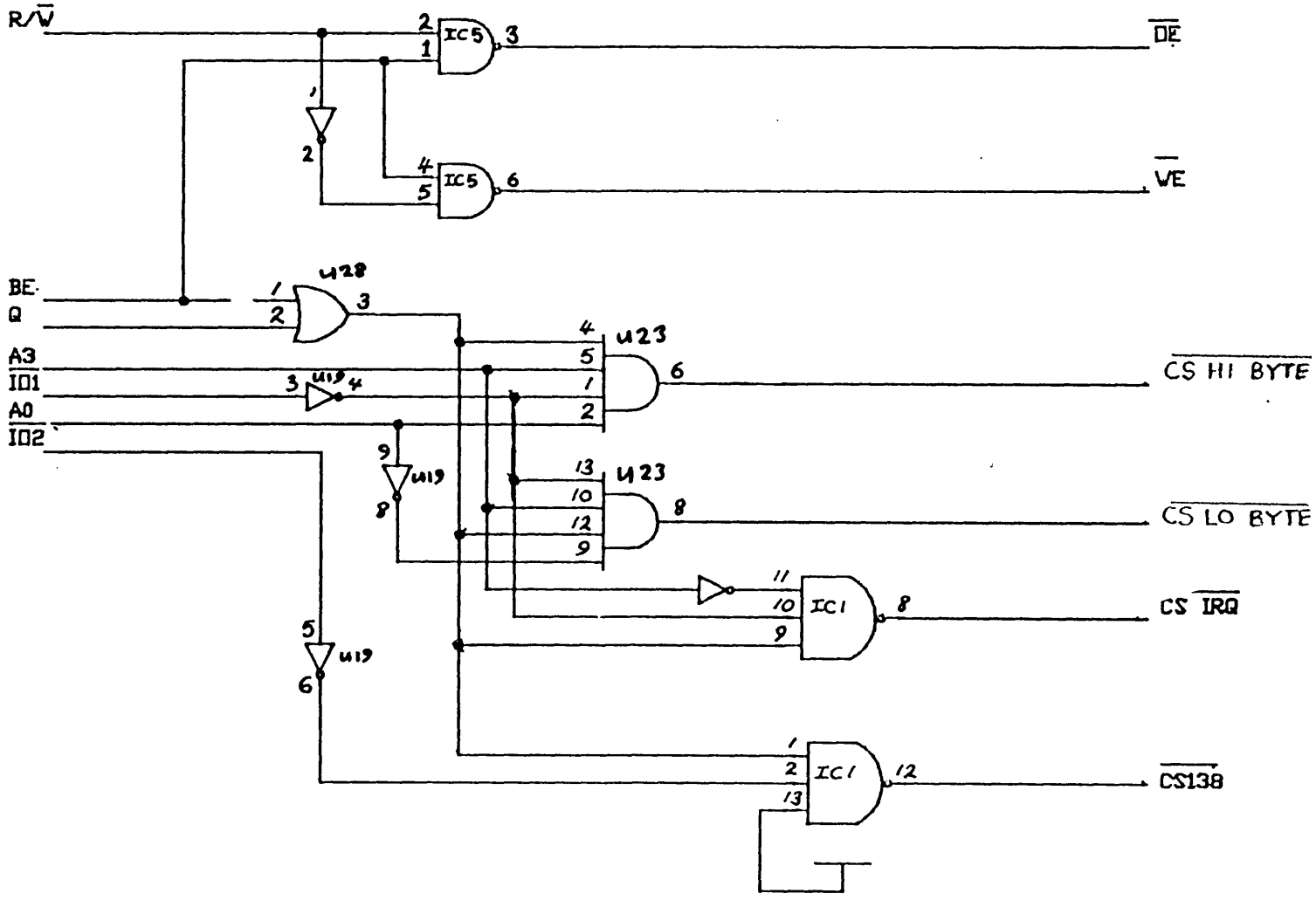
IC no.	DEVICE
31	74LS01
32	74LS10
33	74LS10
34	74LS273
35	74LS133
36	74LS279
37	74LS09
38	74LS20
39	74LS245
40	74LS245
41	74LS245
42	74LS245
43	74LS245
44	74LS245
45	74LS245
46	74LS245
47	74LS245
48	74LS245
49	74LS245
50	74LS245
51	74LS32
52	6116
53	6116
54	6116
55	6116

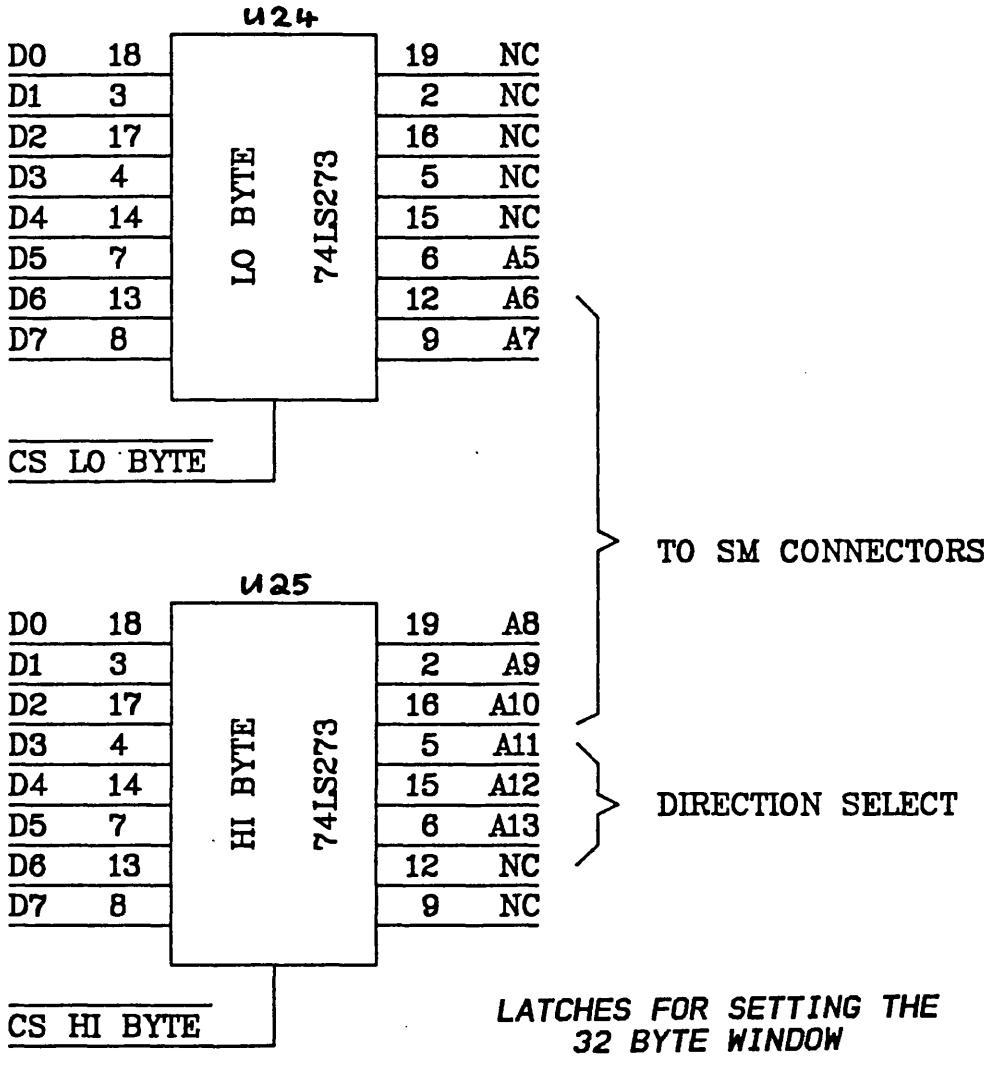
APPENDIX C

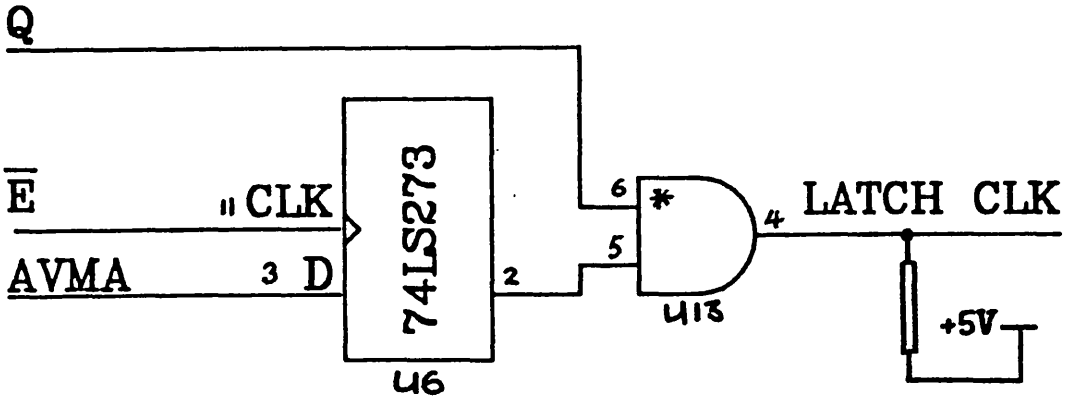
THE INTERFACE BOARD

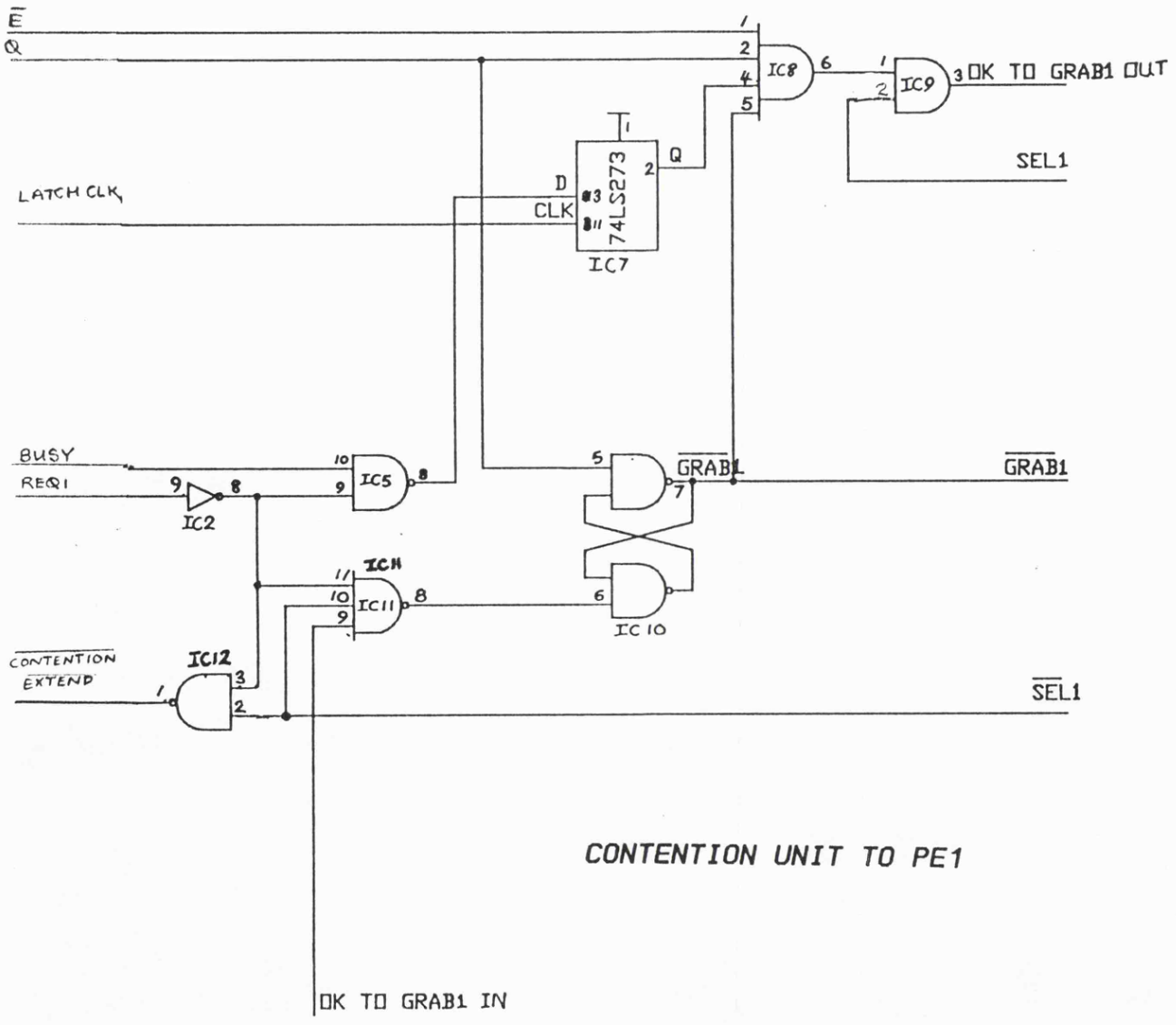
CIRCUIT DIAGRAMS


A11
A12
A13
CS138
A0
A1
A2
SEL
GND
E2
E1
VCC
74LS138
IC3
Y0
Y1
Y2
Y3
Y4
Y5
Y6
Y7
REQ1LO
REQ1HI
REQ2LO
REQ2HI
REQ3LO
REQ3HI
REQ4LO
REQ4HI
IC4
REQ1
REQ2
REQ3
REQ4

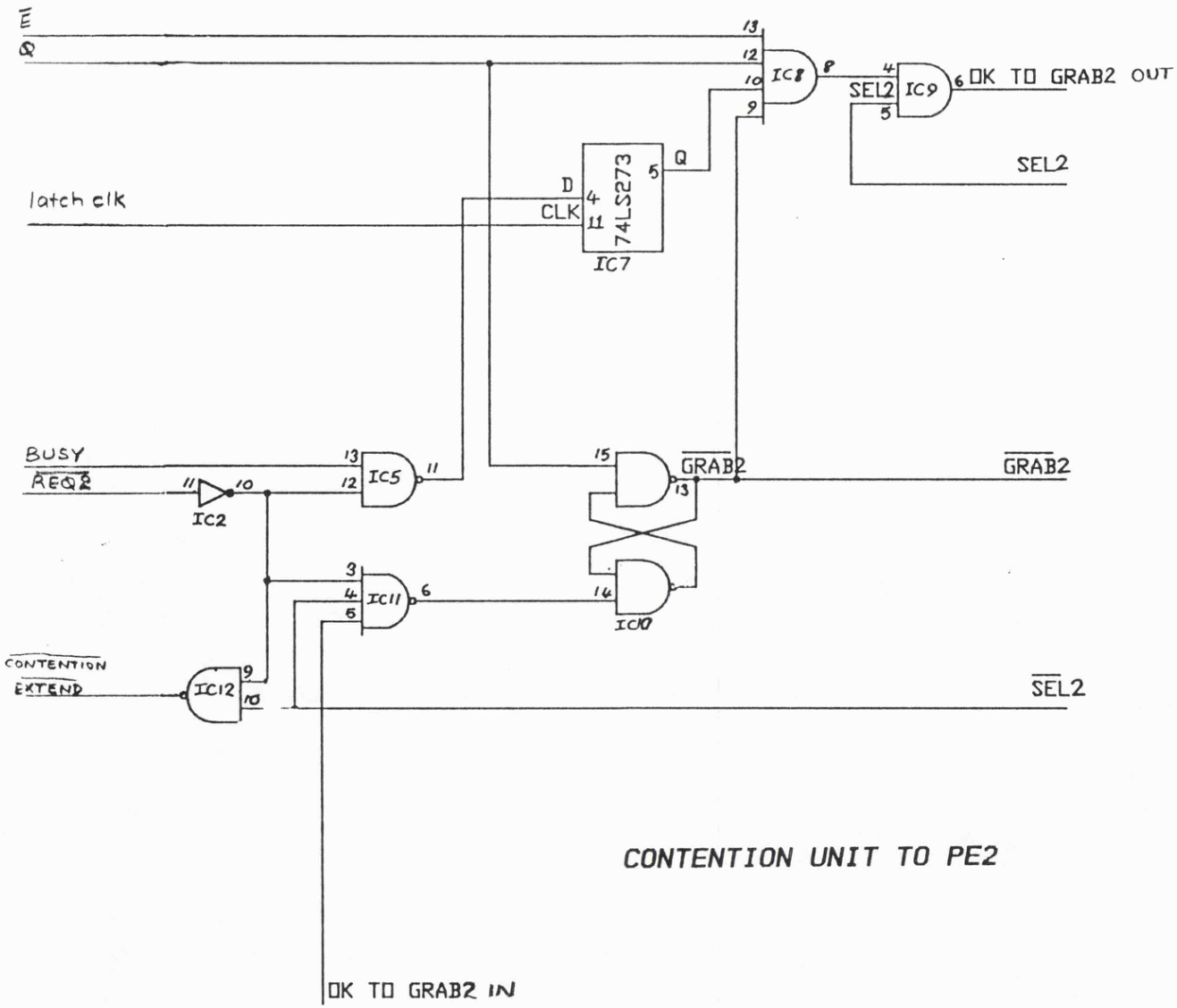

R/W
IC5
DE
IC5
WE
BE
Q
U28
A3
IO1
A0
IO2
U19
U23
CS HI BYTE
U23
CS LO BYTE
IC1
CS IRQ
U19
IC1
CS138


U24
D0 18
D1 3
D2 17
D3 4
D4 14
D5 7
D6 13
D7 8
LO BYTE
74LS273
19 NC
2 NC
16 NC
5 NC
15 NC
6 A5
12 A6
9 A7
CS LO BYTE
TO SM CONNECTORS
U25
D0 18
D1 3
D2 17
D3 4
D4 14
D5 7
D6 13
D7 8
HI BYTE
74LS273
19 A8
2 A9
16 A10
5 A11
15 A12
6 A13
12 NC
9 NC
DIRECTION SELECT
CS HI BYTE
LATCHES FOR SETTING THE
32 BYTE WINDOW

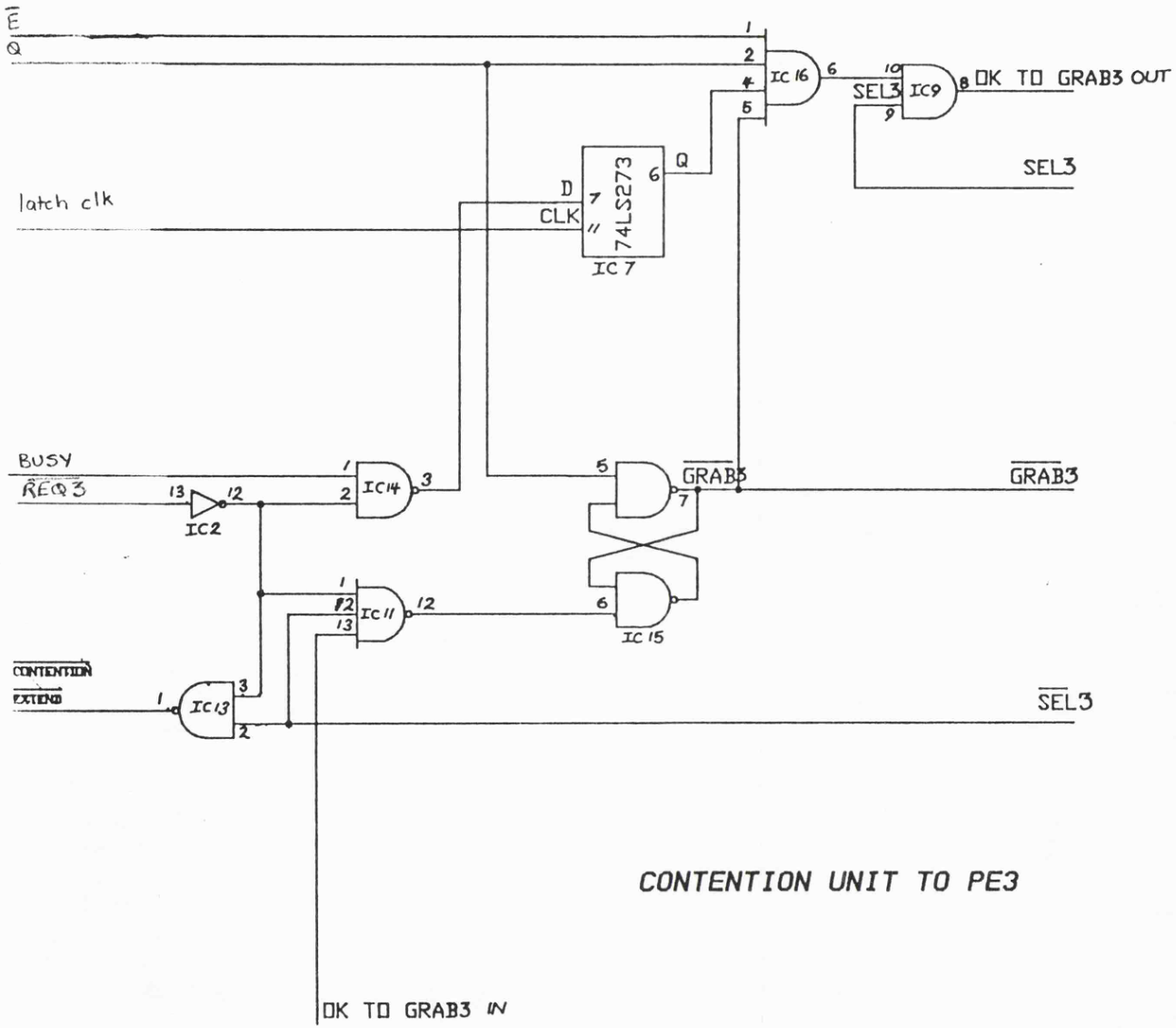

Q
E
11 CLK
AVMA
3 D
74LS273
U6
2
6
5
*
U13
4
LATCH CLK
+5V



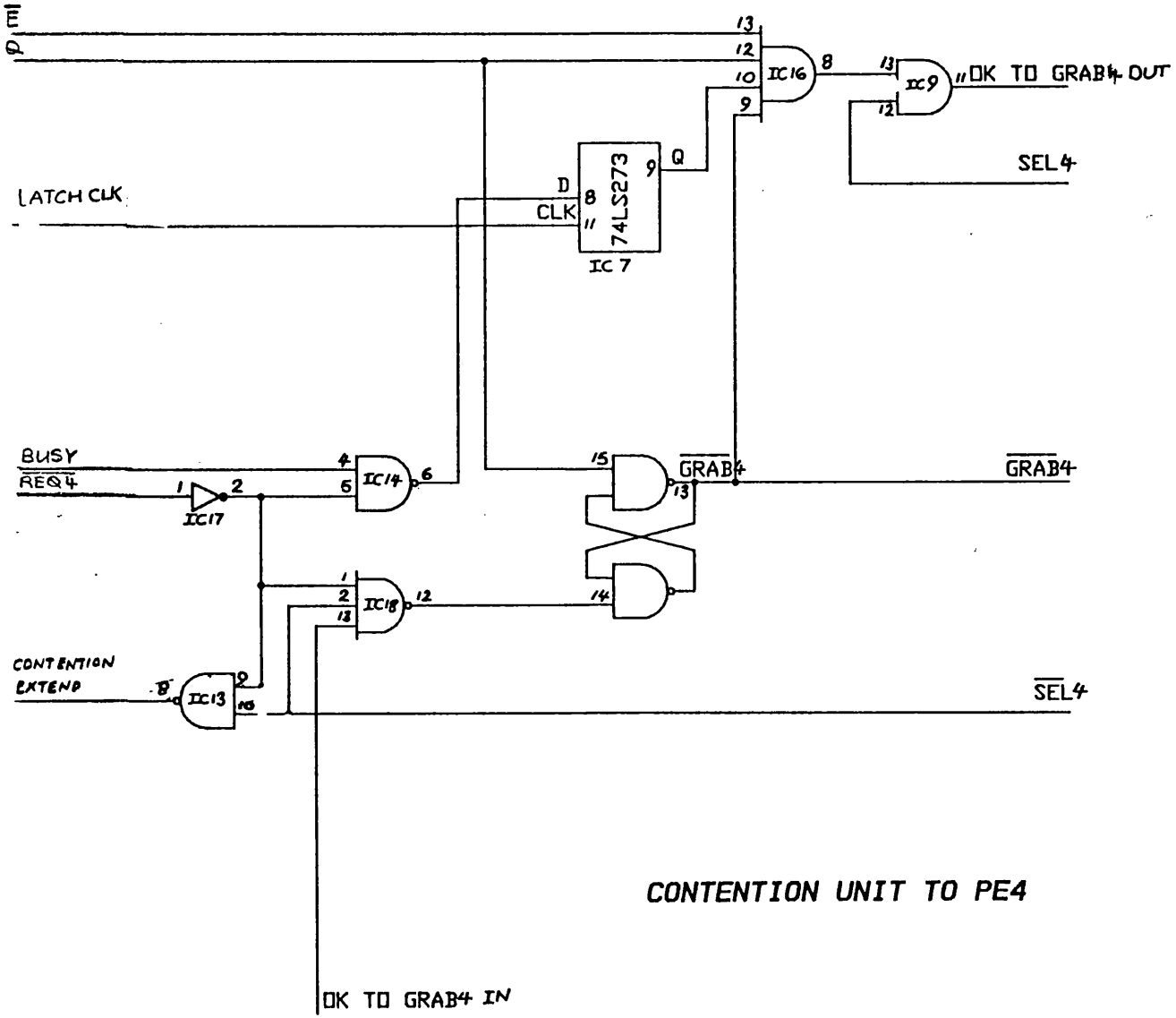
CONTENTION UNIT TO PE1



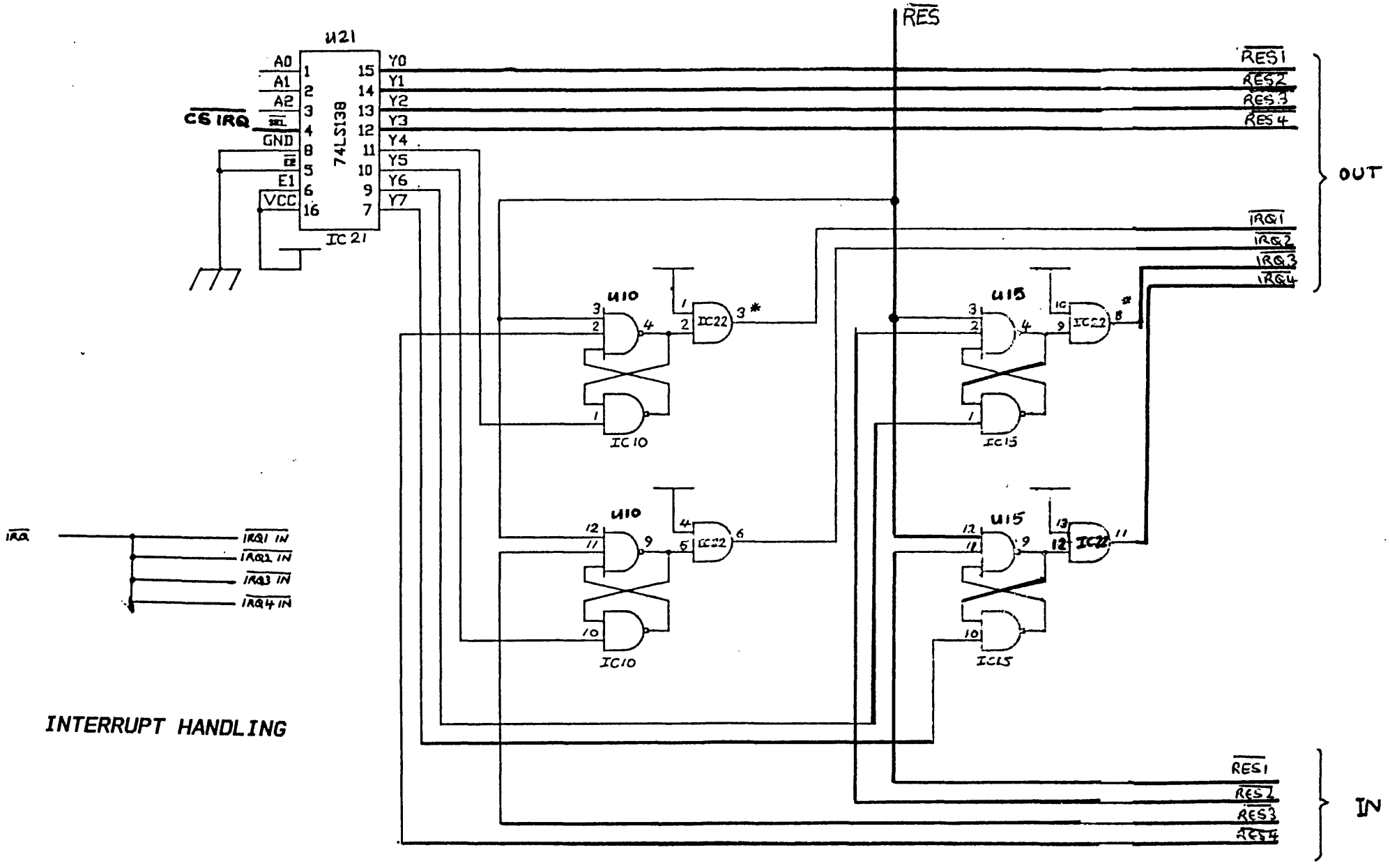
CONTENTION UNIT TO PE2



CONTENTION UNIT TO PE3



CONTENTION UNIT TO PE4



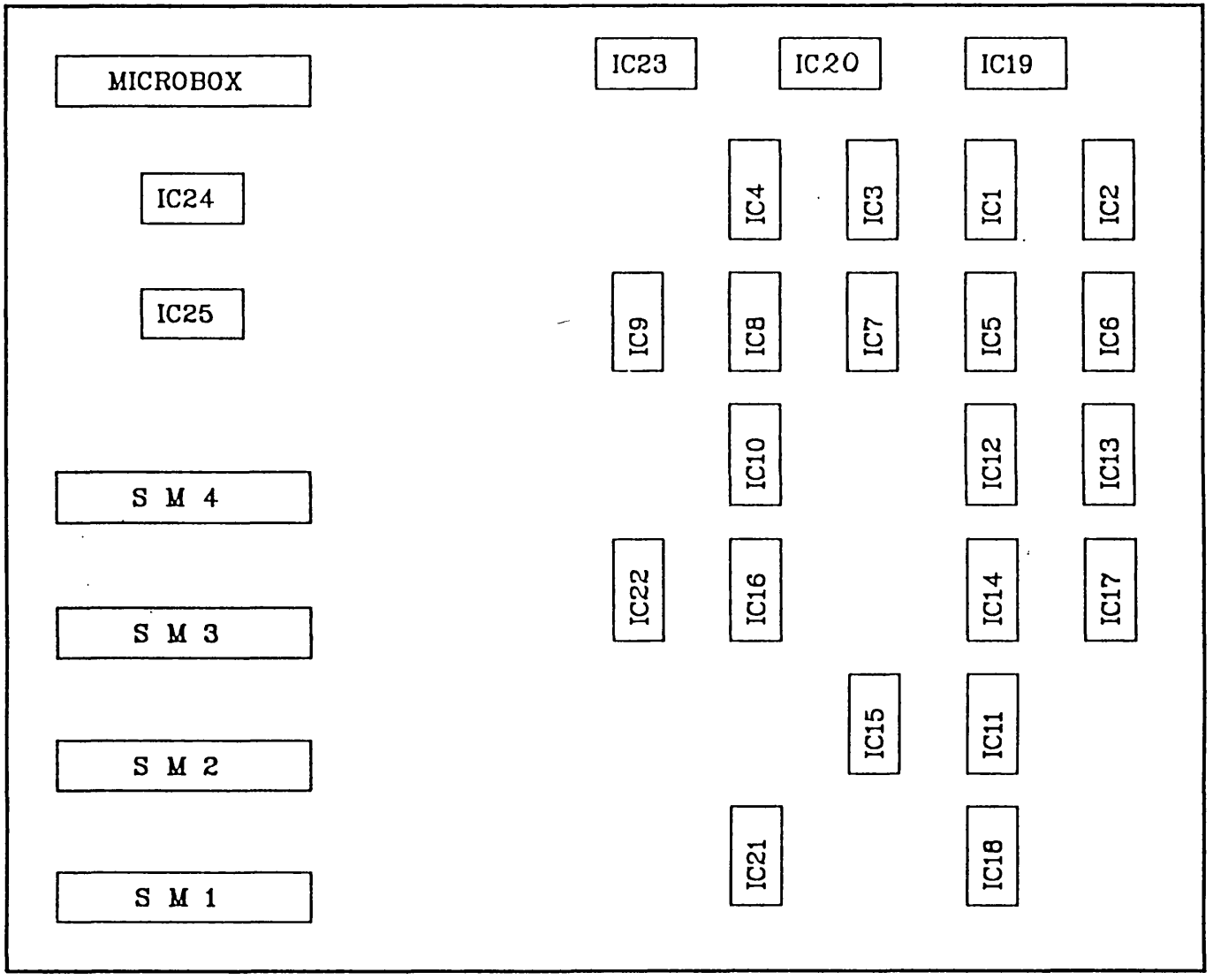
INTERRUPT HANDLING

COMPONENT LISTINGS

THE COMPONENTS OF THE MASTER INTERFACE BOARD

IC no.	DEVICE
1	74LS10
2	74LS04
3	74LS138
4	74LS08
5	74LS00
6	74LS273
7	74LS273
8	74LS21
9	74LS00
10	74LS279
11	74LS10
12	74LS01
13	74LS01
14	74LS00
15	74LS279
16	74LS21
17	74LS32
18	74LS04
19	74LS20
20	74LS04
21	74LS138
22	74LS09
23	74LS373
24	74LS373

COMPONENT POSITIONS


MICROBOX
IC23
IC20
IC19
IC24
IC25
IC4
IC3
IC1
IC2
IC9
IC8
IC7
IC5
IC6
IC10
IC12
IC13
S M 4
IC22
IC16
IC14
IC17
S M 3
IC15
IC11
S M 2
IC21
IC18
S M 1

CONNECTORS

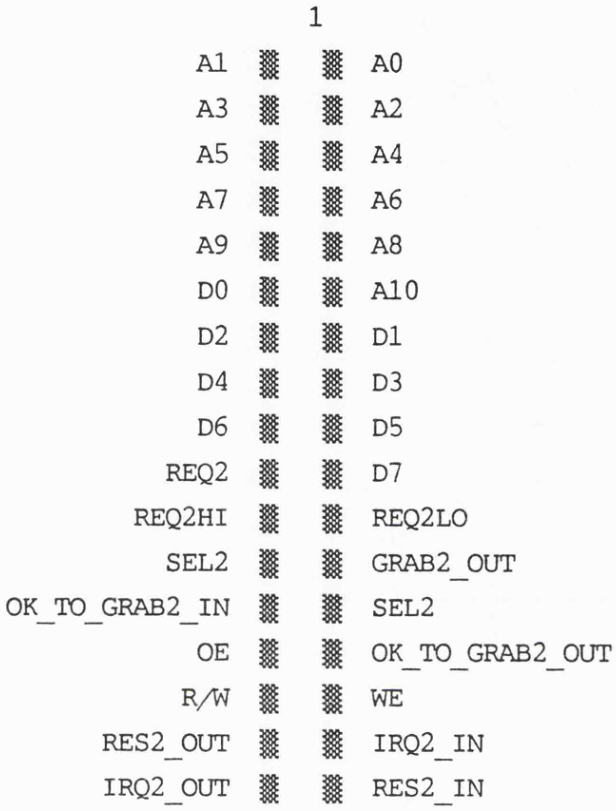
Expansion bus		
Pin No	Inner row	Outer row
1, 2	+5v	+5v
3, 4	Gnd	Gnd
5, 6	BA0	IC19 pin 6
7, 8	*BRTS	BA1
9,10	BD1	BD0
11,12	BD3	BD2
13,14	BD5	BD4
15,16	BD7	BD6
17,18	BA2	BR/W
19,20	BA4	BA3
21,22	16Mhz	BE
23,24	*WDS	Q
25,26	RTC	LPEN
27,28	*I/O2	*RDS
29,30	*I/O1	*I/OBUFF
31,32	*NMI	RST
33,34	*FIRQ	*IRQ
35,36	*TTLVID	VSYNC
37,38	Gnd	Gnd
39,40	-12v	+12v

CONNECTOR TO PE 1

1

A1	A0
A3	A2
A5	A4
A7	A6
A9	A8
D0	A10
D2	D1
D4	D3
D6	D5
REQ1	D7
REQ1HI	REQ1LO
SEL1	GRAB1_OUT
OK_TO_GRAB1_IN	SEL1
OE	OK_TO_GRAB1_OUT
R/W	WE
RES1_OUT	IRQ1_IN
IRQ1_OUT	RES1_IN

CONNECTOR TO PE 2



CONNECTOR TO PE 3

1

A1	A0
A3	A2
A5	A4
A7	A6
A9	A8
D0	A10
D2	D1
D4	D3
D6	D5
REQ3	D7
REQ3HI	REQ3LO
SEL3	GRAB3_OUT
OK_TO_GRAB3_IN	SEL3
OE	OK_TO_GRAB3_OUT
R/W	WE
RES3_OUT	IRQ3_IN
IRQ3_OUT	RES3_IN

CONNECTOR TO PE 4

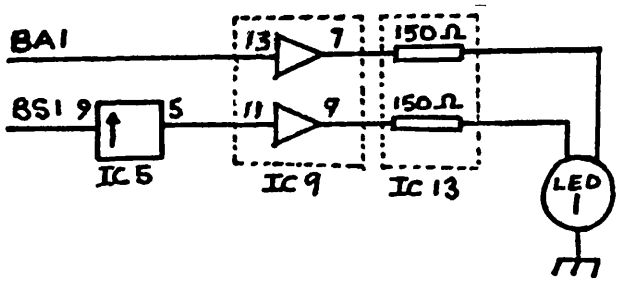
1

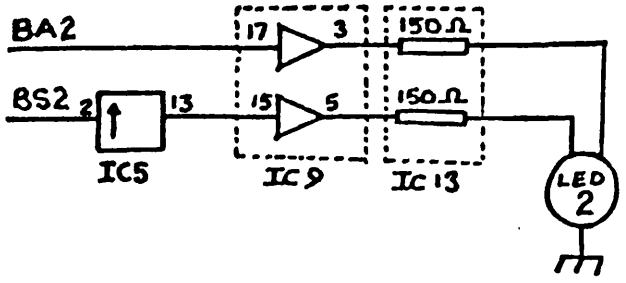
A1	A0
A3	A2
A5	A4
A7	A6
A9	A8
D0	A10
D2	D1
D4	D3
D6	D5
REQ4	D7
REQ4HI	REQ4LO
SEL4	GRAB4_OUT
OK_TO_GRAB4_IN	SEL4
OE	OK_TO_GRAB4_OUT
R/W	WE
RES4_OUT	IRQ4_IN
IRQ4_OUT	RES4_IN

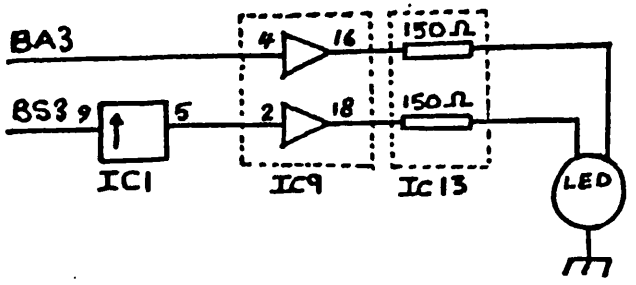
APPENDIX D

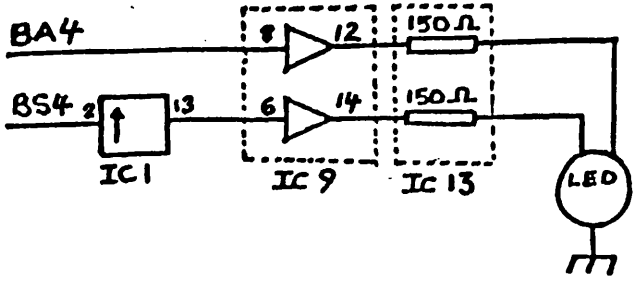
THE STATUS MONITOR BOARD

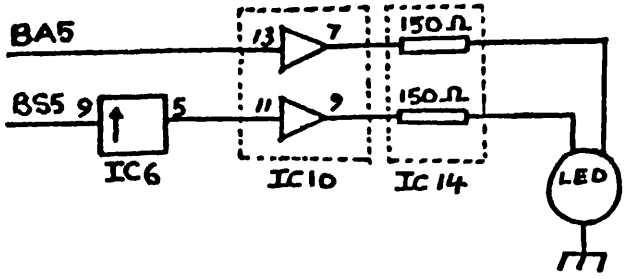
CIRCUIT DIAGRAMS

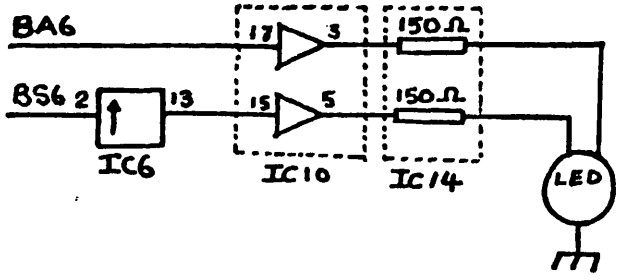

BA1
BS1
9
5
IC5
13
7
11
9
IC9
150Ω
150Ω
IC13
LED
1

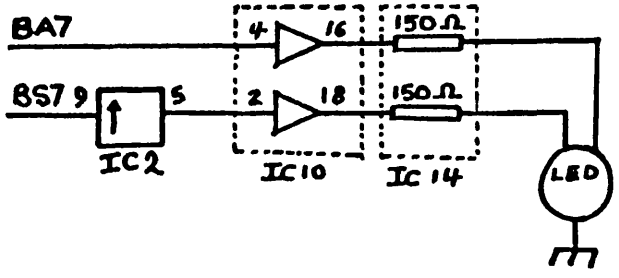

BA2
BS2
2
13
IC5
17
3
15
5
IC9
150Ω
150Ω
IC13
LED
2

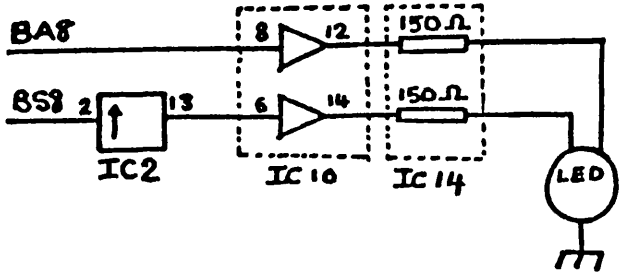

BA3
BS3
9
5
IC1
4
16
2
18
IC9
150Ω
150Ω
IC13
LED

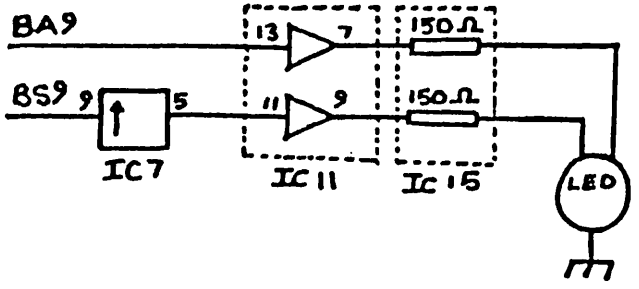

BA4
BS4
2
13
IC1
8
12
6
14
IC9
150Ω
150Ω
IC13
LED

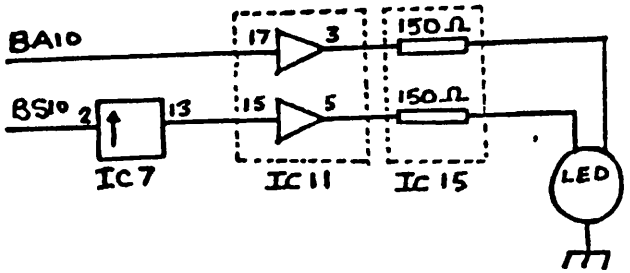

BA5
BS5
9
5
IC6
13
7
11
9
IC10
150Ω
150Ω
IC14
LED

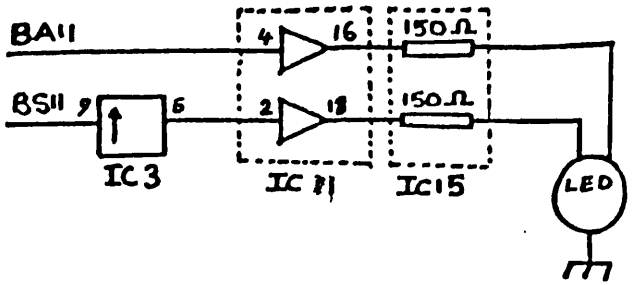

BA6
BS6
2
13
IC6
17
3
15
5
IC10
150Ω
150Ω
IC14
LED

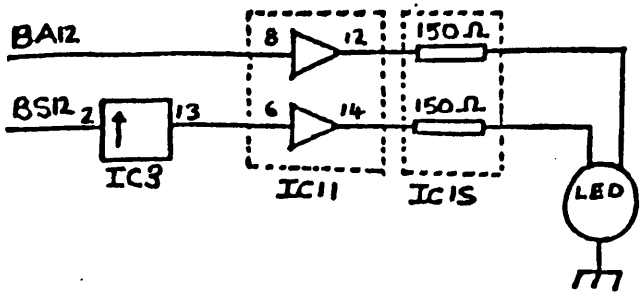

BA7
BS7
9
5
IC2
4
16
2
18
IC10
150Ω
150Ω
IC14
LED

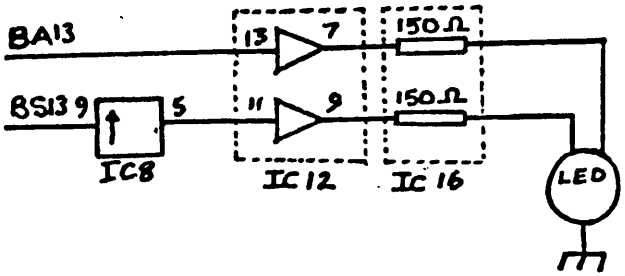

BA8
BS8
2
13
IC2
8
12
6
14
IC10
150Ω
150Ω
IC14
LED

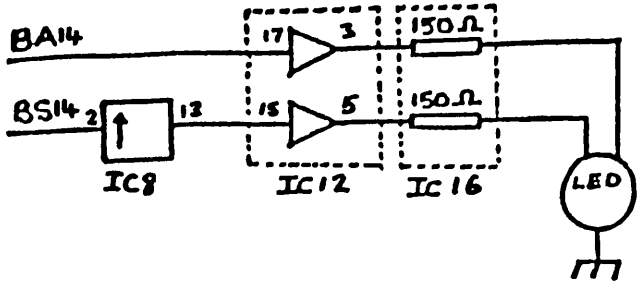

BA9
BS9
9
5
IC7
13
7
11
9
IC11
150Ω
150Ω
IC15
LED

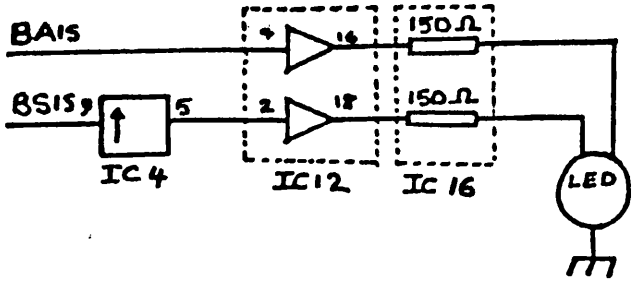


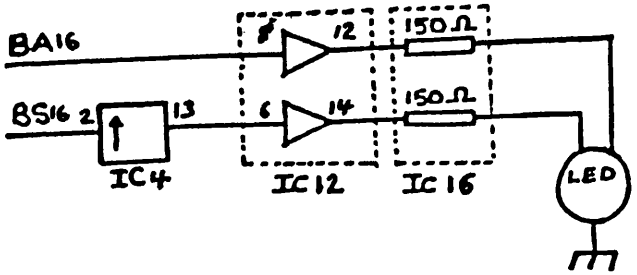




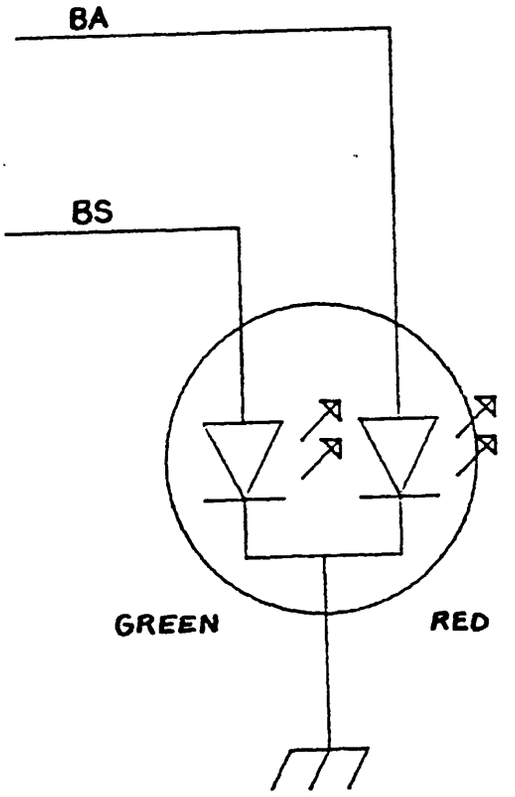








A DETAILED VIEW OF AN LED
CAN BE SEEN OVERLEAF



SIGNAL BA	BS	RESULTANT LED COLOUR
0	0	YELLOW
0	1	GREEN
1	0	RED

DETAILED VIEW OF AN LED

COMPONENT LISTINGS

COMPONENTS OF THE STATUS MONITOR BOARD

IC No.	Device
1	74LS221
2	74LS221
3	74LS221
4	74LS221
5	74LS221
6	74LS221
7	74LS221
8	74LS221
9	74LS244
10	74LS244
11	74LS244
12	74LS244
13	150Ω DIL
14	150Ω DIL
15	150Ω DIL
16	150Ω DIL

ALL TIMING RESISTORS 47 kΩ
ALL TIMING CAPACITORS 470 μF

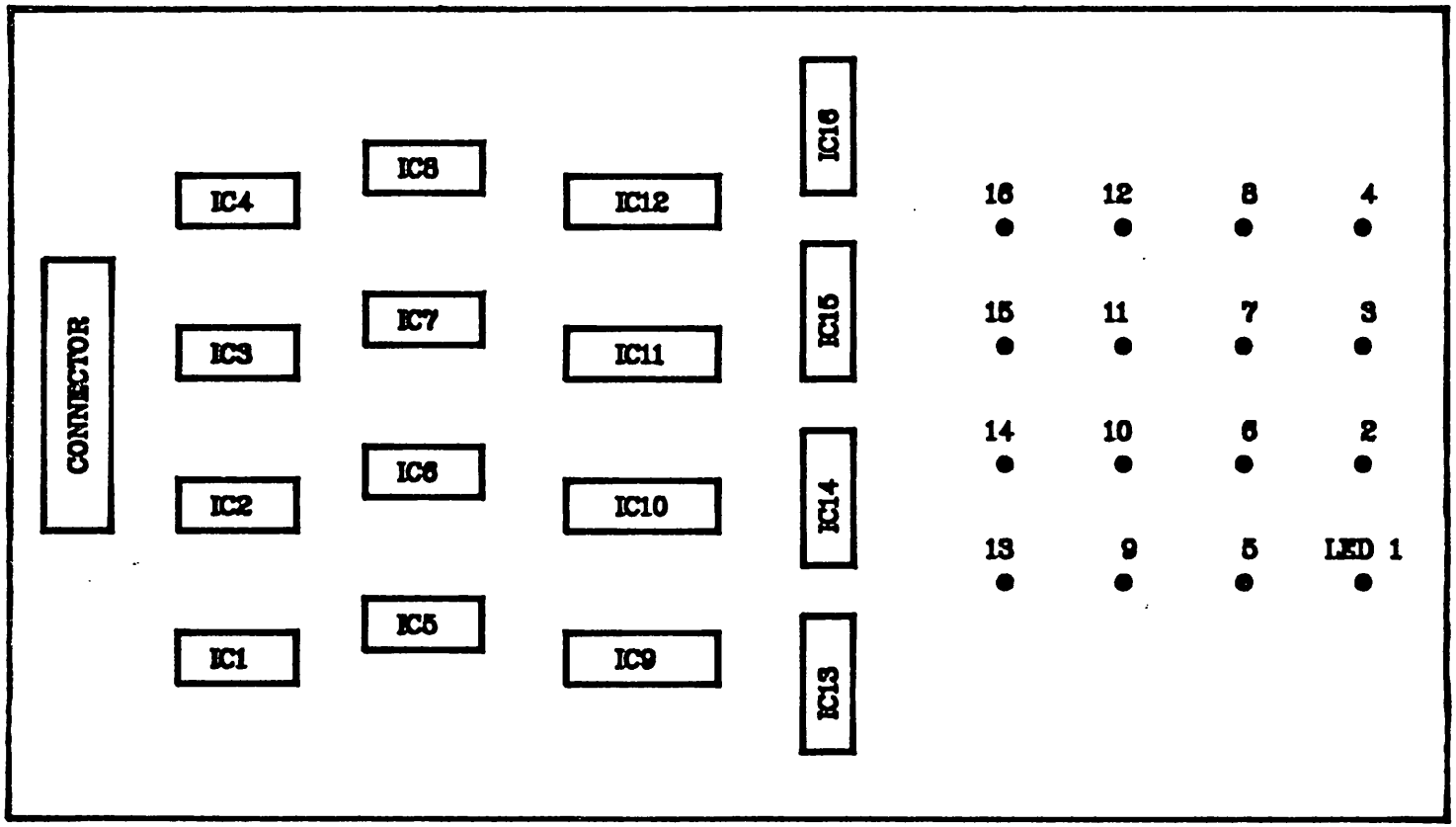
THE CONNECTOR

STATUS MONITOR BOARD CONNECTOR

BS16	▒	▒	BA16
BS15	▒	▒	BA15
BS14	▒	▒	BA14
BS13	▒	▒	BA13
BS12	▒	▒	BA12
BS11	▒	▒	BA11
BS10	▒	▒	BA10
BS9	▒	▒	BA9
BS8	▒	▒	BA8
BS7	▒	▒	BA7
BS6	▒	▒	BA6
BS5	▒	▒	BA5
BS4	▒	▒	BA4
BS3	▒	▒	BA3
BS2	▒	▒	BA2
BS1	▒	▒	BA1
	▒	▒	

(SOLDER SIDE)

COMPONENT POSITIONS

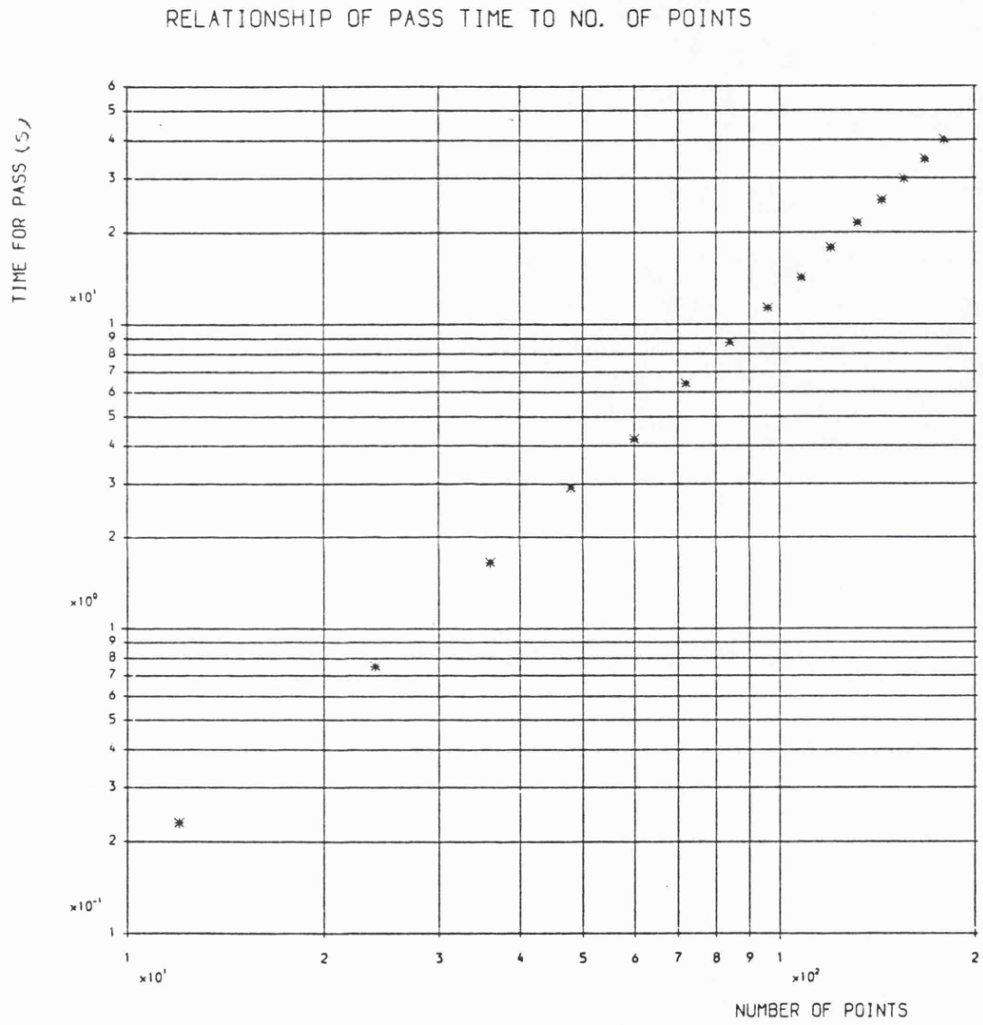


COMPONENT POSITIONS ON THE STATUS MONITOR BOARD

APPENDIX E

VERIFICATION OF THE TIME TO SOLUTION FOR THE 16PE CASE

APPENDIX E Relationship between time to solution (t) and number of points (p^2) for the 16PE case to verify linearity



LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO THE SIZE OF THE PROBLEM MESH

APPENDIX F

PROCESSING TESTS ON THE MULTIPROCESSOR RUNNING THE OPTIMIZED PROGRAM

APPENDIX F Optimized results

Table F.1 shows the time for 5 passes of program 4, with an optimized 'get_address' procedure, for various sizes of architecture and problem.

size	1PE	4PE	9PE	16PE
12x12	3.62	1.28	0.85	0.64
24x24	16.38	5.14	2.98	1.53
36x36	38.63	11.85	5.87	3.41
48x48	70.57	21.43	11.85	5.90
60x60	111.3	34.03	15.97	9.07
72x72	162.1	49.38	23.03	10.03
84x84		67.88	31.58	17.71
96x96		89.08	40.94	23.04
108x108		113.4	51.71	29.11
120x120		140.4	63.84	36.09
132x132		170.5	77.25	43.63
144x144		203.3	92.05	51.73
156x156			107.9	60.78
168x168			125.1	70.47
180x180			143.9	80.80
192x192			163.6	92.00
204x204			184.8	104.0
216x216			207.6	116.7
228x228				129.8
240x240				143.9
252x252				158.7
264x264				174.2
276x276				190.5
288x288				208.0

TABLE F.1 5 Pass time (s)

Table F.2 shows the average pass time of the data in table F.1.

size	1PE	4PE	9PE	16PE
12x12	0.72	0.26	0.17	0.13
24x24	3.28	1.03	0.59	0.31
36x36	7.73	2.37	1.17	0.68
48x48	14.11	4.28	2.12	1.18
60x60	22.27	6.81	3.20	1.81
72x72	32.42	9.88	4.61	2.61
84x84		13.57	6.32	3.54
96x96		17.82	8.19	4.68
108x108		22.67	10.34	5.82
120x120		28.07	12.77	7.21
132x132		34.10	15.45	8.73
144x144		40.67	18.41	10.35
156x156			21.58	12.16
168x168			25.02	14.10
180x180			28.78	16.16
192x192			32.73	18.40
204x204			36.96	20.80
216x216			41.52	23.33
228x228				25.97
240x240				28.79
252x252				31.75
264x264				34.83
276x276				38.09
288x288				41.60

Table F.2 Average pass times (s)

The logarithmic plots in figures F.1, F.2 and F.3 of the data in table F.2 yeald the following equations for the extrapolated data which cannot be obtained experimentally due to memory restrictions:

1PE $$t = \frac{p^{2.0853}}{229.21}$$

4PE $$t = \frac{p^{2.0516}}{658.828}$$

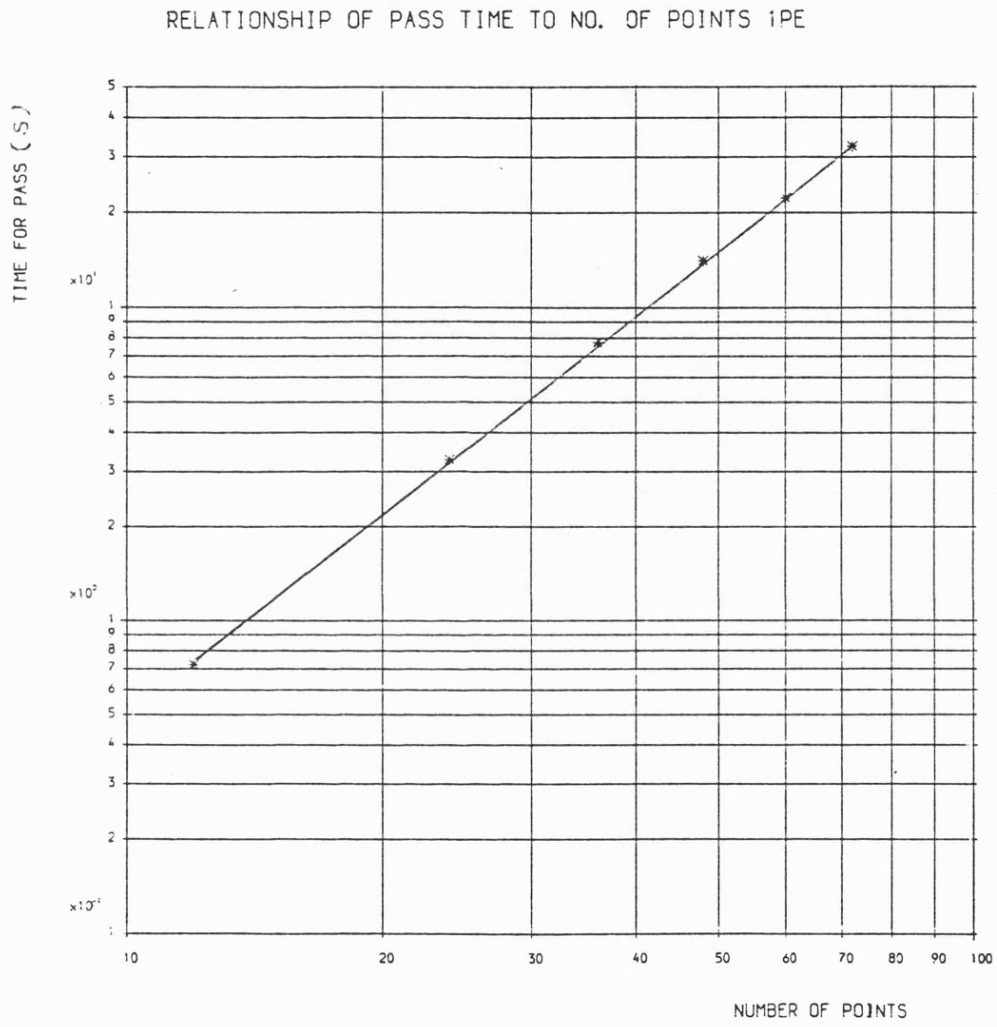


FIGURE F.1 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR 1PE

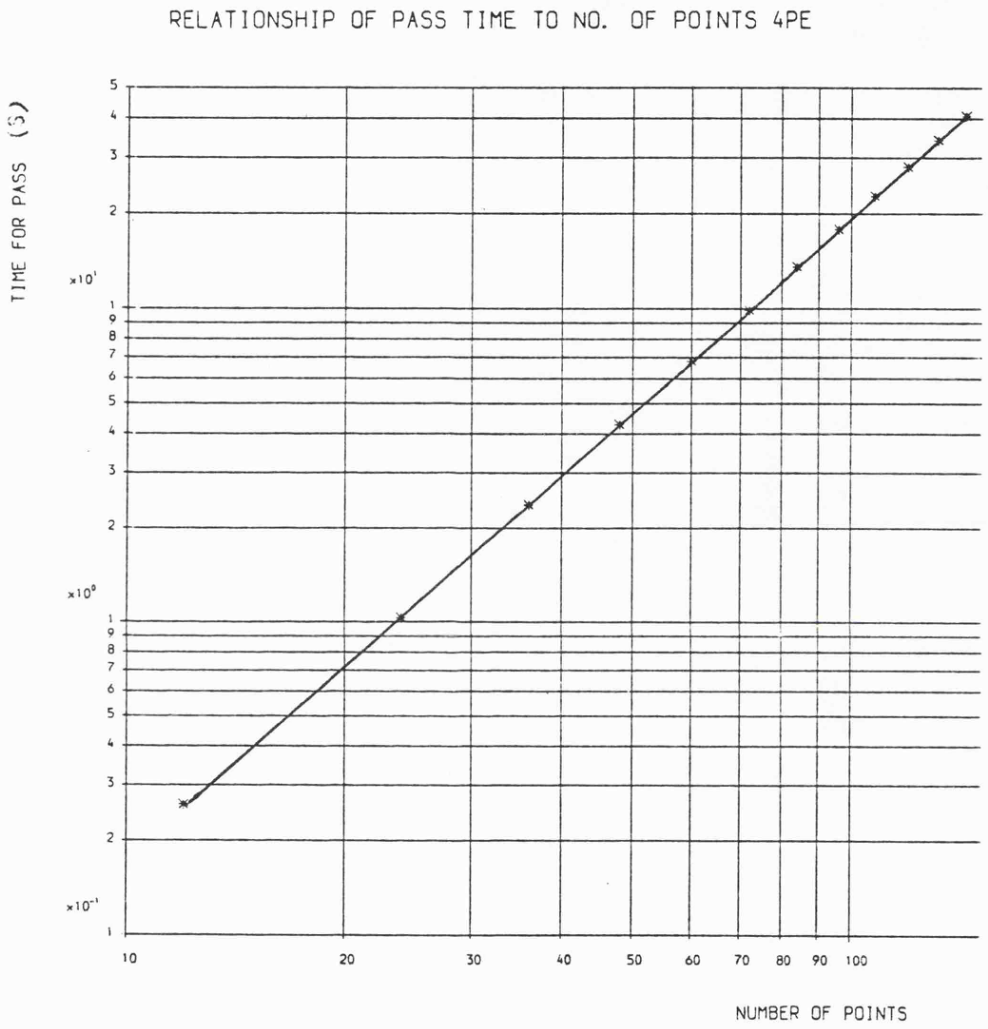


FIGURE F.2 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR 4PE

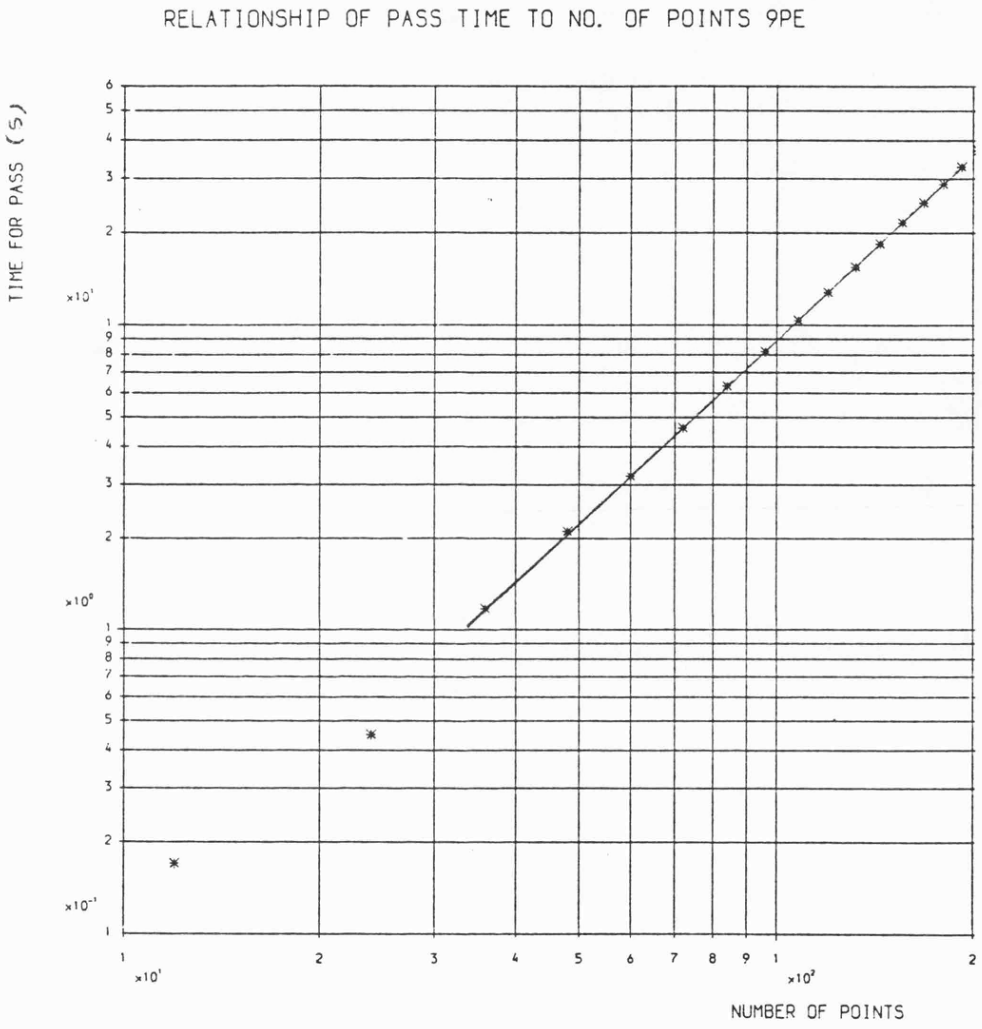


FIGURE F.3 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR 9PE

9PE $$t = \frac{p^2}{1123.699}$$

where t = average pass time
p = number of points along one edge of the problem mesh

Table F.3 shows the completed table of average pass times, with values extrapolated where indicated (using the above equations).

size	1PE	4PE	9PE	16PE
12x12	0.72	0.26	0.17	0.13
24x24	3.28	1.03	0.59	0.31
36x36	7.73	2.37	1.17	0.68
48x48	14.11	4.28	2.12	1.18
60x60	22.27	6.81	3.20	1.81
72x72	32.42	9.88	4.61	2.61
84x84	44.92	13.57	6.32	3.54
96x96	59.36	17.82	8.19	4.68
108x108	75.86	22.67	10.34	5.82
120x120	94.51	28.07	12.77	7.21
132x132	115.3	34.10	15.45	8.73
144x144	138.2	40.67	18.41	10.35
156x156	163.3	47.93	21.58	12.16
168x168	190.6	55.80	25.02	14.10
180x180	220.1	64.28	28.78	16.16
192x192	251.8	73.83	32.73	18.40
204x204	285.8	83.10	36.96	20.80
216x216	321.7	93.44	41.52	23.33
228x228	360.4	104.4	46.26	25.97
240x240	401.1	116.0	51.69	28.79
252x252	444.0	128.2	56.51	31.75
264x264	189.2	141.0	62.02	34.83
276x276	536.8	154.5	67.79	38.09
288x288	586.6	168.6	73.81	41.60

extrapolated area

Table F.3 Average pass times (s)

APPENDIX G

PROCESSING TESTS ON THE INDEPENDENT SYSTEM

APPENDIX G Comparison with independent computer based on the same processor

In order that the performance of the multiprocessor system could be judged not only with regards to one of the component PE of the machine, the same tasks as performed by the multiprocessor (in chapter 5, section 5.2.3) were performed by an independent microcomputer (the MICROBOX II based on the MC6809E microprocessor).

The times for 5 passes over various sizes of problem mesh were taken, the corresponding average pass times of which can be seen in table G.1.

mesh size	average pass time
12x12	0.562
24x24	2.668
36x36	6.318
48x48	11.544
60x60	18.346
72x72	26.794

Table G.1 Average pass times (s) of an independent system

Plotting this logarithmically (figure G.1) yields the following equation:

$$t = \frac{p^{2.1}}{296}$$

where t = average pass time

p = number of points along one edge of the problem mesh

This equation is used to extrapolate for the times which would be seen if sufficient memory was available to run the problem. The average pass times for the MICROBOX II can be seen in table G.2, with the extrapolated values as indicated.

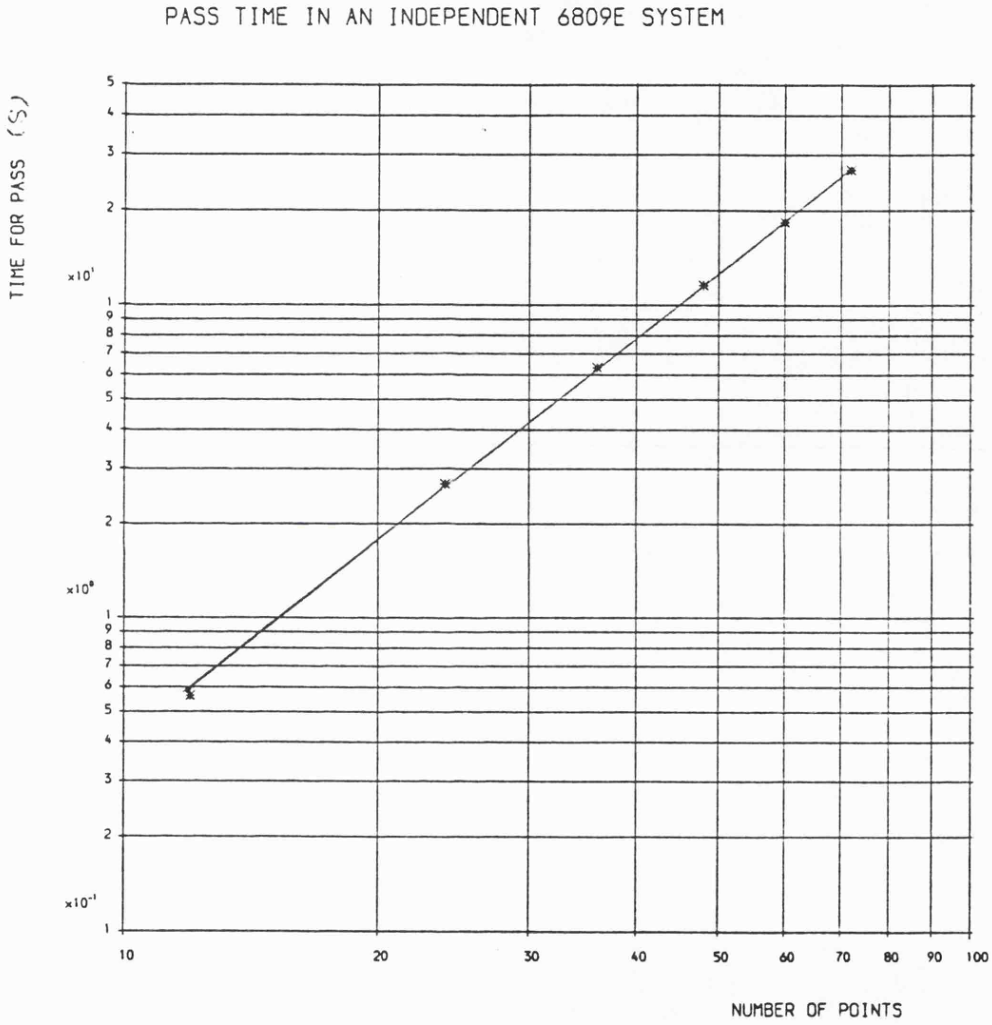


FIGURE G.1 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR THE MICROBOX II

grid size	average pass time
12x12	0.562
24x24	2.668
36x36	6.318
48x48	11.544
60x60	18.346
72x72	26.794
84x84	37.033
96x96	49.019
108x108	62.773
120x120	78.316
132x132	95.667
144x144	114.840
156x156	135.865
168x168	158.740
180x180	183.49
192x192	210.11
204x204	238.63
216x216	269.07
228x228	301.42
240x240	335.67
252x252	372.48
264x264	410.07
276x276	450.19
288x288	492.27

extrapolated area

Table G.2 Average pass times (s)

APPENDIX H

PROBLEM CONVERGENCE TESTS FOR THE MULTIPROCESSOR

APPENDIX H The time for a certain state of problem solution to be reached

Table H.1 shows the time taken for program 4 to reach the state where all the nodes within the problem have reached a certain condition (arbitrary but in this case 2/9th of the boundary value).

size	1PE	4PE	9PE	16PE
12x12	5.06	invalid data		
*15x15	*11.86	...	...	...
*20x20	*42.56	...	...	...
24x24	93.61	28.43	13.85	8.2
36x36	511.2	156.0	71.5	41.5
48x48	1679	508.3	232.9	131.7
*60x60	*4252	...	...	...
*96x96	...	...	...	*2188
*120x120	...	...	...	*5547
*144x144	...	...	*19359	...
*288x288	...	...	...	*181943
	extrapolation area			

TABLE H.1 The time (s) taken to a solution[1]

The logarithmic plots (seen in figures H.1, H.2 and H.3) of the data in table H.1 yeald the following equations from which the time to solution (t) can be found as a function of the problem size (p, where p is the number of points along one edge of the problem mesh):

---

[1]* Indicates extra values used to verify the equations, derived from logarithmic plots of this data, and used to extraplolate for run times of other problem sizes. In the case of the 16PE system the run time estimated using the extrapolation equation and the actual time for p=288 seen in table H.1 differed by 0.7% (which corresponds to 20 minutes over the total run time).

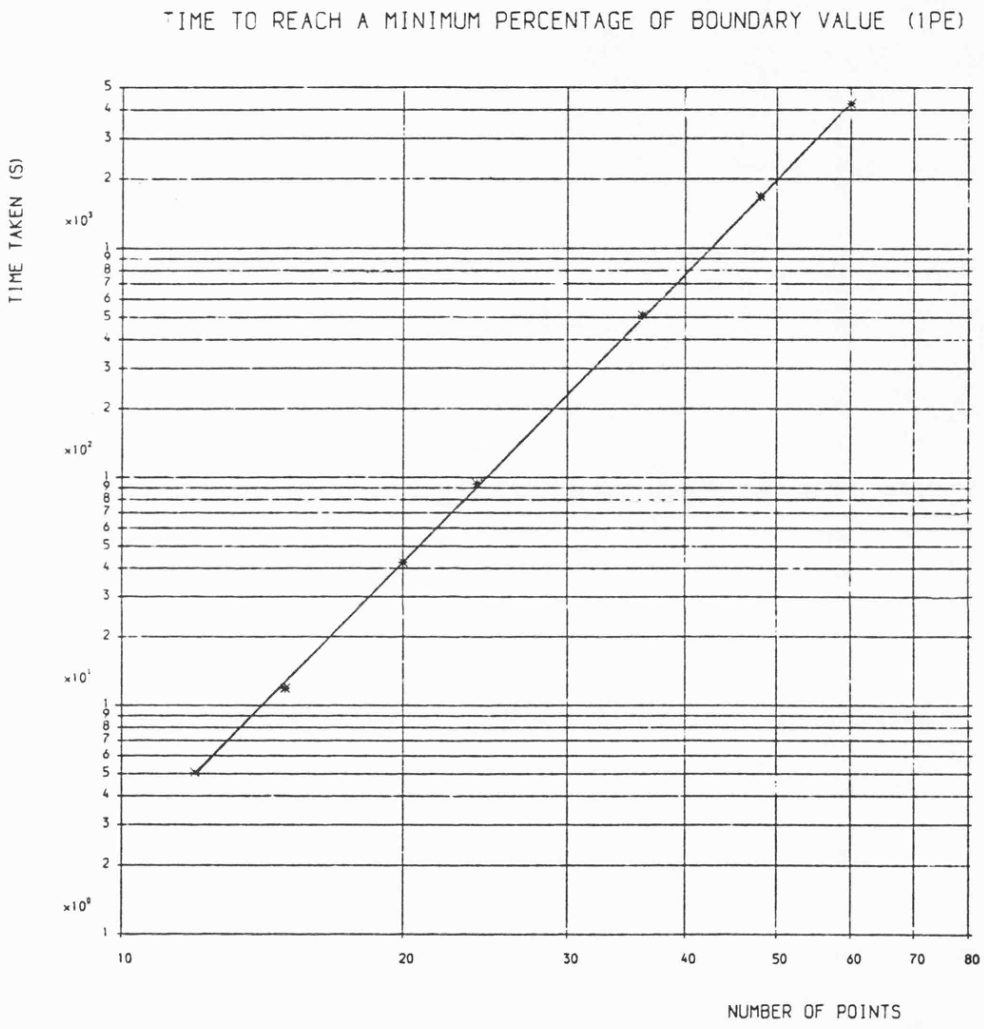


FIGURE H.1 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR 1PE

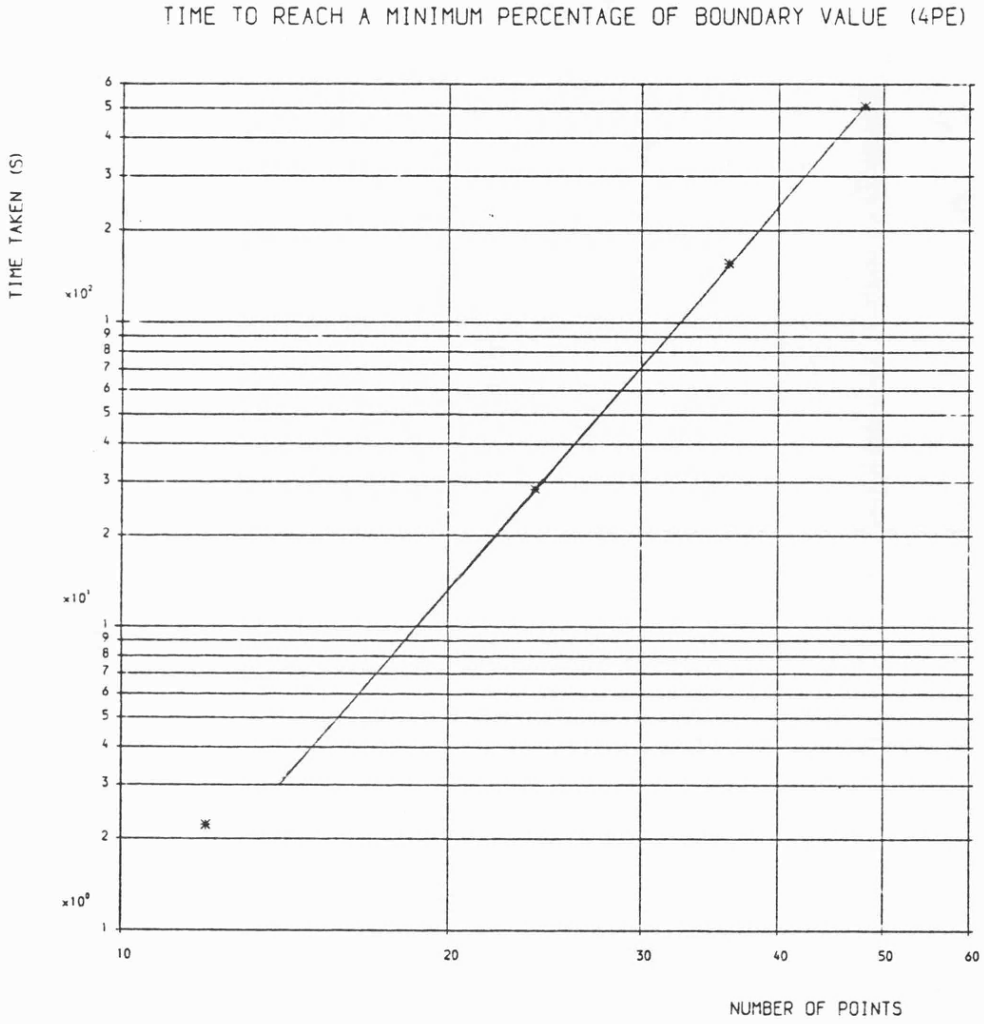


FIGURE H.2 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR 4PE

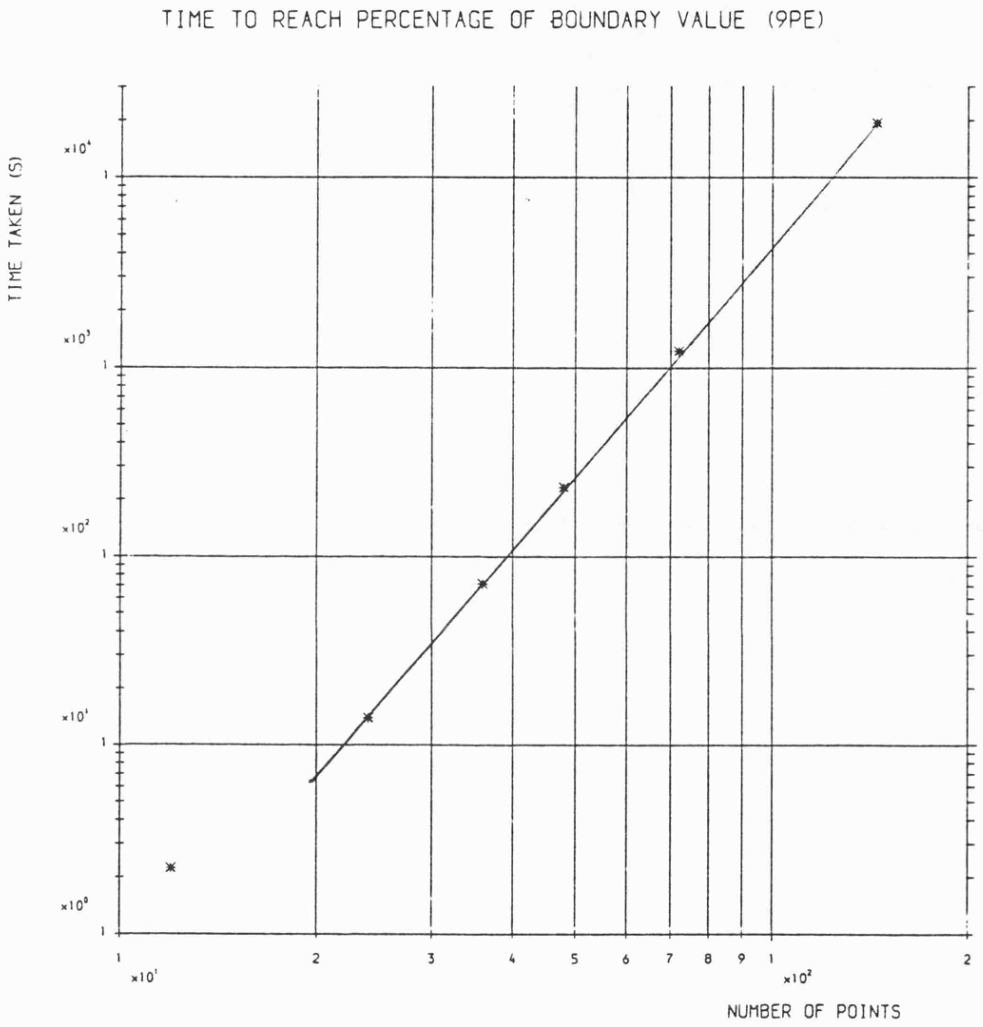


FIGURE H.3 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR 9PE

1PE $$t = \frac{p^{4.165}}{5995.15}$$

4PE $$t = \frac{p^{4.159}}{18618.47}$$

9PE $$t = \frac{p^{4.079}}{30823.18}$$

These equations enable predictions of the run times to be made for the memory restricted architectures within the 16PE system, which may be seen in Table H.2.

size	1PE	4PE	9PE	16PE
12x12	5.06	i n v a l i d		d a t a
24x24	93.61	28.43	13.85	8.2
36x36	511.2	156.0	71.5	41.5
48x48	1679.	508.3	232.9	131.7
72x72	9087	2743	1224	686.4
144x144	163029	49021	20695	11212
216x216	882523	264729	108172	57495
288x288	2924987	875922	349767	183143

extrapolated area

Table H.2 Time (s) to a set state of solution

APPENDIX I

PROBLEM CONVERGENCE TESTS FOR THE INDEPENDENT SYSTEM

APPENDIX I The times for an independent system to achieve solutions

The times taken for an independent 6809E system (the MICROBOX II) running a PL9 suggested construct of the solution algorithm, for problem sizes similar to those of run on the multiprocessor in appendix H, can be seen in Table I.1.

array size	time taken
12x12	4.30
24x24	79.05
36x36	435.4
48x48	1453.3

Table I.1 Time (s) to solution of an independent system

Figure I.1 shows these results logarithmically, from which the following equation relating the time taken to mesh size can be found to be:

$$\text{MICROBOX} \qquad t = \frac{p^{4.2}}{7989.32}$$

where t = time to problem solution

p = number of points along one edge of the problem mesh

This equation provides extrapolated results for other problem sizes, which may be seen in Table I.2.

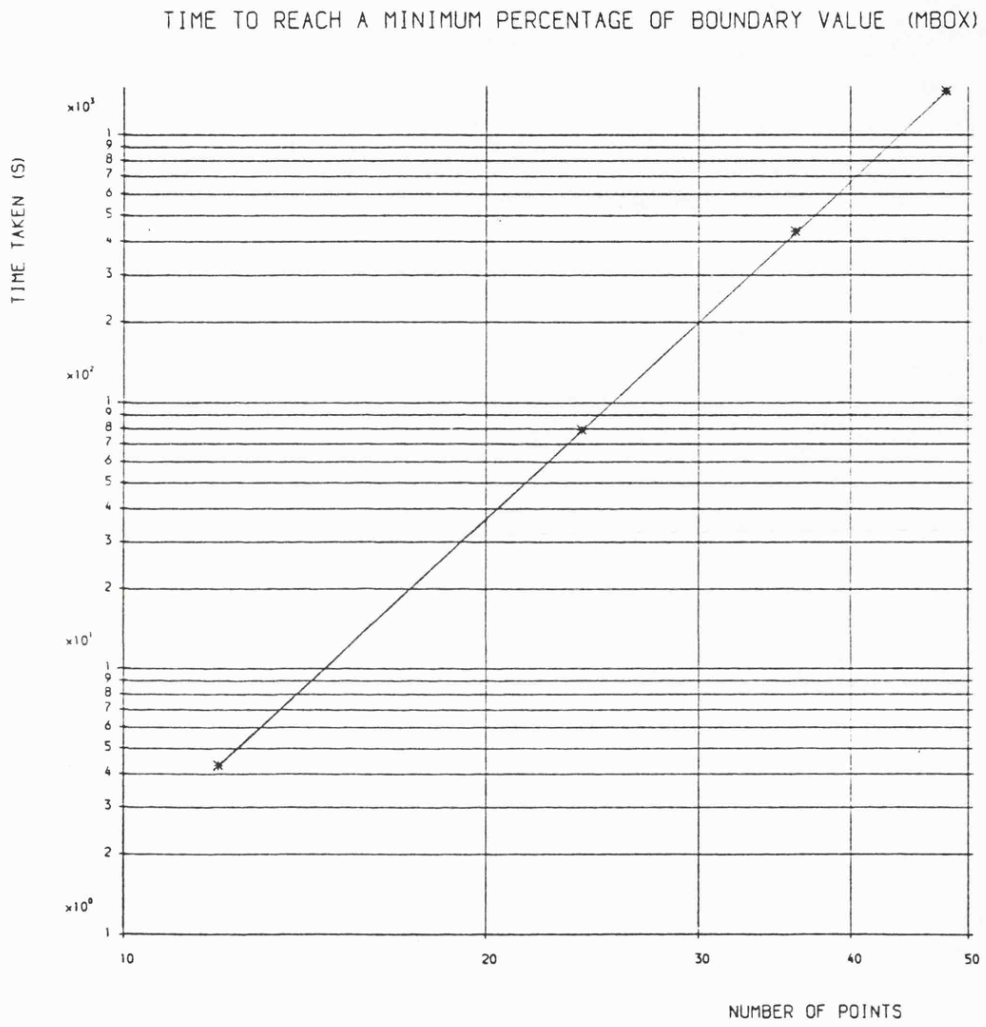


FIGURE I.1 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR THE MICROBOX II

array size	time taken
12x12	4.30
24x24	79.05
36x36	435.4
48X48	1459.9
72x72	7912.0
144x144	147722.5
216x216	812058.8
288x288	2720982.6

Table I.2 Times (s) to solution

APPENDIX J

OLIVETTI M24 PERFORMANCE

APPENDIX J OLIVETTI M24 PERFORMANCE

grid size	time taken (s)
24x24	14
36x36	70
48x48	215
60x60	537

This data has been plotted logarithmically in figure J.1. The logarithmic plot yealds the following relationship:

$$t = \frac{p^{3.9801}}{22246.886}$$

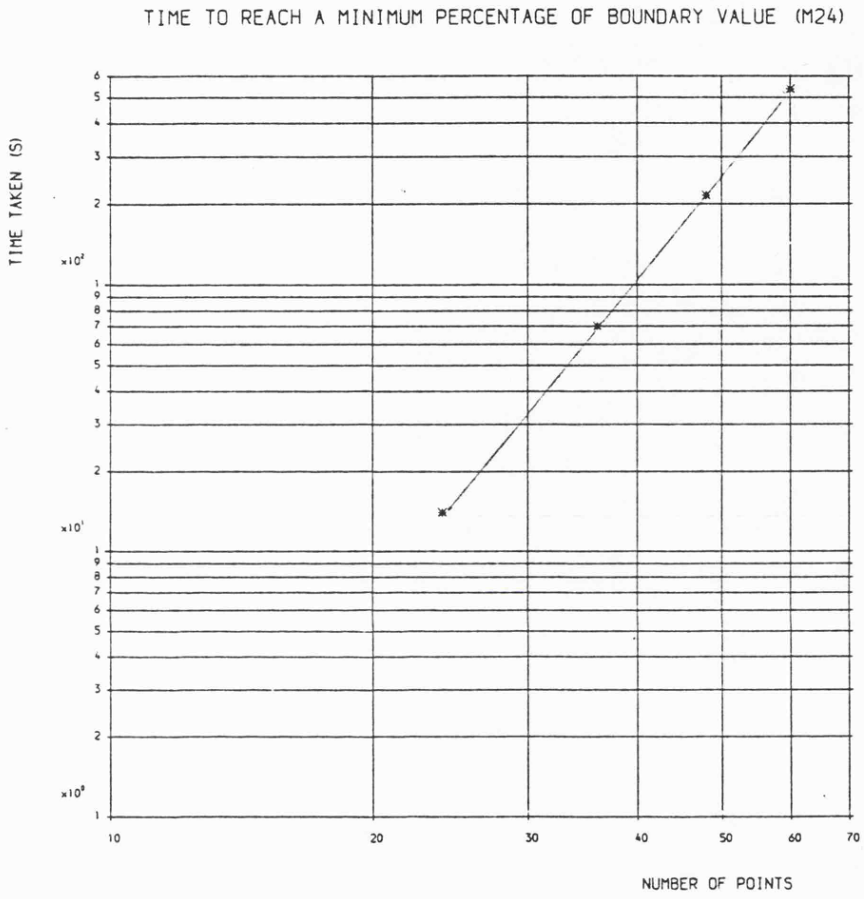


FIGURE J.1 LOGARITHMIC RELATIONSHIP OF THE AVERAGE PASS TIME TO PROBLEM SIZE FOR AN OLIVETTI M24

APPENDIX K

TABULATED SPEEDUPS OF THE SYSTEM SPEEDUP MODEL

APPENDIX K TABULATED SPEEDUPS OBTAINED FROM THE SYSTEM MODEL.

RPOWER 2.085071 BASE_POWER 2.085071
CONST 229.2100 BASE_CONST 229.2100

No. of PE	points on the edge of the grid	speedup
1	0	0.000000
1	12	1.000000
1	24	1.000000
1	36	1.000000
1	48	1.000000
1	60	1.000000
1	72	1.000000
1	84	1.000000
1	96	1.000000
1	108	1.000000
1	120	1.000000
1	132	1.000000
1	144	1.000000
1	156	1.000000
1	168	1.000000
1	180	1.000000
1	192	1.000000
1	204	1.000000
1	216	1.000000
1	228	1.000000
1	240	1.000000
1	252	1.000000
1	264	1.000000
1	276	1.000000
1	288	1.000000

RPOWER 2.030369 BASE_POWER 2.085071
CONST 559.0084 BASE_CONST 229.2100

No. of PE	points on the edge of the grid	speedup
4	0	0.000000
4	12	2.790485
4	24	2.897220
4	36	2.961700
4	48	3.008245
4	60	3.044850
4	72	3.075090
4	84	3.100891
4	96	3.123416
4	108	3.143421
4	120	3.161424
4	132	3.177799
4	144	3.192821
4	156	3.206703
4	168	3.219610
4	180	3.231673
4	192	3.242998
4	204	3.253672
4	216	3.263768
4	228	3.273347
4	240	3.282450
4	252	3.291152
4	264	3.299462
4	276	3.307421
4	288	3.315060

RPOWER 1.999817 BASE_POWER 2.035071
CONST 941.6878 BASE_CONST 229.2100

No. of PE	points on the edge of the grid	speedup
9	0	0.000000
9	12	5.077929
9	24	5.336939
9	36	5.576409
9	48	5.714966
9	60	5.824625
9	72	5.915968
9	84	5.994129
9	96	6.062756
9	108	6.123941
9	120	6.179197
9	132	6.229611
9	144	6.275995
9	156	6.319368
9	168	6.359019
9	180	6.396531
9	192	6.431923
9	204	6.465153
9	216	6.496734
9	228	6.526749
9	240	6.555253
9	252	6.582677
9	264	6.609836
9	276	6.633929
9	288	6.658043

RPOWER 1.978074 BASE_POWER 2.035071
CONST 1363.337 BASE_CONST 229.2100

No. of PE	points on the edge of the grid	speedup
16	0	0.000000
16	12	7.759587
16	24	8.366952
16	36	8.727487
16	48	9.000309
16	60	9.217777
16	72	9.399362
16	84	9.555477
16	96	9.693194
16	108	9.816114
16	120	9.927400
16	132	10.029157
16	144	10.122965
16	156	10.210022
16	168	10.291312
16	180	10.367965
16	192	10.439409
16	204	10.507342
16	216	10.571798
16	228	10.632125
16	240	10.691651
16	252	10.747612
16	264	10.801241
16	276	10.852737
16	288	10.902269

REFERENCES

REFERENCES

[1] Abramsky S.: "SECD-M a virtual machine for applicative multiprogramming", Computer Systems Laboratory, Queen Mary College, London.

[2] Abu-Surfad W. and Kwok A.Y.: "Performance prediction tools for CEDAR: A multiprocessor supercomputer", 12th International Conf. on Computer Architectures, June 1985, pp406-413: ICPR 1086.745 Vol. 13, No. 3, 1985.

[3] Ahmed H.M. et. al.: "Highly concurrent computing structures for matrix arithmetic and signal processing", **[I.E.E.E.] Computer**, Vol. 15, 1982, pp65-78.

[4] Anderson G.A. and Jenson E.D.: "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", **Computing Surveys**, 1975, Vol. 7, pp 197-213.

[5] Baer J.L.: "A Survey of Some Theoretical Aspects of Multiprocessing", **Computing Surveys**, 1973, Vol. 5, pp 31-80.

[6] Barlow R.H., Evans D.J. and Sharehchi J.: "Comparative Study of the Exploitation of Different Levels of Parallelism on Different Parallel Architectures", Proceedings of the 1982 International Conf. on Parallel Processing, IEEE Computer Society Press, pp34-40.

[7] Barlow R.H. and Evans D.J.: "Analysis of the performance of a dual-minicomputer parallel computer system", Proceedings Eurocomp, 1978, On Line Conference Uxbridge, pp 259-276.

[8] Barnes G.H. et. al.: "The ILLIAC IV Computer", IEEE Transactions on Computers, Vol. C-47, No. 8, Aug. 1968, pp 746-757.

[9] Barron I.M.: "The future of the Microprocessor", **Microelectronics**, June 1978, Vol. 8, Pt. No. 4, pp32-36.

[10] Baskett F. and Smith A.J.: "Interference in Multiprocessor Computer Systems with Interleaved Memory", Communications of the ACM, June 1976, Vol. 19, No. 6, pp327-334.

[11] Batcher K.E.: "Fiexible parallel processing and STARAN", 1972 WESTCON Tech. Papers, Session 1-Parallel processing systems, Sept. 1972, pp405-410.

[12] Batcher K.E.: "STARAN parallel processing system hardware", in Proc. AFIPS 1974 National Comp. Conf., Vol. 43, AFIPS Press, Montvale, N.J., 1974, pp405-410.

[13] Baxter A.C. and Holme J.: "A multiprocessor for Field Equations", Proc. 10th Annual Microcomputer Applications Workshop, Strathclyde University, 10-12 Sept. 1986.

[14] Bhuyan L.N. and Agrawal D.P.: "Applications of SIMD computers in signal processing", National Comp. Conf., 1982, pp 135-142.

[15] Bolognin A., Giulu D., Pelagotti F., Pirri F and Pogni F.: "Multiprocessor structures for microprocessors", **Software and microsystems**, Vol. 1, No. 7, Dec. 1982, pp 175-191.

[16] Bowra J.W. and Torng H.C.: "The modeling and design of multiple function unit processors", I.E.E.E. transactions on computers, March 1976, pp210-221.

[17] Brain S.: "The transputer - exploiting the opportunity of VLSI", **Electronic Product Design,** Dec. 1983, pp41-44.

[18] Brain S.: "Applying the transputer", **Electronic Product Design,** Jan. 1984, pp43-48.

[19] Burnett G. and Coffman E.G, Jn.: "A study of interleaved memory systems", AFIPS Conf. Proceedings Spring Joint Computer Conference, 1970, Vol. 36, pp 467-474.

[20] Carruthers J.R.: "Supercomputer design", **Computer Systems,** May 1979, pp33-37.

[21] Christ N.H. and Terrano A.E.: "A Micro-based Supercomputer", **Byte,** April 1986, pp145-160.

[22] Cichanowski G.W.: "Is there a RISC in SEC's future", INSPEC Journal paper 86C18194: DEC Prof. U.S.A, Dec. 1985, Vol. 4, Pt. 12, pp58-69.

[23] Coles R.W.: "The transputer, a component for the fifth generation", **Practical Electronics,** April 1984, pp26-31.

[24] Colon F.C, Clorioso R.M., Kohler W.H. and Dominic W.L.: "Coupling small computers for performance enhancement", Proceedings of the National Computer Conference, 1976, pp 755-764.

[25] Cox G.W. et.al.: "Interprocessor communication and process ispatching in the Intel 432", ACM trans. Comput. Systems, 1983, Vol. 1, pp45-63.

[26] Curry B.J.: "Language based architecture eases system design", **Computer Design,** Jan. 1984, pp127-136.

[27] Davis E.W.: "STARAN parallel processing system software", in Proc. AFIPS 1974 National Comp. Conf., Vol. 43, AFIPS Press, Montvale, N.J., 1974, pp405-410.

[28] Despain A.M. and Patterson: "X-Tree: A tree structured multiprocessor computer architecture", Proc. of the 5th annual symp. on comp. architectures. IEEE/ACM, 1978, pp144-151.

[29] Dettmer R.: "Chip architectures for parallel processing", **Electronics and Power,** March 1985, Vol. 31, No. 3, pp227-231.

[30] Dettmer R.: "Occam and the transputer", **Electronics and Power,** April 1985, Vol. 31, No. 4, pp283-287.

[31] Dew P.M.: "VLSI architectures for problems in numerical computation", taken from:- Paddon D.J. (ed.): "Super Computers and Parallel Computation", Instit. of Maths and I.T. Applications conf. series, No 1, Clarendon Press, Oxford, 1982, pp1-24.

[32] Dew P.M. et. al.: "Application of VLSI Devices to Computational Problems in the Gas industry", Report 163, University of Leeds.

[33] Love H.H.: "Reconfigurable parallel array systems, in designing and programming modern computers", Vol. 1, L.S.I Modular Comp. Systems (S. Kartashev et. al.), Prentice Hall, 1982, pp 33-64.

[34] Ducksbury P.G.: "The implementation of Price's (CRS) Algorithm on an ICL DAP", Numerical Optimization Centre, Technical Report 127.

[35] Enslow P.H.: "Multiprocessors and parallel processing", Computer, July 1977, pp64-70.

[36] Fagan M.: "Goodbye to the Silicon chip", **Practical Computing,** August 1987, Vol. 10, Issue 8, pp 74-76.

[37] Fay D.: "Working with OCCAM", **Microprocessors and Microsystems,** Jan./Feb. 1984, Vol. 8, Pt. No. 1, pp3-15.

[38] Flynn M.J.: "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, Vol. C-21, No. 9, pp 948-960.

[39] **Fortune** : "Reinventing the Computer", **Fortune,** March, 1984, pp 62-70.

[40] Fuller S., Ousterhout J., Raskin L., Rubinfield P. and Swan R.: "Multimicroprocessor, an overview and working example", Proc. I.E.E.E. 1978, Vol 66, pp216-228.

[41] Fung C.K.: "Cashe system design in the tightly coupled microprocessor system", Proc. National Computer Conf., 1976, pp81-87.

[42] Furber S.B. and Wilson A.R.: "The Acorn RISC machine - an architectural view", **I.E.E. Electronics and Power,** June 1987, Vol. 33, No 6, pp402-405.

[43] Grimsdale et. al.: "POLYPROC II - The university of Sussex Multiple Microprocessor System", Proc. 2nd International Conf. on Distributed Systems, Paris, April 1981, pp 95-104.

[44] Haynes L.S. et. al.: "A survey of highly parallel computing", **[I.E.E.E.] Computer**, Vol 15, Jan. 1982, pp9-35.

[45] Hennessy J.: "VLSI RISC Processors", VLSI Design, U.S.A., Vol. 6, No. 10, pp22-32: INSPEC Journal Paper 86 B21726.

[46] Hillis W.D.: "The Connection Machine", **Scientific American,** May 1987, pp 86-93

[47] Hitachi: "Semiconductor data book 8/16-bit microcomputer", Hitachi electronic components.

[48] Hoare C.A.: "OCCAM programming manual", published by Oxford University Press, 1984.

[49] Holme J. and Warrington E.M: "Switched memory in the MC6809E: A correction", Submitted for publication in **Microcomputers and microsystems.**

[50] Jesshope C.R.: "A Reconfigurable Processor Array for VLSI", from "Super Computers and Parallel Computation", Paddon D.J. (ed.): Instit. of Maths and I.T. Applications conf. series, No 1, Clarendon Press, Oxford, 1984, pp 35-40.

[51] Jesshope C.R.: "Optimal methods of applying transputers in large systems", Dept. of Electronics and Information Engineering, The University, Southampton, SO9 5NH.

[52] Kuck: "Parallel Processing Architectures", Proc. of the Singapore Conf. on Parallel Processing, 1975, pp15-39.

[53] Kung H.T.: "Advanced Course on VLSI Architecture", held at the University of Bristol, Treveaen ed..

[54] Kung H.T.: "Why Systolic Architectures", **[I.E.E.E.] Computer**, Jan. 1982, Vol 15, pp37-46.

[55] Kung S.Y. et. al.: "Wavefront Array Processor: Architecture Language and Application", Proc. of M.I.T. Conf. on Advanced Research in VLSI, (P. Penfield Jn. editor), Artech House, 1982, pp4-19.

[56] Lauer P.E. and Hemshere B.C.: "Project on a computer based enviroment for the design and analysis of highly parallel and distributed computing systems", Intrim progress report, ASM/112, June 1983.

[57] Leiserson C.E. and Saxe J.B.: "Optimizing synchronous systems", Proc. 22nd Symp. on Foundations of Computer Science, **I.E.E.E. Computer Society**, 1981, pp23-36.

[58] Malone S.: "DEC MICROVAX 2000, The mini mini", **Practical Computing**, June 1987, Vol. 10, Issue 6, pp42-43.

[59] Manner R. et. al.: "Design and realization of the large-scale multimicroprocessor system 'Heidelberg POLYP'", INSPEC Conf. Paper 85C15047, First International Conf. on Computers and Applications, Beijing, China, 20-22 June 1984, pp264-270.

[60] Mattos P.: "Applying the transputer", **I.E.E. Electronics and Power**, June 1987, Vol. 33, No 6, pp402-405.

[61] Mead C. and Conway L.: "Introduction to VLSI systems", Addison-Wesley.

[62] Mendez R. H.: "The Scalar Performance of Three Suptercomputers CRAY's X-MP/2, FUJITSU's VP-200 and NEC's SX-2", Lecture Notes in Engineering,Supercomputers and Fluid Dynamics, Proceedings of the First Nobeyama Workshop, Sept. 3-6, 1985, Springer Verlag, pp 148-158.

[63] Microbox II reference Manual, Microconcepts, 2 St. Stephens Road, Cheltenam, Glos..

[64] Ostlund N.S., Hibbard P.G. and Whiteside R.A.: "A Case Study in the Application of a Tightly Coupled Multiprocessor to Scientific Computations", in Parallel Computations, edited by G. Rodrigue, Academic Press, 1982.

[65] Perrenod S.C.: "The CRAY-2: The New Standard In Supercomputing", Lecture Notes in Engineering,Supercomputers and Fluid Dynamics, Proceedings of the First Nobeyama Workshop, Sept. 3-6, 1985, Springer Verlag, pp 174-183.

[66] Petre P.: "A Computer Chip", **Fortune**, 14 May 1984, Vol. 109, Pt. No. 10, p74.

[67] Purcell C.J.: "An Introduction to the ETA", Lecture Notes in Engineering,Supercomputers and Fluid Dynamics, Proceedings of the First Nobeyama Workshop, Sept. 3-6, 1985, Springer Verlag, pp 184-200.

[68] RAND: "The feasibility of a special-purpose computer to solve the of the Navier-Stokes equations", E.C. Gritton, W.S. King, I.E. Sutherland, R. Gains, C. Gazley Jn., C.E. Grosch, M.L. Juncosa and H.E. Peterson, Rand Corporation report R-2183-RC, June 1977.

[69] Reyling G.: "Performance and Control of Multiple Microprocessor Systems", **Computer Design**, March 1974, pp 81-86.

[70] Rudolf J.A.: "A production implementation of an associative array processor: STARAN", in Proc. AFIPS 1972 Fall Jt. Computer Conf., Vol. 41, Pt. 1, AFIPS Press, Montvale, N.J., 1972, pp229-241.

[71] Russell R.M.: "The CRAY-1 Computer System", Communications of the A.C.M., 1978, Vol. 21, Pt. 1, pp63-72.

[72] Seigel H.J. et. al.: "A survey of interconnecting methods", Proc. of the Nat. Comp. Conf., 1979, pp529-542.

[73] Searle B.C. and Freberg D.E.: "Microprocessor applications in multiple processor systems", Proc. of the National Computer Conf., 1976, pp749-753.

[74] Seitz C.L.: "Ensemble Architectures for VLSI: A Survey and Taxonomy", Proc. M.I.T. conf. on advanced research in V.L.S.I., (P. Penfield Jn. editor), Artech house, pp130-135.

[75] Siewiorek D. and Swarz R.S.: "The theory and practise of reliable system design", published by Digital Press, 1982.

[76] Siewiorek D. et. al.: "C.vmp: the architecture an implementation of a fault tolerant multiprocessor", INSPEC Conf. Paper 77C26907, 7th Annual Conf. on Fault-Tolerant Computing, U.S.A. 28-30 June 1977, pp37-43.

[77] S.E.R.C.: "Coordinated Program of Reasearch Distributed systems", S.E.R.C. Annual Report, Sept. 1982-1983.

[78] S.E.R.C.: "Coordinated Program of Reasearch Distributed systems 1977-1984", S.E.R.C. Final Report 1984.

[79] Shoja G.C. et. al.: "Some experiences of implementing the Ada concurrency facilities on a distributed multiprocessor computer system", **Software and Microsystems,** Vol. 1, No. 6, Oct. 1982, pp147-152.

[80] Snyder L.: "Introduction to the configurable highly parallel computer", **I.E.E.E. Computer Society,** 1982, Vol. 15, pp47-56.

[81] Speitz W.L.: "Microprocessor Networks", **Computer,** July 1977, pp64-70.

[82] Shaw et. al.: "The CLIP4 System", INSPEC journal paper issue 8741 87123409. Pattern Recognition Lett. (netherlands), Vol. 5, No. 1, pp71-79.

[83] Stone: "Parallel Computers", Introduction to Computer Architectures, SRA 1975, pp318-374.

[84] Taylor R.: "Survey of transputer applications", Inter-departmental Communication (Computer Studies-Engineering), University of Leicester, Oct. 1986.

[85] Thomas E.C.: "ITOS", Unpublished work, Dept. of Physics, University of Leicester.

[86] Warrington E.M. and Thomas E.C: "Switched memory for multi-processor MC6809E systems", **Microprocessors and Microsystems,** Vol. 9, No. 10, Dec. 1985, pp475-480.

[87] Watanabe T.: "NEC Supercomputer SX System", Lecture Notes in Engineering,Supercomputers and Fluid Dynamics, Proceedings of the First Nobeyama Workshop, Sept. 3-6, 1985, Springer Verlag, pp159-164.

[88] Weiman F.R. and Grosch C.E.: "Parallel Processing Research in Computer Science: Relevent to the design of a Navier-Stokes computer", Proc. of the 1977 International Conf. on Parallel Processing, Detroit, 23-26 Aug. 1977, pp175-182; INSPEC conf. paper 78C12487.

[89] Weissberger A.J: "Analysis of multiple-microprocessor system architectures", **Computer Design**, June 1977, pp 151-163.

[90] White F.M.: "Viscous fluid flow", Published by McGraw-Hill, 1974: ISBN_0-07-069710-8.

[91] Widdows L.C. Jn.: "The Minerva Multi-processor", 3rd Annual Symp. on Computer Architectures, Jan. 1976, pp34-39.

[92] WINDRUSH Micro Systems Ltd.: "PL/9 Reference Manual", Worstead Laboratories, North Walsham, Norfolk, NR28 9SA.

[93] WINDRUSH Micro Systems Ltd.: "PL/9 Programmers Reference", Worstead Laboratories, North Walsham, Norfolk, NR28 9SA.

[94] WINDRUSH Micro Systems Ltd.: "PL/9 User Guide", Worstead Laboratories, North Walsham, Norfolk, NR28 9SA.

[95] Witten I.H. and Cleary J.G.: "An introduction to the architecture of the IAPX 432", **Software and Microsystems**, 1983, No. 2, pp29-34.

[96] Wulf W.A. and Bell C.G.: "C.mmp-A multi-mini-processor", Proceedings Fall Joint Computer Conference, Dec. 1972, pp 765-777.

[97] Yau S.S. and Fung H.S.: "Associative Processor Architecture - A survey", **Computing Surveys**, Vol. 9, No. 1, March 1977, pp3-27.

PROGRAM LISTINGS

PROGRAM LISTING 1

LAPLACE FOR AN

OLIVETTI M24

```
      program lap
      dimension a(61,61)
      max=60
      do 1  i=1,max
        do 2  j=1,max
          a(i,j)=1.1
2       continue
1     continue

      do 3 i=1,max
        a(i,1)=9
        a(i,max)=9
        a(1,i)=9
        a(max,i)=9
3     continue


      print*,'go'
      read(*,*)g

4     continue

      all=1
      do 5 i=2,max-1
        do 6 j=2,max-1
          a(i,j)=( a(i-1,j)+a(i+1 ,j)+a(i,j+1)+a(i,j-1) )/4.0
          if(a(i,j).lt.2)  all=0
6       continue
5      continue

      print*,a(max/2,max/2)
      if (all.eq.0) goto 4

      print*,'      ****************************************'


      stop
      end

      subroutine oup(a)
      dimension a(50,50)
      max=60
      do 1 i=1,max
      do 2 j=1,max
      write(*,99) a(i,j)
2     continue
99    format(12f7.3)
1     continue
      return
      end
```

PROGRAM LISTING 2

LAPLACE FOR THE

MICROBOX II

```
0001 GLOBAL REAL ARRAY(6401);
0002 INCLUDE 0.TRUFALSE.DEF:
0003 INCLUDE 0.IOSUBS.LIB:
0004 INCLUDE 0.NUMCON.LIB:
0005
0006 PROCEDURE LAPLACE: INTEGER I.J.MAX.ITERATION:
0007                    BYTE    CH:
0008                    REAL    V1.V2.V3.V4.BOUNDARY:
0009    /* the initialise bit */
0010 MAX=12:
0011 repeat
0012    BOUNDARY=9:
0013    I=1:
0014    REPEAT
0015       J=1:
0016       REPEAT
0017          ARRAY(I+J*MAX)=0:
0018          J=J+1:
0019       UNTIL J=MAX:
0020       I=I+1:
0021    UNTIL I=MAX:
0022    /* the boundary bit */
0023    I=1:
0024    REPEAT:
0025       ARRAY(I+MAX)=BOUNDARY:
0026       ARRAY(MAX+I*MAX)=BOUNDARY:
0027       ARRAY(I+MAX*MAX)=BOUNDARY:
0028       ARRAY(1+I*MAX)=BOUNDARY:
0029       I=I+1:
0030    UNTIL I=MAX+1:
0031 printint(max); print("  "):
0032 print("  ?"); ch=getchar: crlf:
0033    /* iterations */
0034    ITERATION=0:
0035    REPEAT
0036       I=2;
0037       REPEAT
0038          J=2:
0039          REPEAT
0040             V1=ARRAY(I+(J+1)*MAX):
0041             V2=ARRAY(I+1+J*MAX);
0042             V3=ARRAY(I+(J-1)*MAX):
0043             V4=ARRAY(I-1+J*MAX);
0044             ARRAY(I+J*MAX)=(V1+V2+V3+V4)/4;
0045             J=J+1;
0046          UNTIL J=MAX:
0047          I=I+1;
0048       UNTIL I=MAX;
0049       ITERATION=ITERATION+1:
0050    UNTIL ITERATION=5:
0051 PUTCHAR(7);
0052
0053
0054
0055 I=1;
0056 REPEAT
0057    J=1;
0058    REPEAT
0059       PRNUM(FIX(ARRAY(I+J*MAX))); PRINT("  ");
0060       J=J+1:
0061    UNTIL J=MAX+1;
0062    CRLF;
0063    I=I+1;
0064 UNTIL I=MAX+1:
0065 CRLF:
0066 MAX=MAX+12:
0067 until max=84:
0068 CALL $0003:
0069 /EOF
```

PROGRAM LISTING 3

LAPLACE FOR THE

MULTIPROCESSOR

```
0001 /*
0002      ****************************************
0003      *                                      *
0004      *            [illegible]               *
0005      *                                      *
0006      ****************************************
0007
0008
0009 DATE ___ 6th March '87
0010
0011 */
0012
0013 constant          RAM_BASE    =    #0000,
0014                   NORTH       =    #6000,
0015                   EAST        =    #7000,
0016                   SOUTH       =    #8000,
0017                   WEST        =    #9000;
0018
0019 AT #EF12:REAL      trh,brh,blh,tlh: BYTE global_halt_flag;
0020          INTEGER MAX: REAL .array: BYTE data_type,local_id,pe_tag;
0021
0022 AT #A007:BYTE IRRQ;
0023 AT #FFFA:BYTE ID;
0024
0025
0026 include 0.trufalse.def;
0027 /* set origin for get_address so that ldr can use as well */
0028 ORIGIN #D000;
0029
0030 procedure get_address( integer i,j ): real .address;
0031    .address=RAM_BASE+i*4+j*MAX*4;
0032 IF PE_TAG=1 THEN
0033 BEGIN
0034    IF LOCAL_ID
0035    CASE 4 then /* .......... TLH PE .......... */
0036    begin
0037       if j<=1 .and i<=MAX-1 then .address=SOUTH+i*4+j*MAX*4;
0038       if i>=MAX-1  .and j>=1 then .address=EAST+(i-(MAX-1))*MAX*4+4*(j-1);
0039       if i=MAX-1 .and j=1 then .address=.brh;
0040    end;
0041    CASE 1 then /* .......... TRH PE .......... */
0042    begin
0043       if  i<=1 .and j>=1 then .address=WEST+i*MAX*4+4*(j-1);
0044       if  j<=1 .and i>=1 then .address=SOUTH+4*(i-1)+j*MAX*4;
0045       if i=1 .and j=1 then .address=.blh;
0046    end;
0047    CASE 3 then /* .......... BLH PE .......... */
0048    begin
0049       if j>=MAX-1 .and i<=MAX-1 then .address=NORTH+i*4+(j-(MAX-1))*MAX*4;
0050       if i>=MAX-1 .and j<=MAX-1 then .address=EAST+4*(i-(MAX-1))*MAX+4*j;
0051       if i=MAX-1 .and j=MAX-1 then .address=.trh;
0052    end;
0053    ELSE  /* .......... BRH PE .......... */
0054    begin
0055       if i<=1 .and j<=MAX-1 then .address=WEST+i*MAX*4+j*4;
0056       if j>=MAX-1 .and i>=1 then .address=NORTH+4*(i-1)+4*(j-(MAX-1))*MAX;
0057       if i=1 .and j=MAX-1 then .address=.tlh;
0058    end;
0059 END;
0060 IF PE_TAG=2 THEN
0061 BEGIN
0062    if local_id
0063 CASE 1 then
0064 begin
0065 if i<=1 .and j>=1 then .address=WEST+i*4*MAX+(j-1)*4;
0066 if j<=1 .and i>=1 .and i<=MAX then .address=SOUTH+4*(i-1)+j*MAX*4;
0067 if i>=MAX .and j>=1 then .address=EAST+4*(i-MAX)*MAX+4*(j-1);
0068 if i=1 .and j=1 then .address=.blh;
0069 if i=MAX .and j=1 then .address=.brh;
0070 end;
0071 CASE 2 then
0072 begin
0073 if j<=1 .and i>=1 then .address=SOUTH+4*(i-1)+j*4*MAX;
0074 if j>=MAX .and i>=1 then .address=NORTH+4*(i-1)+4*(j-MAX)*MAX;
0075 if i<=1 .and j>=1 .and j<=MAX then .address=WEST+4*i*MAX+4*(j-1);
0076 if i=1 .and j=1 then .address=.blh;
0077 if i=1 .and j=MAX then .address=.tlh;
0078 end;
0079 CASE 3 then
0080 begin
```

```
0081 if i<=1 .and j<=MAX-1 then .address=WEST+4*i*MAX+j*4;
0082 if i>=MAX .and j<=MAX-1 then .address=EAST+4*(i-MAX)*MAX+4*j;
0083 if i>=1 .and i<=MAX .and j>=MAX-1 then .address=NORTH+4*(i-1)+4*(j-(MAX-1)
)*MAX;
0084 if i=1 .and j=MAX-1 then .address=.tlh;
0085 if i=MAX .and j=MAX-1 then .address=.trh;
0086 end;
0087 ELSE
0088 begin
0089 if j<=1 .and i<=MAX-1 then .address=SOUTH+4*i+4*j*MAX;
0090 if j>=MAX .and i<=MAX-1 then .address=NORTH+4*i+4*(j-MAX)*MAX;
0091 if i>=MAX-1 .and j<=MAX .and j>=1 then .address=EAST+4*(i-(MAX-1))*MAX+4*(
j-1);
0092 if i=MAX-1 .and j=1 then .address=.brh;
0093 if i=MAX-1 .and j=MAX then .address=.trh;
0094 end
0095 END;
0096 IF PE_TAG=3 THEN
0097 BEGIN
0098 if i>=1 .and i<=MAX .and j>=MAX then
0099    .address=NORTH+4*(i-1)+4*(j-MAX)*MAX;
0100 if i>=1 .and i<=MAX .and j<=1 then
0101    .address=SOUTH+4*(i-1)+4*j*MAX;
0102 if i>=MAX .and j<=MAX .and j>=1 then
0103    .address=EAST+4*(i-MAX)*MAX+4*(j-1);
0104 if i<=1 .and j>=1 .and j<=MAX then
0105    .address=WEST+4*MAX*j+4*(j-1);
0106 if i=1 .and j=1 then .address=.blh;
0107 if i=MAX .and j=1 then .address=.brh;
0108 if i=1 .and j=MAX then .address=.tlh;
0109 if i=MAX .and j=MAX then .address=.trh;
0110 END;
0111 endproc  .address;
0112
0113 /* back to normal type origins */
0114 ORIGIN $B100;
0115
0116
0117 procedure copy(integer address): real .n, .e, .s, .w;
0118    if address=.brh then /* tlh */
0119    begin
0120       .s=SOUTH+(MAX-1)*4+MAX*4;
0121       s=brh;
0122       .e=EAST;
0123       e=brh;
0124    end;
0125    if address=.blh then /* trh */
0126    begin
0127       .w=WEST+MAX*4;
0128       w=blh;
0129       .s=SOUTH+4*MAX;
0130       s=blh;
0131    end;
0132    if address=.trh then /* blh */
0133    begin
0134       .n=NORTH+(MAX-1)*4;
0135       n=trh;
0136       .e=EAST+(MAX-1)*4;
0137       e=trh;
0138    end;
0139    if address=.tlh then /* brh */
0140    begin
0141       .w=WEST+MAX*4+4*(MAX-1);
0142       w=tlh;
0143       .n=NORTH;
0144       n=tlh;
0145    end;
0146 endproc;
0147
0148
0149
0150 procedure init_array: integer i,j,i_start,i_finish,j_start,j_finish;
0151                      real .add;
0152 /* ELEMENT_POS=PE_TAG    FROM SOURCE PROCESSOR */
0153 /* ELEMENT_TYPE=LOCAL_ID FROM SOURCE PROCESSOR */
0154 IF PE_TAG
0155    CASE 1 THEN
0156       BEGIN
0157          if local_id=1 .or local_id=2 then i_start=1
0158                                          else i_start=0;
0159          if local_id=4 .or local_id=3 then i_finish=MAX
0160                                          else i_finish=MAX+1;
```

```
0161          if local_id=4 .or local_id=1 then j_start=1
0162                                            else j_start=0;
0163          if local_id=4 .or local_id=1 then i_finish=MAX+1
0164                                            else j_finish=MAX;
0165       END;
0166    CASE 2 THEN
0167       BEGIN
0168          if local_id
0169             CASE 4 then
0170                begin
0171                   i_start=0;
0172                   i_finish=MAX;
0173                   j_start=1;
0174                   j_finish=MAX+1;
0175                end;
0176             CASE 3 then
0177                begin
0178                i_start=1;
0179                   i_finish=MAX+1;
0180                   j_start=0;
0181                j_finish=MAX;
0182                end;
0183             ELSE
0184                begin
0185                   i_start=1;
0186                   i_finish=MAX+1;
0187                   j_start=1;
0188                   j_finish=MAX+1;
0189                end;
0190       END;
0191    CASE 3 THEN
0192       BEGIN
0193          i_start=1;
0194          i_finish=MAX+1;
0195          j_start=1;
0196          j_finish=MAX+1;
0197       END;
0198    ELSE
0199       BEGIN
0200          i_start=1;
0201          i_finish=MAX+1;
0202          j_start=1;
0203          j_finish=MAX+1;
0204       END;
0205    i=i_start;
0206    repeat
0207       j=j_start;
0208       repeat
0209          .add=get_address(i,j);
0210 ADD=1.1;
0211 if .add=.trh .or .add=.brh .or .add=.blh .or .add=.tlh then copy(.add);
0212          j=j+1;
0213       until j=j_finish;
0214       i=i+1;
0215    until i=i_finish;
0216 endproc;
0217
0218 procedure update( real a,b,c,d ): real average;
0219    average=(a+b+c+d)/4;
0220 endproc real average;
0221
0222
0223 procedure init_vals: real .res,.top_value,.rhs_value,.bot_value,
0224                          .lhs_value,local_top,local_bot,local_lhs,local_rhs;
0225                       integer i, .local_max:byte .loc_tag, .loc_loc_id;
0226    .local_max=EAST;
0227    .loc_tag=EAST+2;
0228    .loc_loc_id=EAST+3;
0229    .top_value=EAST+4;
0230    .rhs_value=EAST+8;
0231    .bot_value=EAST+12;
0232    .lhs_value=EAST+16;
0233    MAX=local_max;
0234    pe_tag=loc_tag;
0235    local_id=loc_loc_id;
0236    local_top=top_value;
0237    local_rhs=rhs_value;
0238    local_bot=bot_value;
0239    local_lhs=lhs_value;
0240    init_array; /* doing it here stops destruction of init data */
```

```
0241 IF PE_TAG=1 THEN
0242 BEGIN
0243    if local_id=4 then /* .......... TLH PE .......... */
0244    begin
0245       i=0;
0246       repeat
0247          .res=get_address(0,integer(i+1));
0248          res=local_lhs;
0249          .res=get_address(i,MAX);
0250          res=local_top;
0251          i=i+1;
0252       until i=MAX;
0253    end;
0254    if local_id=3 then /* .......... BLH PE .......... */
0255    begin
0256       i=0;
0257       repeat
0258          .res=get_address(0,i);
0259          res=local_lhs;
0260          .res=get_address(i,0);
0261          res=local_bot;
0262          i=i+1;
0263       until i=MAX;
0264    end;
0265    if local_id=1 then /* .......... TRH PE .......... */
0266    begin
0267       i=1;
0268       repeat
0269          .res=get_address(i,MAX);
0270          res=local_top;
0271          .res=get_address(MAX,i);
0272          res=local_rhs;
0273          i=i+1;
0274       until i=MAX+1;
0275    end;
0276    if local_id=2 then /* .......... BRH PE .......... */
0277    begin
0278       i=1;
0279       repeat
0280          .res=get_address(i,0);
0281          res=local_bot;
0282          .res=get_address(MAX,i);
0283          res=local_rhs;
0284          i=i+1;
0285       until i=MAX+1;
0286    end;
0287 END;
0288 IF PE_TAG=4 THEN
0289 BEGIN
0290    i=1;
0291    repeat
0292       .res=get_address(i,1);
0293       res=local_bot;
0294       .res=get_address(MAX,i);
0295       res=local_rhs;
0296       .res=get_address(i,MAX);
0297       res=local_top;
0298       .res=get_address(1,i);
0299       res=local_lhs;
0300       i=i+1;
0301    until i=MAX+1;
0302 END;
0303 IF PE_TAG=2 THEN
0304 BEGIN
0305 if local_id
0306    CASE 1 then /* top */
0307       begin
0308          i=1;
0309          repeat
0310             .res=get_address(i,MAX);
0311             res=local_top;
0312             i=i+1;
0313          until i=MAX+1;
0314       end;
0315    CASE 2 then /* rhs */
0316       begin
0317          i=1
0318          repeat
0319             .res=get_address(MAX,i);
0320             res=local_lhs;
```

```
0321            i=i+1;
0322         until i=MAX+1;
0323      end;
0324   CASE 3 then /* bottom */
0325      begin
0326         i=1;
0327         repeat
0328            .res=get_address(i,0);
0329            res=local_bot;
0330            i=i+1;
0331         until i=MAX+1;
0332      end;
0333   ELSE /* lhs */
0334      begin
0335         i=1;
0336         repeat
0337            .res=get_address(0,i);
0338            res=local_lhs;
0339            i=i+1;
0340         until i=MAX+1;
0341      end;
0342 END;
0343 endproc;
0344
0345 procedure iterate: real .el1..el2..el3,.el4..res: integer i,j,
0346                    i_start,i_finish,j_start,j_finish;
0347 /* a bit of thought and this is my new attempt */
0348 IF PE_TAG=1 THEN
0349 BEGIN
0350 i_start=1;
0351 i_finish=MAX;
0352 j_start=1;
0353 j_finish=MAX;
0354 END;
0355 IF PE_TAG=4 THEN
0356 BEGIN
0357    i_start=2;
0358    i_finish=MAX;
0359    j_start=2;
0360    j_finish=MAX;
0361 END;
0362 IF PE_TAG=2 THEN
0363 BEGIN
0364    if local_id=4 .or local_id=2 then
0365       begin
0366          i_start=1;
0367          i_finish=MAX;
0368          j_start=1;
0369          j_finish=MAX+1;
0370       end;
0371    else
0372       begin
0373          i_start=1;
0374          i_finish=MAX+1;
0375          j_start=1;
0376          j_finish=MAX;
0377       end;
0378 END;
0379 IF PE_TAG=3 THEN
0380 BEGIN
0381    i_start=1;
0382    i_finish=MAX+1;
0383    j_start=1;
0384    j_finish=MAX+1;
0385 END;
0386   i=i_start;
0387    repeat
0388       j=j_start;
0389       repeat
0390          .el1=get_address(i,integer(j+1));
0391          .el2=get_address(integer(i+1),j);
0392          .el3=get_address(i,integer(j-1));
0393          .el4=get_address(integer(i-1),j);
0394          .res=get_address(i,j);
0395          res=update(el1,el2,el3,el4);
0396          if .res=.trh .or .res=.blh .or .res=.tlh .or .res=.brh
0397          then copy(.res);
0398 /*
0399 .el1=get_address(i,MAX+1);
0400 .el2=get_address(MAX+1,j);
```

```
0401 .el3=get_address(i,0);
0402 .el4=get_address(0,j);
0403 if el1<>9 then gen #13;
0404 if el2<>9 then gen #13;
0405 if el3<>9 then gen #13;
0406 if el4<>9 then gen #13;
0407 */
0408            j=j+1;
0409        until j=j_finish;
0410        i=i+1;
0411    until i=i_finish;
0412 endproc;
0413
0414 procedure terminate_program: integer dummy;
0415    gen #13;
0416    repeat
0417        dummy=dummy;
0418    forever;
0419 endproc;
0420
0421 /* ORIGIN AT B000 FOR FAST START LOCATION */
0422 ORIGIN #B000;
0423 procedure main: INTEGER COUNT;
0424    COUNT=0;
0425    ccr=ccr and #ef; /* keeps pe recognising irqs */
0426    init_vals;
0427 REPEAT
0428    ITERATE;
0429    COUNT=COUNT+1;
0430 FOREVER;
0431 terminate_program;
0432 /EOF
    #
```

PROGRAM LISTING 4

OPTIMISED 'GET_ADDRESS'

PROCEDURE

THE INCLUDED FILE: GET_ADDD.PL9

```
0001 procedure get_address( integer i,j,k ): integer address: byte la,lb,lm,T1,
T2,lmm;
0002 la=byte(i);
0003 lb=byte(j);
0004 lm=byte(MAX);
0005 lmm=lm-1;
0006 address=$0000;
0007 IF PE_TAG=1 THEN
0008 BEGIN
0009    IF LOCAL_ID
0010    CASE 4 then /* ......... TLH PE ......... */
0011    begin
0012       if lb<=1 .and la<=lmm then
0013       address=SOUTH+ shift( i,2 );
0014       if lb=1 .and la<=lmm then
0015       address=address+shift(MAX,2);
0016 if address=$0000 then
0017 begin
0018       if la>=lmm  .and lb>=1 then
0019       address=EAST+ shift( (j-1),2 );
0020       if la=lm .and lb>=1 then
0021       address=address+shift(MAX,2);
0022 end;
0023       if la=lmm .and lb=1 then address=.brh;
0024    end;
0025    CASE 1 then /* ......... TRH PE ......... */
0026    begin
0027       if  la<=1 .and lb>=1 then
0028       address=WEST+ shift( (j-1),2 );
0029       if la=1 .and lb>=1 then
0030       address=address+ shift( MAX,2 );
0031 if address=$0000 then
0032 begin
0033       if  lb<=1 .and la>=1 then
0034       address=SOUTH+ shift( (i-1),2 );
0035       if lb=1 .and la>=1 then
0036       address=address+ shift( MAX,2 );
0037 end;
0038       if la=1 .and lb=1 then address=.blh;
0039    end;
0040    CASE 3 then /* ......... BLH PE ......... */
0041    begin
0042       if lb>=lmm .and la<=lmm then
0043       address=NORTH+ shift( i,2 );
0044       if lb=lm .and la<=lmm then
0045       address=address+ shift( MAX,2 );
0046 if address=$0000 then
0047 begin
0048       if la>=lmm .and lb<=lmm then
0049       address=EAST+ shift( j,2 );
0050       if la=lm .and lb<=lmm then
0051       address=address+ shift( MAX,2 );
0052 end;
0053       if la=lmm .and lb=lmm then address=.trh;
0054    end;
0055    ELSE        /* ......... BRH PE ......... */
0056    begin
0057       if la<=1 .and lb<=lmm then
0058       address=WEST+ shift( j,2 );
0059       if la=1 .and lb<=lmm then
0060       address=address+ shift( MAX,2 );
0061 if address=$0000 then
0062 begin
0063       if lb>=lmm .and la>=1 then
0064       address=NORTH+ shift( (i-1),2 );
0065       if lb=lm .and la>=1 then
0066       address=address+ shift( MAX,2 );
0067 end;
0068       if la=1 .and lb=lmm then address=.tlh;
0069    end;
0070 END;
0071 IF PE_TAG=2 THEN
0072 BEGIN
0073    if local_id
0074    CASE 1 then
0075    begin
0076
0077 if lb>=1 then
0078 begin
0079       if la<=1 then
0080       address=WEST + shift( (j-1),2 );
```

```
0081        if la=1 then
0082        address=address+shift(MAX,2):
0083        if la>=lm then
0084        address=EAST + shift( (j-1),2 ):
0085        if la=lm+1 then
0086        address=address+shift(MAX,2):
0087 end:
0088
0089 if la>=1 .and la<=lm then
0090 begin
0091        if lb<=1  then
0092        address=SOUTH + shift( (i-1),2 ):
0093        if lb=1  then
0094        address=address+shift(MAX,2):
0095 end:
0096
0097        if la=1 .and lb=1 then address=.blh:
0098        if la=lm .and lb=1 then address=.brh:
0099    end:
0100    CASE 2 then
0101    begin
0102
0103 if la>=1 then
0104 begin
0105        if lb<=1 then
0106        address=SOUTH + shift( (i-1),2 ):
0107        if lb=1 then
0108        address=address+shift(MAX,2):
0109        if lb>=lm then
0110        address=NORTH + shift( (i-1),2 ):
0111        if lb=lm+1 then
0112        address=address+shift(MAX,2):
0113 end;
0114
0115 if lb>=1 .and lb<=lm then
0116 begin
0117        if la<=1 then
0118        address=WEST + shift( (j-1),2 ):
0119        if la=1 then
0120        address=address+shift(MAX,2):
0121 end:
0122
0123        if la=1 .and lb=1 then address=.blh:
0124        if la=1 .and lb=lm then address=.tlh:
0125    end:
0126    CASE 3 then
0127    begin
0128
0129 if lb<=lmm then
0130 begin
0131        if la<=1 then
0132        address=WEST + shift( j,2 ):
0133        if la=1 then
0134        address=address+shift(MAX,2):
0135        if la>=lm then
0136        address=EAST + shift( j,2 ):
0137        if la=lm+1 then
0138        address=address+shift(MAX,2):
0139 end;
0140
0141 if la>=1 .and la<=lm then
0142 begin
0143        if lb>=lmm then
0144        address=NORTH+ shift( (i-1),2 ):
0145        if lb=lm then
0146        address=address+shift(MAX,2):
0147 end;
0148
0149        if la=1 .and lb=lmm then address=.tlh:
0150        if la=lm .and lb=lmm then address=.trh:
0151    end:
0152    ELSE
0153    begin
0154
0155 if la<=lmm then
0156 begin
0157        if lb<=1 then
0158        address=SOUTH + shift( i,2 ):
0159        if lb=1 then
0160        address=address+shift(MAX,2):
```

```
0161        if lb>=lm then
0162        address=NORTH + shift( i,2 ):
0163        if lb=lm+1 then
0164        address=address+shift(MAX,2):
0165 end;
0166
0167 if lb<=lm .and lb>=1 then
0168 begin
0169        if la>=lmm then
0170        address=EAST + shift( (j-1),2 ):
0171        if la=lm then
0172        address=address+shift(MAX,2):
0173 end;
0174
0175        if la=lmm .and lb=1 then address=.brh:
0176        if la=lmm .and lb=lm then address=.trh:
0177    end
0178 END;
0179 IF PE_TAG=3 THEN
0180 BEGIN
0181
0182 if la>=1 .and la<=lm then
0183 begin
0184        if lb>=lm then
0185           address=NORTH + shift( (i-1),2 ):
0186        if lb=lm+1 then
0187           address=address+shift(MAX,2):
0188
0189        if lb<=1 then
0190           address=SOUTH + shift( (i-1),2 ):
0191        if lb=1 then
0192           address=address+shift(MAX,2):
0193 end;
0194
0195 if lb>=1 .and lb<=lm then
0196 begin
0197        if la>=lm then
0198           address=EAST + shift( (j-1),2 ):
0199        if la=lm+1 then
0200           address=address+shift(MAX,2):
0201
0202        if la<=1 then
0203           address=WEST + shift( (j-1),2 ):
0204        if la=1 then
0205           address=address+shift(MAX,2):
0206 end;
0207
0208 if address<>$0000 then
0209 begin
0210 if la=1 then
0211 begin
0212    if lb=1 then address=.blh:
0213    if lb=lm then address=.tlh:
0214 end;
0215 if la=lm then
0216 begin
0217    if lb=1 then address=.brh:
0218    if lb=lm then address=.trh;
0219 end;
0220 end;
0221 END;
0222 if address=$0000 then address=MAX*shift(j,2)+shift(i,2);
0223 endproc  address;
0224 /EOF
    #
```

PROGRAM LISTING 5

LAPLACE FOR THE MULTIPROCESSOR

CONVERGENCE TESTS

```
0001 /*
0002      ***********************************
0003      *                                 *
0004      *          LAPLACE PROBLEM        *
0005      *                                 *
0006      ***********************************
0007
0008
0009 DATE ___ 6th March '87
0010
0011 */
0012
0013 constant          RAM_BASE    =    $0000,
0014                   NORTH       =    $6000,
0015                   EAST        =    $7000,
0016                   SOUTH       =    $8000,
0017                   WEST        =    $9000:
0018
0019 AT $EF12:REAL      trh,brh,blh,tlh: BYTE global_halt_flag:
0020          INTEGER MAX: REAL .array: BYTE data_type,local_id,pe_tag:
0021          BYTE problem_type_flag: integer swi_vector:
0022
0023 /* variables used purely by this program */
0024 AT $EE00: integer i_start_iteration, j_start_iteration,
0025                   i_finish_iteration, j_finish_iteration:
0026
0027 include 0.trufalse.def:
0028 maths=$d100:
0029
0030 /* set origin for get_address so that ldr can use as well */
0031 ORIGIN $C000:
0032
0033 INCLUDE 1.GET_ADDd.PL9:
0034 procedure swi_john:
0035 endproc:
0036
0037 origin $b100:
0038
0039 procedure copy(integer address): real .n, .e, .s, .w:
0040    if address=.brh then /* tlh */
0041    begin
0042       .s=SOUTH + shift( MAX,3 ) - 4:
0043       s=brh:
0044       .e=EAST:
0045       e=brh:
0046    end:
0047    if address=.blh then /* trh */
0048    begin
0049       .w=WEST+ shift( MAX,2 ):
0050       w=blh:
0051       .s=SOUTH+ shift( MAX,2 ):
0052       s=blh:
0053    end:
0054    if address=.trh then /* blh */
0055    begin
0056       .n=NORTH+ shift( (MAX-1),2 ):
0057       n=trh:
0058       .e=EAST+ shift( (MAX-1),2 ):
0059       e=trh:
0060    end:
0061    if address=.tlh then /* brh */
0062    begin
0063       .w=WEST+ shift( MAX,3 ) - 4:
0064       w=tlh:
0065       .n=NORTH:
0066       n=tlh:
0067    end:
0068 endproc:
0069
0070
0071
0072 procedure init_array: integer i,j,i_start,i_finish,j_start,j_finish:
0073                       real .add:
0074 INCLUDE 1.LP_ARRAY.PL9:
0075 INCLUDE 1.LP_ITER.PL9:
0076    i=i_start:
0077    repeat
0078       j=j_start:
0079       repeat
0080          .add=get_address(i,j,0):
```

```
0081 ADD=1.1:
0082 if .add=.trh .or .add=.brh .or .add=.blh .or .add=.tlh then copy(.add);
0083          j=j+1:
0084       until j=j_finish:
0085       i=i+1;
0086    until i=i_finish:
0087 endproc:
0088
0089 procedure update( real a,b,c,d ): real average:
0090    average=(a+b+c+d)/4:
0091 endproc real average:
0092
0093
0094 procedure init_vals: real .res..top_value,.rhs_value..bot_value,
0095                          .lhs_value,local_top,local_bot,local_lhs,local_rhs:
0096                        integer i, .local_max:byte .loc_tag, .loc_loc_id:
0097    .local_max=EAST;
0098    .loc_tag=EAST+2:
0099    .loc_loc_id=EAST+3:
0100    .top_value=EAST+4:
0101    .rhs_value=EAST+8:
0102    .bot_value=EAST+12:
0103    .lhs_value=EAST+16:
0104    MAX=local_max:
0105    pe_tag=loc_tag:
0106    local_id=loc_loc_id:
0107    local_top=top_value:
0108    local_rhs=rhs_value:
0109    local_bot=bot_value:
0110    local_lhs=lhs_value:
0111    init_array: /* doing it here stops destruction of init data */
0112 INCLUDE 1.LP_VALS.PL9:
0113 gen #13:
0114 endproc:
0115
0116 procedure iterate: real .el1,.el2,.el3,.el4,.res: integer i,j:
0117                        byte all_above_flag:
0118 all_above_flag=true:
0119   i=i_start_iteration:
0120    repeat
0121       j=j_start_iteration:
0122       repeat
0123          .el1=get_address(i,integer(j+1),0):
0124          .el2=get_address(integer(i+1),j,0):
0125          .el3=get_address(i,integer(j-1),0):
0126          .el4=get_address(integer(i-1),j,0):
0127          .res=get_address(i,j,0):
0128          res=update(el1,el2,el3,el4):
0129 if res<2 then all_above_flag=false:
0130          if .res=.trh .or .res=.blh .or .res=.tlh .or .res=.brh
0131          then copy(.res):
0132          j=j+1:
0133       until j=j_finish_iteration:
0134       i=i+1:
0135    until i=i_finish_iteration:
0136 if all_above_flag=true then gen #31:
0137 endproc:
0138
0139 procedure terminate_program: integer dummy:
0140    gen #13:
0141    repeat
0142       dummy=dummy:
0143    forever:
0144 endproc:
0145
0146 /* ORIGIN AT B000 FOR EASY START LOCATION */
0147 ORIGIN #B000:
0148 procedure main: INTEGER COUNT:
0149 swi_vector=.swi_john:
0150    COUNT=0:
0151    ccr=ccr and #ef: /* keeps pe recognising irqs */
0152    init_vals:
0153 REPEAT
0154    ITERATE:
0155    COUNT=COUNT+1:
0156 forever:
0157 terminate program:
0158 /EOF
    #
```

THE INCLUDED FILE: LP_ITER.PL9

```
0001 IF PE_TAG=1 THEN
0002 BEGIN
0003 i_start_iteration=1:
0004 i_finish_iteration=MAX:
0005 j_start_iteration=1:
0006 j_finish_iteration=MAX:
0007 END:
0008 IF PE_TAG=4 THEN
0009 BEGIN
0010    i_start_iteration=2:
0011    i_finish_iteration=MAX:
0012    j_start_iteration=2:
0013    j_finish_iteration=MAX:
0014 END:
0015 IF PE_TAG=2 THEN
0016 BEGIN
0017    if local_id=4 .or local_id=2 then
0018       begin
0019          i_start_iteration=1:
0020          i_finish_iteration=MAX:
0021          j_start_iteration=1:
0022          j_finish_iteration=MAX+1:
0023       end;
0024    else
0025       begin
0026          i_start_iteration=1:
0027          i_finish_iteration=MAX+1:
0028          j_start_iteration=1:
0029          j_finish_iteration=MAX:
0030       end:
0031 END:
0032 IF PE_TAG=3 THEN
0033 BEGIN
0034    i_start_iteration=1:
0035    i_finish_iteration=MAX+1:
0036    j_start_iteration=1:
0037    j_finish_iteration=MAX+1:
0038 END:
0039 /EOF
   #
```

THE INCLUDED FILE: LP_ARRAY.PL9

```
0001 /* ELEMENT POS=PE_TAG     FROM SOURCE PROCESSOR */
0002 /* ELEMENT TYPE=LOCAL_ID FROM SOURCE PROCESSOR */
0003 IF PE_TAG
0004    CASE 1 THEN
0005       BEGIN
0006          if local_id=1 .or local_id=2 then i_start=1
0007                                          else i_start=0;
0008          if local_id=4 .or local_id=3 then i_finish=MAX
0009                                          else i_finish=MAX+1;
0010          if local_id=4 .or local_id=1 then j_start=1
0011                                          else j_start=0;
0012          if local_id=4 .or local_id=1 then i_finish=MAX+1
0013                                          else i_finish=MAX;
0014       END;
0015    CASE 2 THEN
0016       BEGIN
0017          if local_id
0018             CASE 4 then
0019                begin
0020                   i_start=0;
0021                   i_finish=MAX;
0022                   i_start=1;
0023                   i_finish=MAX+1;
0024                end;
0025             CASE 3 then
0026                begin
0027                i_start=1;
0028                   i_finish=MAX+1;
0029                   i_start=0;
0030                i_finish=MAX;
0031                end;
0032             ELSE
0033                begin
0034                   i_start=1;
0035                   i_finish=MAX+1;
0036                   i_start=1;
0037                   i_finish=MAX+1;
0038                end;
0039       END;
0040    CASE 3 THEN
0041       BEGIN
0042          i_start=1;
0043          i_finish=MAX+1;
0044          j_start=1;
0045          j_finish=MAX+1;
0046       END;
0047    ELSE
0048       BEGIN
0049          i_start=1;
0050          i_finish=MAX+1;
0051          j_start=1;
0052          j_finish=MAX+1;
0053       END;
0054 /EOF
    #
```

THE INCLUDED FILE: LP_VALS.PL9

```
0001 IF PE_TAG=1 THEN
0002 BEGIN
0003    if local_id=4 then /* .......... TLH PE .......... */
0004    begin
0005       i=0;
0006       repeat
0007          .res=get_address(0,integer(i+1),0);
0008          res=local_lhs;
0009          .res=get_address(i,MAX,0);
0010          res=local_top;
0011          i=i+1;
0012       until i=MAX;
0013    end;
0014    if local_id=3 then /* .......... BLH PE .......... */
0015    begin
0016       i=0;
0017       repeat
0018          .res=get_address(0,i,0);
0019          res=local_lhs;
0020          .res=get_address(i,0,0);
```

```
0021            res=local_bot;
0022            i=i+1;
0023         until i=MAX;
0024      end;
0025      if local_id=1 then /* .......... TRH PE .......... */
0026      begin
0027         i=1:
0028         repeat
0029            .res=get_address(i,MAX,0);
0030            res=local_top:
0031            .res=get_address(MAX,i,0);
0032            res=local_rhs;
0033            i=i+1:
0034         until i=MAX+1;
0035      end:
0036      if local_id=2 then /* .......... BRH PE .......... */
0037      begin
0038         i=1:
0039         repeat
0040            .res=get_address(i,0,0);
0041            res=local_bot:
0042            .res=get_address(MAX,i,0);
0043            res=local_rhs;
0044            i=i+1:
0045         until i=MAX+1;
0046      end;
0047 END;
0048 IF PE_TAG=4 THEN
0049 BEGIN
0050      i=1;
0051      repeat
0052         .res=get_address(i,1,0):
0053         res=local_bot:
0054         .res=get_address(MAX,i,0):
0055         res=local_rhs:
0056         .res=get_address(i,MAX,0):
0057         res=local_top:
0058         .res=get_address(1,i,0):
0059         res=local_lhs:
0060         i=i+1:
0061      until i=MAX+1:
0062 END:
0063 IF PE_TAG=2 THEN
0064 BEGIN
0065 if local_id
0066      CASE 1 then /* top */
0067         begin
0068            i=1:
0069            repeat
0070               .res=get_address(i,MAX,0):
0071               res=local_top:
0072               i=i+1:
0073            until i=MAX+1:
0074         end:
0075      CASE 2 then /* rhs */
0076         begin
0077            i=1
0078            repeat
0079               .res=get_address(MAX,i,0):
0080               res=local_lhs:
0081               i=i+1:
0082            until i=MAX+1:
0083         end:
0084      CASE 3 then /* bottom */
0085         begin
0086            i=1:
0087            repeat
0088               .res=get_address(i,0,0):
0089               res=local_bot:
0090               i=i+1:
0091            until i=MAX+1:
0092         end:
0093      ELSE /* lhs */
0094         begin
0095            i=1:
0096            repeat
0097               .res=get_address(0,i,0):
0098               res=local_lhs:
0099               i=i+1:
0100            until i=MAX+1:
0101         end;
0102 END:
0103 /EOF
    #
```

PROGRAM LISTING 6

THE MULTIPROCESSOR OPERATING SYSTEM

```
0001 /* PARALLEL LOADER */
0002
0003 /*       ROM VERSION        */
0004
0005 /* loader version 7 APR 87 */
0006
0007 CONSTANT RAM_BASE     =   $0000,
0008          NORTH        =   $6000,
0009          EAST         =   $7000,
0010          SOUTH        =   $8000,
0011          WEST         =   $9000;
0012
0013 AT $EF00: INTEGER JP,WI:
0014           BYTE    CKSUM,ECHO,TEST1,TEST2,
0015                   VALUE4,VALUE5,L,R,.RAN..SM:
0016           INTEGER INTERRUPT_VECTOR: REAL trh,brh,blh,tlh:
0017              BYTE global_halt_flag:
0018              INTEGER MAX: REAL .array: BYTE DATA_TYPE,local_id,pe_tag:
0019              BYTE    problem_type_flag:
0020              INTEGER SWI_VECTOR;
0021
0022 AT $2000: BYTE TESTER;
0023 /* WEST BLOCK PIGEON HOLES */
0024
0025 AT $9FF0: INTEGER SIZE_OF_DATA_FROM_WEST, DATA_POINTER_FROM_WEST:
0026              BYTE MESSAGE_FROM_WEST,
0027                   TO_FROM_WEST,
0028                   FROM_FROM_WEST,
0029                   INTERRUPT_ID_FROM_WEST:
0030           INTEGER SIZE_OF_DATA_TO_WEST,
0031                   DATA_POINTER_TO_WEST:
0032              BYTE MESSAGE_TO_WEST,
0033                   TO_TO_WEST,
0034                   FROM_TO_WEST,
0035                   INTERRUPT_ID_TO_WEST;
0036
0037 /* EAST BLOCK PIGEON HOLES */
0038
0039 AT $7FF0: INTEGER SIZE_OF_DATA_TO_EAST,
0040                   DATA_POINTER_TO_EAST:
0041              BYTE MESSAGE_TO_EAST,
0042                   TO_TO_EAST,
0043                   FROM_TO_EAST,
0044                   INTERRUPT_ID_TO_EAST:
0045           INTEGER SIZE_OF_DATA_FROM_EAST,
0046                   DATA_POINTER_FROM_EAST:
0047              BYTE MESSAGE_FROM_EAST,
0048                   TO_FROM_EAST,
0049                   FROM_FROM_EAST,
0050                   INTERRUPT_ID_FROM_EAST:
0051 AT $A000: BYTE RES_N, RES_E, RES_S, RES_W,
0052                IRQ_N, IRQ_E, IRQ_S, IRQ_W;
0053 AT $FFF4: BYTE ID;
0054
0055 INCLUDE 1.TRUFALSE.DEF;
0056 include 1.status6.def;
0057
0058 /* the position of get address to be loaded */
0059 ORIGIN $C000;
0060 /* ghost procedure to give correct jumps and call values */
0061 procedure get_address( integer i,j,k ): real .address;
0062    /*  dummy    */
0063    /*           */
0064    /*           */
0065 endproc  .address;
0066
0067 origin $f300;
0068
0069 PROCEDURE send_irq_east;
0070    IRQ_E=$FF;
0071 ENDPROC;
0072
0073 PROCEDURE send_irq_west;
0074    IRQ_W=$FF;
0075 ENDPROC;
0076
0077 PROCEDURE enable_interrupts;
0078    CCR = CCR AND $EF;
0079 ENDPROC;
0080
0081 PROCEDURE disable_interrupts;
0082    CCR = CCR OR $10;
0083 ENDPROC;
0084
0085 PROCEDURE synchronize;
0086    GEN $13;
0087 ENDPROC;
0088
0089 PROCEDURE initialise;
0090    /* processor has an identity ID         */
```

```
0091     /* remember I am am name not a number! */
0092     /* enable interrupts */
0093     global_halt_flag=FALSE;
0094     enable_interrupts;
0095 ENDPROC;
0096
0097 PROCEDURE reset_irq_from_east;
0098     RES_E=0;
0099 ENDPROC;
0100
0101 PROCEDURE reset_irq_from_west;
0102     RES_W=0;
0103 ENDPROC;
0104
0105 PROCEDURE reset_other_irqs;
0106     RES_N=0;
0107     RES_S=0;
0108     RES_E=0;
0109     RES_W=0;
0110 ENDPROC;
0111
0112 PROCEDURE pass_packet_e2w: byte .SM,E, .SM,N;
0113 .   /* copy commencing now: */
0114     INTERRUPT_ID_TO_WEST   =  INTERRUPT_ID_FROM_EAST;
0115     FROM_TO_WEST           =  FROM_FROM_EAST;
0116     TO_TO_WEST             =  TO_FROM_EAST;
0117     MESSAGE_TO_WEST        =  MESSAGE_FROM_EAST;
0118     DATA_POINTER_TO_WEST   =  DATA_POINTER_FROM_EAST;
0119     SIZE_OF_DATA_TO_WEST   =  SIZE_OF_DATA_FROM_EAST;
0120 ENDPROC;
0121
0122 PROCEDURE pass_packet_w2e:
0123     /* copy commencing now: */
0124     INTERRUPT_ID_TO_EAST   =  INTERRUPT_ID_FROM_WEST;
0125     FROM_TO_EAST           =  FROM_FROM_WEST;
0126     TO_TO_EAST             =  TO_FROM_EAST;
0127     MESSAGE_TO_EAST        =  MESSAGE_FROM_WEST;
0128     DATA_POINTER_TO_EAST   =  DATA_POINTER_FROM_WEST;
0129     SIZE_OF_DATA_TO_EAST   =  SIZE_OF_DATA_FROM_WEST;
0130 ENDPROC;
0131
0132 PROCEDURE reset_packet_east:
0133     INTERRUPT_ID_FROM_EAST = FALSE;
0134     reset_irq_from_east;
0135 ENDPROC;
0136
0137 PROCEDURE reset_packet_west:
0138     INTERRUPT_ID_FROM_WEST = FALSE;
0139     reset_irq_from_west;
0140 ENDPROC;
0141
0142 PROCEDURE pass_packet_east:
0143     /* pass packet w2e */
0144     pass_packet_w2e;
0145     /* reset original packet */
0146     reset_packet_west;
0147     /* send irq east with message */
0148     send_irq_east;
0149 ENDPROC;
0150
0151 PROCEDURE INCH: BYTE TEST2;
0152     TEST2=SM;
0153     .SM=.SM+1;
0154 ENDPROC BYTE TEST2;
0155
0156 PROCEDURE packet_east( BYTE MESSAGE_TO_SEND: INTEGER DATA_POINTER,
0157                                                      DATA_SIZE     );
0158     INTERRUPT_ID_TO_EAST = TRUE;
0159     FROM_TO_EAST         = ID;
0160     TO_TO_EAST           = MASTER;
0161     MESSAGE_TO_EAST      = MESSAGE_TO_SEND;
0162     DATA_POINTER_TO_EAST = DATA_POINTER;
0163     SIZE_OF_DATA_TO_EAST = DATA_SIZE;
0164 ENDPROC;
0165
0166 PROCEDURE packet_west( BYTE MESSAGE_TO_SEND: INTEGER DATA_POINTER,
0167                                                      DATA_SIZE     );
0168     INTERRUPT_ID_TO_WEST = TRUE;
0169     FROM_TO_WEST         = MASTER;
0170     TO_TO_WEST           = ID;
0171     MESSAGE_TO_WEST      = MESSAGE_TO_SEND;
0172     DATA_POINTER_TO_WEST = DATA_POINTER;
0173     SIZE_OF_DATA_TO_WEST = DATA_SIZE;
0174 ENDPROC;
0175
0176
0177 procedure dump: integer compact(0):byte element_pos,element_type;
0178     real     .loc_of_data,.packet_pos;
0179     integer i_start,i_finish,j_start,j_finish,i,j,init_loc,data_size;
0180     integer block_count,k,k_start,k_finish;
```

```
0181     element_pos=pe_tag;
0182     element_type=local_id;
0183     init_loc=4:8c0;
0184 /* ELEMENT_POS=PE_TAG     FROM SOURCE PROCESSOR */
0185 /* ELEMENT_TYPE=LOCAL_ID FROM SOURCE PROCESSOR */
0186 IF ELEMENT_POS
0187     CASE 1 THEN
0188        BEGIN
0189           if element_type=1 .or element_type=2 then i_start=1
0190                                                 else i_start=0;
0191           if element_type=4 .or element_type=3 then i_finish=MAX
0192                                                 else i_finish=MAX+1;
0193           if element_type=4 .or element_type=1 then j_start=1
0194                                                 else j_start=0;
0195           if element_type=4 .or element_type=1 then j_finish=MAX+1
0196                                                 else j_finish=MAX;
0197        END;
0198     CASE 2 THEN
0199        BEGIN
0200           if element_type
0201              CASE 4 then
0202                 begin
0203                    i_start=0;
0204                    i_finish=MAX;
0205                    j_start=1;
0206                    j_finish=MAX+1;
0207                 end;
0208              CASE 3 then
0209                 begin
0210                 i_start=1;
0211                    i_finish=MAX+1;
0212                    j_start=0;
0213                 j_finish=MAX;
0214                 end;
0215              ELSE
0216                 begin
0217                    i_start=1;
0218                    i_finish=MAX+1;
0219                    j_start=1;
0220                    j_finish=MAX+1;
0221                 end;
0222        END;
0223     CASE 3 THEN
0224        BEGIN
0225           i_start=1;
0226           i_finish=MAX+1;
0227           j_start=1;
0228           j_finish=MAX+1;
0229        END;
0230     ELSE
0231        BEGIN
0232           i_start=1;
0233           i_finish=MAX+1;
0234           j_start=1;
0235           j_finish=MAX+1;
0236        END;
0237 if problem_type_flag=1 then
0238 begin
0239     /* Navier_Stokes */
0240     k_start=0;
0241     k_finish=2;
0242 end;
0243 else
0244 begin
0245     /* default Laplace */
0246     k_start=0;
0247     k_finish=1;
0248 end;
0249 ccr=ccr or #10;
0250 k=k_start;
0251 repeat
0252     block_count=0;
0253     .packet_pos=init_loc;
0254     i=i_start;
0255     repeat
0256        j=i_start;
0257        repeat
0258           .loc_of_data=get_address(i,j,k);
0259           packet_pos=loc_of_data;
0260           .packet_pos=.packet_pos+4;
0261           block_count=block_count+1;
0262           if block_count=400 .or i*j=(i_finish-1)*(j_finish-1) then
0263           begin
0264              packet_east(DUMPING,#0800,compact);
0265              send_irq_east;
0266              gen #13;       /* wait for irq acknowledge                    */
0267              interrupt_id_from_east=false; /* reset cause */
0268              reset_irq_from_east;          /* reset cause */
0269              block_count=0;/* re_initialise count for next block if any */
0270              .packet_pos=init_loc; /* re_initialise this also            */
```

```
0271            end;
0272            j=j+1;
0273        until j=j_finish;
0274        i=i+1;
0275    until i=i_finish;
0276 k=k+1;
0277 until k=k_finish;
0278 ccr=ccr and def;
0279 endproc;
0280
0281
0282
0283
0284 PROCEDURE stop: integer reset;
0285    reset = #F000;
0286    jump reset;
0287 ENDPROC;
0288
0289 PROCEDURE pass_load_west: byte .SM_E, .SM_W;
0290    /* copy the program across first */
0291    .SM_E=#7000;
0292    .SM_W=#9000;
0293    REPEAT
0294        SM_W=SM_E;
0295        .SM_E=.SM_E+1;
0296        .SM_W=.SM_W+1;
0297    UNTIL .SM_E=#7030;
0298    /* copy the packet now */
0299    pass_packet_e2w;
0300    /* reset the original packet */
0301    reset_packet_east;
0302    /* send irq west with message */
0303    send_irq_west;
0304 ENDPROC;
0305
0306 PROCEDURE pass_packet_west;
0307    /* pass the run packet */
0308    pass_packet_e2w;
0309    /* reset the original packet */
0310    reset_packet_east;
0311    /* send irq west with message */
0312    send_irq_west;
0313 ENDPROC;
0314
0315 PROCEDURE pass_stop_program_west;
0316    /* ............................ TO BE WRITTEN */
0317 ENDPROC;
0318
0319 procedure halt;
0320    /* set the global interrupt flag, so that if irq'ed again    */
0321    /* if it is not the continue run then it will remain halted */
0322    global_halt_flag=TRUE;
0323    /* send message to MP to say its halted */
0324    packet_east(PROCESSOR_HALTED,NULL,NULL);
0325    send_irq_east;
0326 endproc;
0327
0328
0329 procedure resume;
0330    /* set the global halt flag so that execution can continue */
0331    /* after the interrupt has been serviced, (by this routine)*/
0332    global_halt_flag=FALSE;
0333    /* send the MP a message to say starting again */
0334    packet_east(CONTINUED_RUN,NULL,NULL);
0335    send_irq_east;
0336 endproc;
0337
0338
0339
0340 procedure pass_dump_data_east: real     .loc_of_data,.dest_of_data;
0341                                integer i,i_end;
0342    .loc_of_data =#9800;
0343    .dest_of_data=#7800;
0344    i=1;
0345    i_end=401;
0346    repeat
0347        dest_of_data =loc_of_data;
0348        .loc_of_data =.loc_of_data+4;
0349        .dest_of_data=.dest_of_data+4;
0350        i=i+1;
0351    until i=i_end;
0352    pass_packet_east;
0353 endproc;
0354
0355
0356
0357 PROCEDURE INHEX: BYTE VALUE1,FLAG;
0358    FLAG=0;
0359        VALUE1=INCH;
0360        /* VALID HEX? */
```

```
0361        IF VALUE1<'0 .OR VALUE1>'F THEN FLAG=1;
0362        IF VALUE1>'9 .AND VALUE1<'A THEN FLAG=1;
0363        IF VALUE1>'9 THEN VALUE1=VALUE1-#07;
0364        IF FLAG=1 THEN
0365        BEGIN
0366           PACKET_EAST(PROGRAM_LOAD_ERROR,NULL,NULL);
0367           send_irq_east;
0368        END;
0369     VALUE1=VALUE1-#30;
0370 ENDPROC BYTE VALUE1;
0371
0372 PROCEDURE BITE: BYTE VALUE2,VALUE3,CH_HEX;
0373     VALUE2=INHEX;
0374     VALUE2=SHIFT(VALUE2,4) AND #F0;
0375     VALUE3=INHEX;
0376     CH_HEX=VALUE2+VALUE3;
0377     CKSUM=CKSUM+CH_HEX;
0378 ENDPROC BYTE CH_HEX;
0379
0380 PROCEDURE BADDR: INTEGER WORD(0): BYTE WORDHI,WORDLO;
0381     WORDHI=BITE;
0382     WORDLO=BITE;
0383 ENDPROC INTEGER WORD;
0384
0385 PROCEDURE LOAD: BYTE FINISHED,COUNT,DUMMY;
0386    .SM=#7000;
0387     REPEAT
0388     UNTIL 'S'=INCH;
0389     IF SM='9' THEN
0390     BEGIN
0391        FINISHED=TRUE;
0392        PACKET_EAST(PROGRAM_LOADED_OK,NULL,NULL);
0393     END;
0394     ELSE
0395     BEGIN
0396        IF SM<>'1' THEN LOAD;
0397        .SM=.SM+1;
0398        CKSUM=0;
0399        COUNT=BITE;
0400        COUNT=COUNT-2;
0401        .RAM=BADDR;
0402        REPEAT
0403           COUNT=COUNT-1;
0404           IF COUNT<>0 THEN
0405           BEGIN
0406              RAM=BITE;
0407              .RAM=.RAM+1;
0408           END;
0409        UNTIL COUNT=0;
0410        DUMMY=BITE;
0411        CKSUM=CKSUM+1;
0412        IF CKSUM=0 THEN
0413        BEGIN
0414           PACKET_EAST(GOT_CODE_OK,NULL,NULL);
0415        END;
0416        ELSE
0417        BEGIN
0418           PACKET_EAST(PROGRAM_LOAD_ERROR,NULL,NULL);
0419        END;
0420     END;
0421     SEND_IRQ_EAST;
0422 ENDPROC;
0423
0424 PROCEDURE run: INTEGER .START_ADDRESS;
0425    .START_ADDRESS = #7FFA;
0426     /*
0427        yes i know this is defined as being the address
0428        of the data. but in the run procedure it contains
0429        the address from which the program should be run
0430     */
0431     packet_east( PROGRAM_RUNNING_OK,NULL,NULL );
0432     reset_irq_from_east;
0433     send_irq_east;
0434     JUMP START_ADDRESS;
0435 ENDPROC;
0436
0437
0438
0439 procedure irq2;
0440  if INTERRUPT_ID_FROM_EAST=TRUE then
0441   begin
0442    reset_irq_from_east;
0443    if ID=ID_FROM_EAST then
0444    begin
0445     if MESSAGE_FROM_EAST=LOAD_PROGRAM
0446     then load;
0447     else
0448      if MESSAGE_FROM_EAST=RUN_PROGRAM then run;
0449      else
0450      if MESSAGE_FROM_EAST=RUN_LAPLACE then
```

```
0451        begin
0452           problem_type_flag=0;
0453           run;
0454        end;
0455        else
0456        if MESSAGE_FROM_EAST=RUN_NS then
0457        begin
0458           problem_type_flag=1;
0459           run;
0460        end;
0461        else
0462         if MESSAGE_FROM_EAST=PROGRAM_STOPPED_OK
0463         then stop;
0464         else
0465          if MESSAGE_FROM_EAST=DUMP_DATA
0466          then dump;
0467          else
0468           if MESSAGE_FROM_EAST=HALT_PROCESSOR
0469           then halt;
0470           else
0471            if MESSAGE_FROM_EAST=CARRY_ON
0472            then resume;
0473            else
0474             begin
0475              packet_east(PROGRAM_LOAD_ERROR,NULL,NULL);
0476              send_irq_east;
0477             end;
0478    end;
0479   else
0480    begin
0481     if MESSAGE_FROM_EAST=LOAD_PROGRAM .OR
0482        MESSAGE_FROM_EAST=RUN_LAPLACE  .OR
0483        MESSAGE_FROM_EAST=RUN_NS
0484     then pass_load_west;
0485     else
0486      if MESSAGE_FROM_EAST=RUN_PROGRAM .OR
0487         MESSAGE_FROM_EAST=STOP_PROGRAM .OR
0488         MESSAGE_FROM_EAST=DUMP_DATA .OR
0489         MESSAGE_FROM_EAST=HALT_PROCESSOR .OR
0490         MESSAGE_FROM_EAST=CARRY_ON
0491      then pass_packet_west;
0492      else
0493       begin
0494        packet_east(TRANSMISSION_ERROR_E2W,NULL,NULL);
0495        send_irq_east;
0496       end;
0497    end;
0498    reset_packet_east;
0499    end;
0500    else
0501     if INTERRUPT_ID_FROM_WEST=TRUE
0502     then
0503     begin
0504      reset_irq_from_west;
0505      if MESSAGE_FROM_WEST=GOT_CODE_OK     .OR
0506         MESSAGE_FROM_WEST=PROGRAM_LOADED_OK  .OR
0507         MESSAGE_FROM_WEST=PROGRAM_STOPPED_OK     .OR
0508         MESSAGE_FROM_WEST=PROGRAM_RUNNING_OK .OR
0509         MESSAGE_FROM_WEST=PROCESSOR_HALTED   .OR
0510         MESSAGE_FROM_WEST=CONTINUED_RUN
0511      then pass_packet_east;
0512      else
0513       if MESSAGE_FROM_WEST=DUMPING
0514       then pass_dump_data_east;
0515       else
0516        begin
0517         packet_east(TRANSMISSION_ERROR_W2E,NULL,NULL);
0518         send_irq_east;
0519        end;
0520     end;
0521    else
0522     begin
0523      reset_other_irqs;
0524      packet_east(UNEXPECTED_INTERRUPT,NULL,NULL);
0525      send_irq_east;
0526     end;
0527  if global_halt_flag=TRUE then synchronize;
0528 endproc;
0529
0530 origin 4f100;
0531 procedure irq;
0532    call INTERRUPT_VECTOR;
0533 endproc;
0534
0535 procedure swi;
0536    call SWI_VECTOR;
0537 endproc;
```

```
0538
0539 ORIGIN $1000;
0540 STACK=$DFFF;
0541 PROCEDURE MAIN: byte .tester:
0542     initialise:
0543     .tester=$2000;
0544     INTERRUPT_VECTOR=.irq2;
0545     REPEAT
0546         tester=$55;
0547     FOREVER;
0548 ENDPROC;
0549 /EOF
     #
```

```
0001 /* CONSTANTS FOR STATUS REGISTER IN SHARED MEMORY */
0002
0003 CONSTANT MASTER                    = $00,
0004          LOAD_PROGRAM              = $01,
0005          PROGRAM_LOADED_OK         = $02,
0006          RUN_PROGRAM               = $03,
0007          PROGRAM_RUNNING_OK        = $04,
0008          STOP_PROGRAM              = $05,
0009          PROGRAM_STOPPED_OK        = $06,
0010          GOT_CODE_OK               = $07,
0011          PROGRAM_LOAD_ERROR        = $08,
0012          UNEXPECTED_INTERRUPT      = $09,
0013          TRANSMISSION_ERROR_E2W    = $0A,
0014          TRANSMISSION_ERROR_W2E    = $0B,
0015          DUMP_DATA                 = $0C,
0016          DUMPING                   = $0D,
0017          HALT_PROCESSOR            = $0E,
0018          PROCESSOR_HALTED          = $0F,
0019          CONTINUED_RUN             = $10,
0020          CARRY_ON                  = $11,
0021          RUN_LAPLACE               = $12,
0022          RUN_NS                    = $14,
0023          NULL                      = $0000;
0024 /EOF
     #
```

THE MULTIPROCESSOR OPERATING SYSTEM

THE FOLLOWING GIVES A BRIEF DESCRIPTION OF THE FUNCTION OF THE MAIN PROCEDURES OF THE MULTIPROCESSOR OPERATING SYSTEM:

PROCEDURE	INPUTS	ACTION
send_irq_east	-	sends an irq request to the east
send_irq_west	-	sends an irq request to the west
enable_interrupts	-	enables interrupts
disable_interrupts	-	disable interrupts
synchronize	-	one of two things: (a) if interrupts are enabled executing this instruction waits for an interrupt to arrive and then services that interrupt and then continues with execution of the next instruction (b) if interrupts are disabled executing this instruction waits for an interrupt to arrive and then

		continues with execution of the next instruction
initialise	-	enables interrupts
reset_irq_from_east	-	resets an irq from the east
reset_irq_from_west	-	resets an irq from the west
reset_other_irqs	-	is an interrupt occurs from an unexpected direction the unexpected irq is reset by this call
pass_packet_e2w	-	copies a packet at the top of memory in a direction east to west
pass_packet_w2e	-	copies a packet at the top of memory in a direction west to east
reset_packet_east	-	resets the irq flag and the irq from the direction east
reset_packet_west	-	resets the irq flag and the irq from the direction west
INCH	-	obtains a character from a sm location

pass_packet_east	-	sends packet received from west to the east, with associated interrupt handshakes
packet_east		message,dp,dsplaces a packet of information in east sm block pigeon holes. the message to pass is message, dp and ds are data pointer and data size pointers (null if not needed)
pass_load_west	-	copies motorola format packet of program at the base of east block sm to a corresponding section of west sm, and sends it on with associated interrupt handshakes
pass_run_program_west	-	pass the message to run from east sm to west sm with associated interrupt handshakes
INHEX	-	checks for a valid hex character after call to INCH, reporting any error to

		master, then converts it to numeric form
BITE	-	performs two calls to INHEX and outputs a hex byte
BADDR	-	performs two calls to BITE to obtain a valid hex address
LOAD	-	loads from sm a packet of program in motorola format, reporting correct input, correct termination of load and any errors to master
run	-	reads a start address from sm and jumps to it
irq	-	checks source of interrupt. either: loads a program; runs a program or passes on global communication as a result of getting an interrupt
main	-	initialises and synchronizes

PROGRAM LISTING 7

THE OVERSEER

```
0001 AT $CC14: integer lp;
0002
0003 include 1.os_a.pl9;
0004 include 1.status7.def;
0005 include 1.trufalse.def;
0006 include 1.iosubs.lib;
0007 include 1.hexio.lib;
0008 include 1.flex2.lib;
0009 include 1.numcon.lib;
0010 include 0.realcon.lib;
0011 include 0.realio.lib;
0012 include 1.os_b.pl9;
0013 /* ------------------------------------------------------------------ */
0014 procedure file_init: byte st(15);
0015    st(0)='1'; st(1)='.'; st(2)='R'; st(3)='.'; st(4)='P';
0016    st(5)='L'; st(6)='9';
0017    lp=.st;
0018    get_filename(.fcb);
0019    open_for_write(.fcb);
0020    SET_BINARY(.FCB);
0021    file_dump_option=true;
0022 endproc;
0023
0024 procedure write_real(real input): real fred(0): byte a,b,c,d;
0025    fred=input;
0026    write(.fcb,a);
0027    write(.fcb,b);
0028    write(.fcb,c);
0029    write(.fcb,d);
0030 endproc;
0031 /* ------------------------------------------------------------------ */
0032 procedure set_sm_pointer_to_data_start: integer wo(0):byte hi,lo;
0033    wo=0;
0034    lo=pe_no;
0035    wo=shift( ((wo-1) and $0003 ),12);
0036    wo=wo or $0800;
0037    pointer_value=wo;
0038    sm_pointer=wo;
0039 endproc;
0040
0041 /* ------------------------------------------------------------------ */
0042 include 1.os_c.pl9;
0043
0044 procedure p_menu3:byte x,y;
0045    x=35;
0046    y=15;
0047    clear_menu(0);
0048    move_cursor(x,y); print("MENU");
0049    move_cursor(x-1,y+1); print("******");
0050    x=x-4;
0051    move_cursor(x-7,y+3); print("Further possible options :");
0052    move_cursor(x-4,y+5); print("Positions        (type P)");
0053    move_cursor(x-4,y+6); print("Jump to restart (type J)");
0054    move_cursor(x-4,y+7); print("Refresh screen  (type F)");
0055    move_cursor(70,y+8);
0056 endproc;
0057
0058 procedure p_menu2: byte x,y;
0059    x=35;
0060    y=15;
0061    clear_menu(5);
0062    move_cursor(x,y); print("MENU");
0063    move_cursor(x-1,y+1); print("******");
0064    x=x-4;
0065    move_cursor(x-7,y+3); print("Further options :");
0066    move_cursor(x-4,y+5); print("Halt processors (type H)");
0067    move_cursor(x-4,y+6); print("Execute Laplace (type E)");
0068    move_cursor(x-4,y+7); print("Continue run    (type C)");
0069    move_cursor(x-4,y+8); print("Dump results    (type D)");
0070    /* extra commands here */
0071    move_cursor(70,y+8);
0072 endproc;
0073 include 1.os_d.pl9;
0074
0075
0076 /* ------------------------------------------------------------------ */
0077 include 1.loader_a.pl9;
0078 include 1.os_f.pl9;
0079 procedure retreive_data(byte irq_origin): integer .data_position,.sm,tt;
0080                                  real     .sm_real: byte ch, .ssm;
0081                                  integer i,j,i_pos,j_pos,temp,block_count;
0082    putchar(12);
0083    print("DATA FROM DIRECTION = ");
0084    put_hex_byte(irq_origin);
0085    crlf;
0086    crlf;
0087    pe_no=irq_origin;
0088    set_sm_pointer_to_data_start;
0089    .sm_real=$FF40;
0090    block_count=0;
```

```
0091     ccr = ccr or #10;
0092     i=0;
0093     repeat
0094        j=0;
0095        repeat
0096           /* o/p to disk in real life here */
0097           pnum((fix(sm_real))); print("  ");
0098 if file_dump_option=true then write_real(sm_real);
0099           .sm_real=.sm_real+4;
0100           block_count=block_count+1;
0101           if block_count=400 .or i*j=(max-1)*(max-1) then
0102           begin
0103              crlf;
0104              print("END OF BLOCK");
0105              crlf;
0106              pe_no=irq_origin;
0107              packet_to_mem(true,master,pe_no,carry_on,null,null);
0108              interrupt(direction_of_pe_no(pe_no));
0109              if i*j<>(max-1)*(max-1) then
0110              gen #13;
0111              /* reset h/w s/w irq */
0112              /* wherever it came from reset it ! */
0113              resirq(1); resirq(2); resirq(3); resirq(4);
0114              set_sm_pointer_top;
0115              .ssm=#ff57;
0116              ssm=false;
0117              block_count=0;
0118              set_sm_pointer_to_data_start;
0119              .sm_real=#ff40;
0120           end;
0121           else
0122           if .sm_real=#FF60 then
0123           begin
0124              .sm_real=#FF40;
0125              pointer_value=pointer_value+#0020;
0126              sm_pointer=pointer_value;
0127           end;
0128           j=j+1;
0129        until j=max;
0130        crlf;
0131        i=i+1;
0132     until i=max;
0133     crlf;
0134     CRLF;
0135     print("END OF PROCESSORS DATA");
0136     crlf;
0137 /*
0138 if pe_to_load=1 then close_file(.fcb);
0139 if pe_to_load=4 .and irq_origin=4 then close_file(.fcb);
0140 if pe_to_load=9 .and irq_origin=9 then close_file(.fcb);
0141 if pe_to_load=16 .and irq_origin=16 then close_file(.fcb);
0142 */
0143     ccr=ccr and #ef;
0144 endproc;
0145 /* ------------------------------------------------------------ */
0146 procedure irq: byte id,pe_no_save,.ssm: byte irq_origin:integer .wrong_val
ue:
0147     pe_no_save=pe_no;
0148     pe_no=1;
0149     .ssm=#ff57;
0150     /* set sm pointer */
0151     set_sm_pointer_top;
0152     if ssm=true then id=1
0153     else
0154     begin
0155        pe_no=2;
0156        set_sm_pointer_top;
0157        if ssm=true then id=2
0158        else
0159        begin
0160           pe_no=3;
0161           set_sm_pointer_top;
0162           if ssm=true then id=3
0163           else
0164           begin
0165              pe_no=4;
0166              set_sm_pointer_top;
0167              id=4;
0168           end;
0169        end;
0170     end;
0171     /* id now contains the direction from which the irq
0172        came, and the direction which it must be reset    */
0173     resirq(id);
0174     delay(#0010);
0175     .ssm=#ff57; ssm=false; /* a sw reset of interrupting pe */
0176     /* find the origin of the message */
0177     .ssm=#ff56;
0178     irq_origin=ssm;
0179     /* let the message be in ssm */
0180     .ssm=#ff54;
```

```
0181     status_box_update(irq_origin,ssm,0);
0182     if silent_running=FALSE then putchar(7);
0183     if ssm=dumping then
0184     begin
0185        retreive_data(irq_origin);
0186     end;
0187     if ssm=wrong_read then
0188     begin
0189        .wrong_value=#ff40;
0190        set_sm_pointer_to_data_start;
0191        status_box_update(irq_origin,ssm,wrong_value);
0192     end;
0193     delay(10010);
0194     res1=0;
0195     res2=0;
0196     res3=0;
0197     res4=0;
0198     reset_sm_pointer;
0199     pe_no=pe_no_save;
0200 endproc;
0201
0202
0203 include 1.os_e.pl9;
0204
0205
0206 include 1.loader_b.pl9;
0207
0208
0209 include 1.mul_ldr2.pl9;
0210
0211 /* ---------------------------------------------------------------- */
0212
0213 procedure dump: byte i,x,y,answer;
0214     /* clear screen and display a status type message */
0215     clear_menu(0);
0216     x=10;
0217     y=16;
0218     move_cursor(x,y); print("DUMP DATA");
0219     move_cursor(x-1,y+1); print("------------");
0220     /* check its the instruction required */
0221     move_cursor(x,y+4); print("ARE YOU SURE (Y/N) ? ");
0222     answer=getchar;
0223     if answer='y' .or answer='Y' then
0224     begin
0225     /* ok lets get on with it then */
0226        i=1;
0227        /* put packet information onto screen */
0228        paket_init;
0229        paket_fill(false,master,0,dump_data,null,null);
0230        /* loop until all have dumped */
0231 putchar(12);
0232        repeat
0233           disable_interrupts;
0234           paket_init;
0235           pe_no=pe(i);
0236           paket_fill(true,master,pe(i),dump_data,null,null);
0237           packet_to_mem(true,master,pe(i),dump_data,null,null);
0238           interrupt(direction_of_pe_no(pe(i)));
0239           synch;
0240           enable_interrupts;
0241           i=i+1;
0242        until i=pe_to_load+1;
0243     end;
0244 endproc;
0245
0246
0247 procedure send_halt: byte i,x,y,answer;
0248     /* clear screen and display a status type message */
0249     clear_menu(0);
0250     x=10;
0251     y=16;
0252     move_cursor(x,y); print("HALT ALL PROCESSING ELEMENTS");
0253     move_cursor(x-1,y+1); print("------------------------------");
0254     /* check its the instruction required */
0255     move_cursor(x,y+4); print("ARE YOU SURE (Y/N) ? ");
0256     answer=getchar;
0257     if answer='y' .or answer='Y' then
0258     begin
0259     /* ok lets get on with it then */
0260        i=1;
0261        /* put packet information onto screen */
0262        /* loop until all have dumped */
0263        repeat
0264           disable_interrupts;
0265           pe_no=pe(i);
0266           packet_to_mem(true,master,pe(i),halt_processor,null,null);
0267           interrupt(direction_of_pe_no(pe(i)));
0268           synch;
0269           enable_interrupts;
0270           i=i+1;
```

```
0271        until i=pe_to_load+1;
0272     end;
0273 endproc;
0274
0275 procedure exe_run: integer ttl,top_val,bot_val,rhs_val,lhs_val,n(0);
0276                          byte     n_hi,n_lo,buffer(20),.sm_byte,tt,i,x;
0277                          byte     y,answer;
0278                          integer .sm_integer; real .sm_real;
0279     /* clear screen and display a status type message */
0280     clear_menu(0);
0281     x=10;
0282     y=15;
0283     move_cursor(x,y);      print("MULTIPLE RUN");
0284     move_cursor(x-1,y+1); print("-------------");
0285     /* check its the instruction required */
0286     move_cursor(x,y+4); print("ARE YOU SURE (Y/N) ? ");
0287     answer=getchar;
0288     if answer='y' .or. answer='Y' then
0289     begin
0290        clear_menu(0);
0291        /* check alls ok */
0292        move_cursor(x,y+8); print("EVERYTHING O.K.     (Y/N)  ? ");
0293        answer=getchar;
0294        if answer='y' .or. answer='Y' then
0295        begin
0296           /* ok lets get on with it then */
0297           i=1;
0298           /* put packet information onto screen */
0299           paket_init;
0300           paket_fill(false,master,0,run_laplace,$B000,null);
0301           /* loop until all have dumped */
0302           repeat
0303              /* put initialise data into base of sm */
0304              pe_no=pe(i);
0305              disable_interrupts;
0306              paket_fill(true,master,pe(i),run_laplace,$B000,null);
0307              packet_to_mem(true,master,pe(i),run_laplace,$B000,null);
0308              interrupt(direction_of_pe_no(pe(i)));
0309              synch;
0310              enable_interrupts;
0311              i=i+1;
0312           until i=pe_to_load+1;
0313        end;
0314     end;
0315 endproc;
0316
0317
0318 procedure exe_laplace: integer ttl,top_val,bot_val,rhs_val,lhs_val,n(0);
0319                          byte     n_hi,n_lo,buffer(20),.sm_byte,tt,i,x;
0320                          byte     y,answer;
0321                          integer .sm_integer; real .sm_real;
0322     /* clear screen and display a status type message */
0323     clear_menu(0);
0324     x=10;
0325     y=15;
0326     move_cursor(x,y);      print("RUN LAPLACE");
0327     move_cursor(x-1,y+1); print("-------------");
0328     /* check its the instruction required */
0329     move_cursor(x,y+4); print("ARE YOU SURE (Y/N) ? ");
0330     answer=getchar;
0331     if answer='y' .or. answer='Y' then
0332     begin
0333        clear_menu(0);
0334        move_cursor(x,y);  print("HA HA RUNNING A LAPLACE PROBLEM MASTER");
0335        move_cursor(x-1,y+1); print("----------------------------------------
-");
0336        move_cursor(x,y+1); crlf;
0337        print("                 top_value"); crlf;
0338        print("            ----------------"); crlf;
0339        print("          |                  |"); crlf;
0340        print("lhs_value |        n x n     | rhs_value"); crlf;
0341        print("          |                  |"); crlf;
0342        print("            ----------------"); crlf;
0343        print("                 bot_value");
0344        /* get the initial values */
0345        x=42;
0346        move_cursor(x,y+2); print("VALUE FOR n ? ");
0347        n=getnum(input(.buffer,20));
0348        max=n_lo;
0349        move_cursor(x,y+3); print("VALUE FOR top_value ? ");
0350        top_val=getnum(input(.buffer,20)); print("  "); put_hex_address(top_
val);
0351        move_cursor(x,y+4); print("VALUE FOR rhs_value ? ");
0352        rhs_val=getnum(input(.buffer,20)); print("  "); put_hex_address(rhs_
val);
0353        move_cursor(x,y+5); print("VALUE FOR bot_value ? ");
0354        bot_val=getnum(input(.buffer,20)); print("  "); put_hex_address(bot_
val);
0355        move_cursor(x,y+6); print("VALUE FOR lhs_value ? ");
0356        lhs_val=getnum(input(.buffer,20)); print("  "); put_hex_address(lhs_
val);
0357        /* check alls ok */
0358        move_cursor(x,y+8); print("EVERYTHING O.K.     (Y/N)  ? ");
0359        answer=getchar;
0360        if answer='y' .or. answer='Y' then
```

```
0361        begin
0362           /* ok lets get on with it then */
0363           i=1;
0364           /* put packet information onto screen */
0365           paket_init;
0366           paket_fill(false,master,0,run_laplace,#B000,null);
0367           /* loop until all have dumped */
0368           repeat
0369              /* put initialise data into base of sm */
0370              pe_no=pe(i);
0371              set_sm_pointer_bot;
0372              .sm_integer=#FF40;
0373              sm_integer=n;
0374 .sm_byte=#FF42;
0375 sm_byte=pe_tags(pe(i)); .sm_byte=.sm_byte+1;
0376 sm_byte=local_ids(pe(i));
0377              .sm_real=#FF44;
0378              sm_real=float(top_val); .sm_real=.sm_real+4;
0379              sm_real=float(rhs_val); .sm_real=.sm_real+4;
0380              sm_real=float(bot_val); .sm_real=.sm_real+4;
0381              sm_real=float(lhs_val);
0382              disable_interrupts;
0383              paket_fill(true,master,pe(i),run_laplace,#B000,null);
0384              packet_to_mem(true,master,pe(i),run_laplace,#B000,null);
0385              interrupt(direction_of_pe_no(pe(i)));
0386              synch;
0387              enable_interrupts;
0388              i=i+1;
0389           until i=pe_to_load+1;
0390        end;
0391     end;
0392 endproc;
0393
0394 procedure j_resume: byte i,x,y,answer;
0395     /* clear screen and display a status type message */
0396     clear_menu(0);
0397     x=10;
0398     y=16;
0399     move_cursor(x,y);      print("CONTINUE EXECUTION");
0400     move_cursor(x-1,y+1); print("-------------------");
0401     /* check its the instruction required */
0402     move_cursor(x,y+4); print("ARE YOU SURE (Y/N) ? ");
0403     answer=getchar;
0404     if answer='y .or answer='Y then
0405     begin
0406     /* ok lets get on with it then */
0407     disable_interrupts;
0408     move_cursor(x,y+4); print("CONTINUE IN WHICH PE? ");
0409     pe_no=get_hex_byte;
0410     packet_to_mem(true,master,pe_no,carry_on,null,null);
0411     interrupt(direction_of_pe_no(pe_no));
0412     synch;
0413     enable_interrupts;
0414     end;
0415 endproc;
0416
0417
0418 procedure resume: byte i,x,y,answer;
0419     /* clear screen and display a status type message */
0420     clear_menu(0);
0421     x=10;
0422     y=16;
0423     move_cursor(x,y);      print("CONTINUE EXECUTION");
0424     move_cursor(x-1,y+1); print("-------------------");
0425     /* check its the instruction required */
0426     move_cursor(x,y+4); print("ARE YOU SURE (Y/N) ? ");
0427     answer=getchar;
0428     if answer='y .or answer='Y then
0429     begin
0430     /* ok lets get on with it then */
0431        i=1;
0432        /* put packet information onto screen */
0433        /* loop until all have dumped */
0434        repeat
0435           disable_interrupts;
0436           pe_no=pe(i);
0437           packet_to_mem(true,master,pe(i),carry_on,null,null);
0438           interrupt(direction_of_pe_no(pe(i)));
0439           synch;
0440           enable_interrupts;
0441           i=i+1;
0442        until i=pe_to_load+1;
0443     end;
0444 endproc;
0445
0446 /* ------------------------------------------------------------ */
0447
0448 procedure just_l:byte x,y,ch2;
0449     clear_menu(0);
0450     x=10; y=16;
```

```
0451     move_cursor(x,y); print("LOAD PE WITH A PROGRAM");
0452     move_cursor(x,y+1); print("-----------------------");
0453     paket_init;
0454     paket_fill(false,master,0,load_program,null,null);
0455     x=10; y=19;
0456     move_cursor(x,y); print("                          ");
0457     move_cursor(x,y); print("Which program");
0458     move_cursor(x,y+1); print("? ");
0459     /* call filename */
0460     file;
0461     move_cursor(x,y+3); print("                                  ");
0462     move_cursor(x,y+3); print("Destination PE no.");
0463     move_cursor(x,y+4); print("? ");
0464     /* get dest */
0465     pe_no=get_hex_byte;
0466     paket_fill(true,master,pe_no,load_program,null,null);
0467     move_cursor(x,y+3); print("                                        ");
0468     move_cursor(x,y+4); print("                                        ");
0469     move_cursor(x,y+3); print("READY TO SEND (Y/N)");
0470     ch2=getchar;
0471     if ch2='y' .or. ch2='Y' then
0472     begin
0473        clear_menu(40);
0474        move_cursor(40,17); print("TRANSMITTING ");
0475        hexdump;
0476        if error then report_error(.fcb);
0477        close_file(.fcb);
0478        end_record;
0479     end;
0480 endproc;
0481
0482 procedure just_r: byte x,y,ch2: integer address;
0483     clear_menu(0);
0484     x=10; y=16;
0485     move_cursor(x,y);  print("RUN PROGRAM IN A PE");
0486     move_cursor(x,y+1);print("--------------------");
0487     paket_init;
0488     paket_fill(false,master,0,run_program,null,null);
0489     x=10; y=18;
0490     move_cursor(x,y  ); print("                                  ");
0491     move_cursor(x,y  ); print("Run program in which PE");
0492     move_cursor(x,y+1); print("? ");
0493     pe_no=get_hex_byte;
0494     move_cursor(x,y+3); print("                                  ");
0495     move_cursor(x,y+4); print("                                  ");
0496     move_cursor(x,y+3); print("Start address");
0497     move_cursor(x,y+4); print("? ");
0498     address=get_hex_address;
0499     paket_fill(true,master,pe_no,run_program,address,null);
0500     move_cursor(x,y+3); print("                                  ");
0501     move_cursor(x,y+4); print("                                  ");
0502     move_cursor(x,y+3); print("READY TO SEND (Y/N) ?");
0503     ch2=getchar;
0504     if ch2='Y' .or. ch2='y' then
0505     begin
0506        disable_interrupts;
0507        packet_to_mem(true,master,pe_no,run_program,address,null);
0508        interrupt( direction_of_pe_no(pe_no) );
0509        synch;
0510        enable_interrupts;
0511        disable_interrupts;
0512     end;
0513     enable_interrupts;
0514 endproc;
0515
0516 procedure qt: byte i,.sm,s(5);
0517     s(1)=1; s(2)=5; s(3)=9; s(4)=40d;
0518     i=1;
0519     repeat
0520        pe_no=i;
0521        disable_interrupts;
0522        set_sm_pointer_bot;
0523        .sm=#FF40;
0524        sm='S;
0525        .sm=.sm+1;
0526        sm='9;
0527        packet_to_mem(true,master,i,load_program,null,null);
0528        interrupt(direction_of_pe_no(pe_no));
0529        synch;
0530        enable_interrupts;
0531        i=i+1;
0532     until i=17;
0533 endproc;
0534
0535
0536
0537 procedure silent;
0538     clear_menu(0);
0539     move_cursor(10,16); print("sound on = 1");
0540     move_cursor(10,17); print("sound off= 2");
```

```
0541     move_cursor(15,19);
0542     if getchar='2 then silent_running=TRUE
0543                   else silent_running=FALSE;
0544 endproc;
0545
0546 procedure results_to_file: byte ch;
0547     clear_menu(0);
0548     move_cursor(10,15);
0549     print("DO YOU WISH TO DUMP RESULTS TO FILE ?");
0550     move_cursor(17,17); ch=getchar;
0551     move_cursor(12,20);
0552     print("IF RESULTS FILE EXISTS DELETE IT ");
0553     if ch='y .or ch='Y then file_init;
0554     move_cursor(20,22);
0555     print("OK ?"); ch=getchar;
0556 endproc;
0557
0558 procedure les_options (byte order );
0559     if order='j .or order='J then warms;
0560     if order='L .or order='l then l_menu;
0561     if order='R .or order='r then r_menu;
0562     if order='s .or order='S then stop;
0563     if order='p .or order='P then compass;
0564     if order='h .or order='H then send_halt;
0565     if order='e .or order='E then exe_laplace;
0566     if order='c .or order='C then resume;
0567     if order='f .or order='F then refresh_display;
0568     if order='d .or order='D then dump;
0569     if order='t .or order='T then qt;
0570     if order='q .or order='Q then silent;
0571     if order='w .or order='W then just_l;
0572     if order='g .or order='G then just_r;
0573     if order='s .or order='S then exe_run;
0574     if order='v .or order='V then results_to_file;
0575     if order='z .or order='Z then close_file(.fcb);
0576     if order='b .or order='B then j_resume;
0577 endproc;
0578
0579 procedure main: byte order: integer restart;
0580     init;
0581     file_dump_option=false;
0582     restart=#0003;
0583        enable_interrupts;
0584 L1:
0585        p_menu;
0586        order=getchar;
0587        if order<>#0a .and order<>#0d then
0588        begin
0589           les_options(order);
0590           goto L1;
0591        end;
0592 L2:
0593        p_menu2;
0594        order=getchar;
0595        if order<>#0a .and order<>#0d then
0596        begin
0597           les_options(order);
0598           goto L2;
0599        end;
0600 L3:
0601        p_menu3;
0602        order=getchar;
0603        if order<>#0a .and order<>#0d then
0604        begin
0605           les_options(order);
0606           goto L3;
0607        end;
0608 L4:
0609        p_menu4;
0610        order=getchar;
0611        if order<>#0a .and order<>#0d then
0612        begin
0613           les_options(order);
0614           goto L4;
0615        end;
0616        goto L1;
0617 /EOF
     #
```

THE INCLUDED FILE: OS_A.PL9

```
0001 /*
0002
0003
0004                          ****************
0005                          * THE OVERSEER *
0006                          ****************
0007 */
0008    /* last mod 10 oct 86 */
0009    /* os1 */
0010
0011 constant end_of_file=8;
0012
0013 at $c840: byte fcb,error(319):
0014 at $cc09: byte ttyset_pause:
0015 at $7fff: byte status:
0016 at $ff20: byte res1,res2,res3,res4,
0017                irq1,irq2,irq3,irq4:
0018        integer sm_pointer:
0019 at $df84: integer irq_vector:
0020 at $a500: integer pointer_value: byte pe_no,max:integer .array,gef;
0021
0022 global byte checksum,loading:
0023        integer ccopy,length,address(0):
0024        byte address_high,address_low:
0025        byte ttyset_pause_save:
0026        byte stringi(20):
0027        byte buffer(255),erflag,keychar,
0028        .sm, ch, silent_running, pe_to_load, pe(17),
0029        local_ids(17),pe_tags(17):
0030        byte file_dump_option;
0031
0032 /EOF
    #n
```

THE INCLUDED FILE: STATUS7.DEF

```
0001 /* CONSTANTS FOR STATUS REGISTER IN SHARED MEMORY */
0002
0003 CONSTANT MASTER                    = $00,
0004          LOAD_PROGRAM              = $01,
0005          PROGRAM_LOADED_OK         = $02,
0006          RUN_PROGRAM               = $03,
0007          PROGRAM_RUNNING_OK        = $04,
0008          STOP_PROGRAM              = $05,
0009          PROGRAM_STOPPED_OK        = $06,
0010          GOT_CODE_OK               = $07,
0011          PROGRAM_LOAD_ERROR        = $08,
0012          UNEXPECTED_INTERRUPT      = $09,
0013          TRANSMISSION_ERROR_E2W    = $0A,
0014          TRANSMISSION_ERROR_W2E    = $0B,
0015          DUMP_DATA                 = $0C,
0016          DUMPING                   = $0D,
0017          HALT_PROCESSOR            = $0E,
0018          PROCESSOR_HALTED          = $0F,
0019          CONTINUED_RUN             = $10,
0020          CARRY_ON                  = $11,
0021          RUN_LAPLACE               = $12,
0022          wrong_read                = $14,
0023          NULL                      = $0000;
0024
0025 BYTE mt "MASTER             ",
0026         "LOADING A PROGRAM ",
0027         "PROGRAM LOADED OK ",
0028         "INSTRUCTED TO RUN ",
0029         "RUNNING A PROGRAM ",
0030         "INSTRUCTED TO STOP",
0031         "STOPPED OK        ",
0032         "GOT SECTION OK    ",
0033         "PROGRAM LOAD ERROR",
0034         "UNEXPECTED  I R Q ",
0035         "TRANS ERR E-W     ",
0036         "TRANS ERR W-E     ",
0037         "INSTRUCTED TO DUMP",
0038         "DUMPING DATA      ",
0039         "INSTRUCTED TO HALT",
0040         "PROCESSOR HALTED  ",
0041         "CONTINUED RUN     ",
0042         "INSTRUCTED TO CONT",
0043         "RUN_LAPLACE       ",
0044         "                  ";
0045 /EOF
```

THE INCLUDED FILE: OS_B.PL9

```
0001 at $cc14: integer the_file;
0002 procedure delay(integer i): integer c;
0003    c=0;
0004    repeat
0005       c=c+1;
0006    until c=i;
0007 endproc;
0008
0009
0010 procedure direction_of_pe_no( byte no );
0011 endproc ( ( (no-1) and $03 ) +1 );
0012
0013 procedure interrupt (byte direction);
0014    if direction=1 then irq1=$ff;
0015    if direction=2 then irq2=$ff;
0016    if direction=3 then irq3=$ff;
0017    if direction=4 then irq4=$ff;
0018 endproc;
0019
0020 procedure resirq(byte no);
0021    if no=1 then res1=0;
0022    if no=2 then res2=0;
0023    if no=3 then res3=0;
0024    if no=4 then res4=0;
0025 endproc;
0026
0027 procedure enable_interrupts;
0028    CCR = CCR AND $EF;
0029 endproc;
0030
0031 procedure disable_interrupts;
0032    CCR = CCR OR $10;
0033 endproc;
0034
0035 procedure set_sm_pointer_top;
0036    integer wo(0):byte hi,lo;
0037    wo=0; /* top two dc bits are zero */
0038    lo=pe_no;
0039    /* re pointer needs to be xxNN 1111 1110 0000 */
0040    wo = shift( ( (wo-1) and $0003 ) , 12 );
0041    wo=wo or $0FE0;
0042    sm_pointer=wo;
0043 endproc;
0044
0045 procedure set_sm_pointer_bot;
0046    integer wo(0): byte hi,lo;
0047    wo=0;
0048    lo=pe_no;
0049    wo= shift( ( (wo-1) and $0003 ) , 12 );
0050    wo=wo;
0051    pointer_value=wo;
0052    sm_pointer=wo;
0053 endproc;
0054 /EOF
```

THE INCLUDED FILE: OS_C.PL9

```
0001 procedure reset_sm_pointer;
0002    sm_pointer=pointer_value;
0003 endproc;
0004
0005 procedure move_cursor(byte x,y);
0006    putchar($0e);
0007    putchar(y+$20);
0008    putchar(x+$20);
0009 endproc;
0010
0011 procedure overseer_title;
0012    move_cursor(11,0);
0013    print("_____________________oo00 M   ER 00oo_____________________");
0014    CRLF; move_cursor(11,1);
0015    print("V                    V                    V                    V");
0016 endproc;
0017
0018 procedure status_box_init(byte no):byte x,y;
0019    x=((no-1) and $03 )*20+2;
0020    if no<17 then y=11;
0021    if no<13 then y=8;
0022    if no<9 then y=5;
0023    if no<5 then y=2;
0024    move_cursor(x,y);
0025    print("PROCESSOR No. ");
0026    move_cursor(x+14,y);
0027    put_hex_byte(no);
0028    move_cursor(x,y+1);
0029    print("status");
0030 endproc;
```

THE INCLUDED FILE: OS_D.PL9

```
0001
0002 procedure paket_init:byte x,y;
0003    x=40;
0004    y=14;
0005    clear_menu(x);
0006    move_cursor(x,y+2); print("INTERRUPT_ID ...");
0007    move_cursor(x,y+3); print("FROM ...........");
0008    move_cursor(x,y+4); print("TO .............");
0009    move_cursor(x,y+5); print("MESSAGE ........");
0010    move_cursor(x,y+6); print("DATA_POINTER ...");
0011    move_cursor(x,y+7); print("DATA_SIZE ......");
0012 endproc;
0013
0014 procedure paket_fill(byte irq_id,from,to,message:integer dp,ds): byte x,y;
0015    x=57; y=16;
0016    move_cursor(x,y); put_hex_byte(irq_id);
0017    move_cursor(x,y+1); put_hex_byte(from);
0018    move_cursor(x,y+2); put_hex_byte(to);
0019    move_cursor(x,y+3); put_hex_byte(message);
0020    move_cursor(x,y+4); put_hex_address(dp);
0021    move_cursor(x,y+5); put_hex_address(ds);
0022 endproc;
0023
0024 procedure realmaths: real .a_real;
0025    a_real=10.3;
0026    a_real=a_real*2.4;
0027 endproc;
0028
0029 /EOF
```

THE INCLUDED FILE: OS_F.PL9

```
0001 procedure packet_to_mem(byte irq_id,from,to,mess:integer dp,ds):integer tm
p1,tmp2,wor(0):byte hi,lo;
0002    tmp1=pointer_value;
0003    tmp2=.sm;
0004    status_box_update(to,mess,0);
0005    set_sm_pointer_top;
0006    .sm=$FF58;
0007    wor=ds;
0008    sm=hi; .sm=.sm+1; sm=lo; .sm=.sm+1;
0009    wor=dp;
0010    sm=hi; .sm=.sm+1; sm=lo; .sm=.sm+1;
0011    sm=mess; .sm=.sm+1;
0012    sm=to;   .sm=.sm+1;
0013    sm=from; .sm=.sm+1;
0014    sm=irq_id;
0015    .sm=tmp2;
0016    pointer_value=tmp1;
0017    sm_pointer=pointer_value;
0018 /*
0019 put_mem;
0020 */
0021 endproc;
0022 /EOF
```

THE INCLUDED FILE: OS_E.PL9

```
0001 procedure init: byte i;
0002    putchar(12);
0003    irq_vector=.irq;
0004    .array=$a510;
0005    ccopy=$8000;
0006    overseer_title;
0007    i=1;
0008    repeat
0009       status_box_init(i);
0010       i=i+1;
0011    until i=17;
0012    disable_interrupts;
0013    silent_running=FALSE;
0014    gef=$0000;
0015 endproc;
0016
0017
0018 procedure synch;
0019    gen $13;
0020 endproc;
0021 /EOF
```

THE INCLUDED FILE: LOADER_A.PL9

```
0001 procedure put_char(byte char);
0002    if char=lf then return;
0003 if silent_running=false then
0004 begin
0005    if char=cr then call $cd24 /* FLEX PCRLF */
0006                 else
0007                 begin
0008 /*
0009                    move_cursor(54,17);
0010                    putchar(char);
0011 */
0012                 end;
0013 end;
0014 if char=cr then
0015 begin
0016    set_sm_pointer_bot;
0017    .sm=$ff40;
0018 end;
0019 else
0020 begin
0021 ccopy=ccopy+1;
0022    repeat
0023       sm=char;
0024 /*
0025    move_cursor(40,19);put_hex_address(.sm);print("  ");put_hex_address(poi
nter_value);
0026    move_cursor(52,19);put_hex_byte(sm);print("  "); put_hex_address(ccopy)
;
0027 */
0028    until char=sm;
0029    if .sm and $001F = $001F then
0030    begin
0031       pointer_value=pointer_value+$0020;
0032       sm_pointer=pointer_value;
0033       .sm=$FF40;
0034    end;
0035       else .sm=.sm+1;
0036 end;
0037 endproc;
0038
0039 procedure put_crlf;
0040    put_char(cr);
0041    put_char(lf);
0042 endproc;
0043
0044 procedure put_hex(byte digit);
0045    digit=(digit and $f)+'0;
0046    if digit>'9 then digit=digit+7;
0047    put_char(digit);
0048 endproc;
0049
0050 procedure put_byte(byte item);
0051    put_hex(shift(item,-4));
0052    put_hex(item);
0053    checksum=checksum+item;
0054 endproc;
0055
0056 procedure put_address(integer item);
0057    put_byte(swap(item));
0058    put_byte(item);
0059 endproc;
0060 /EOF
```

THE INCLUDED FILE: LOADER_B.PL9

```
0001 procedure put_record: byte bb:
0002    byte count:
0003    integer position,marker;
0004    position=0;
0005 disable_interrupts;
0006    repeat
0007       packet_to_mem(true,master,pe_no,load_program,null,null);
0008       set_sm_pointer_bot;
0009       .SM=$FF40;
0010      if length-position>=16 then count =16
0011                             else count=length-position;
0012       marker=position+count;
0013       put_char('S);
0014       put_char('1);
0015       checksum=0;
0016       put_byte(count+3);
0017       put_address(address);
0018       repeat
0019          put_byte(buffer(position));
0020          position=position+1;
```

```
0021        until position=marker;
0022        put_byte(not(checksum));
0023        address=address+count;
0024        interrupt(direction_of_pe_no(pe_no));
0025        synch;
0026        enable_interrupts;
0027        disable_interrupts;
0028     until position=length;
0029 endproc;
0030
0031 procedure end_record;
0032     packet_to_mem(true,master,pe_no,load_program,null,null);
0033     disable_interrupts;
0034     .sm=$ff40;
0035     set_sm_pointer_bot;
0036     put_char('S);
0037     put_char('9);
0038     interrupt(direction_of_pe_no(pe_no));
0039     synch;
0040     enable_interrupts;
0041     disable_interrupts;
0042 endproc;
0043
0044 procedure get_record:byte char:integer i;
0045     repeat
0046        disable_interrupts;
0047        char=read(.fcb);
0048        if error then begin disable_interrupts; return; end;
0049     if char=$16 then
0050     begin
0051        read(.fcb); if error then begin disable_interrupts; return; end;
0052        read(.fcb); if error then begin disable_interrupts; return; end;
0053     end;
0054     until char=$02;
0055     address_high=read(.fcb);
0056     if error then return;
0057     address_low=read(.fcb);
0058     if error then return;
0059     length=integer(read(.fcb));
0060     if error then return;
0061     i=0;
0062     repeat
0063        buffer(i)=read(.fcb);
0064        if error then return;
0065        i=i+1;
0066     until i=length;
0067 endproc;
0068
0069 procedure hexdump;
0070 the_file=.stringi;
0071     get_filename(.fcb);
0072     if error then return;
0073     set_extension(.fcb,0);
0074     open_for_read(.fcb);
0075     if error then return;
0076     set_binary(.fcb);
0077     repeat
0078        get_record;
0079        if error=end_of_file then
0080        begin
0081           error=false;
0082           return;
0083        end;
0084        if error then return;
0085        put_record;
0086     forever;
0087 endproc;
0088
0089
0090 procedure file:byte letter:integer i;
0091     i=0;
0092     stringi(i)='1;  i=i+1;
0093     stringi(i)='.;  i=i+1;
0094     repeat
0095        letter=getchar;
0096        if letter<>$0d then
0097        begin
0098           stringi(i)=letter;  i=i+1;
0099        end;
0100     until letter=$0d;
0101     stringi(i)='.;  i=i+1;
0102     stringi(i)='c;  i=i+1;
0103     stringi(i)='m;  i=i+1;
0104     stringi(i)='d;  i=i+1;
0105     stringi(i)=$04;
0106     i=1; the_file=.stringi;
0107 endproc;
0108
0109 /EOF
```

THE INCLUDED FILE: MUL_LDR2.PL9

```
0001 procedure put_mem:integer tmp1,tmp2;
0002 move_cursor(0,0);
0003    tmp1=pointer_value;
0004    tmp2=.sm;
0005    set_sm_pointer_top;
0006    .sm=#FF5F;
0007    repeat
0008       put_hex_address(.sm);
0009       print("  ");
0010       put_hex_byte(sm);
0011       crlf;
0012       .sm=.sm-1;
0013    until .sm=#FF50;
0014    sm_pointer=tmp1
0015    pointer_value=tmp1;
0016    .sm=tmp2;
0017 putchar(getchar);
0018 endproc;
0019
0020 procedure puta(byte i:integer ii);
0021    put_hex_address(ii);
0022    print("  ");
0023    put_hex_byte(i);
0024    crlf;
0025 endproc;
0026
0027 procedure set_up_which_pes_to_load( byte flag, .pe );
0028    .pe=.pe+1; /* set to first element - 0th unused */
0029    if flag
0030       CASE 1 then begin pe=1; pe_tags(pe)=4; local_ids(pe)=1; end;
0031       CASE 4 then
0032              begin
0033                 pe=5; pe_tags(pe)=1; local_ids(pe)=4; .pe=.pe+1;
0034                 pe=6; pe_tags(pe)=1; local_ids(pe)=3; .pe=.pe+1;
0035                 pe=1; pe_tags(pe)=1; local_ids(pe)=1; .pe=.pe+1;
0036                 pe=2; pe_tags(pe)=1; local_ids(pe)=2;
0037              end;
0038       CASE 9 then
0039              begin
0040                 pe=9; pe_tags(pe)=1; local_ids(pe)=4; .pe=.pe+1;
0041                 pe=10; pe_tags(pe)=2; local_ids(pe)=4; .pe=.pe+1;
0042                 pe=11; pe_tags(pe)=1; local_ids(pe)=3; .pe=.pe+1;
0043                 pe=5; pe_tags(pe)=2; local_ids(pe)=1;  .pe=.pe+1;
0044                 pe=6; pe_tags(pe)=3; local_ids(pe)=0;  .pe=.pe+1;
0045                 pe=7; pe_tags(pe)=2; local_ids(pe)=3;  .pe=.pe+1;
0046                 pe=1; pe_tags(pe)=1; local_ids(pe)=1;  .pe=.pe+1;
0047                 pe=2; pe_tags(pe)=2; local_ids(pe)=2;  .pe=.pe+1;
0048                 pe=3; pe_tags(pe)=1; local_ids(pe)=2;
0049              end;
0050       ELSE
0051              begin
0052                 pe=13; pe_tags(pe)=1; local_ids(pe)=4; .pe=.pe+1;
0053                 pe=14; pe_tags(pe)=2; local_ids(pe)=4; .pe=.pe+1;
0054                 pe=15; pe_tags(pe)=2; local_ids(pe)=4; .pe=.pe+1;
0055                 pe=16; pe_tags(pe)=1; local_ids(pe)=3; .pe=.pe+1;
0056                 pe=9; pe_tags(pe)=2; local_ids(pe)=1;  .pe=.pe+1;
0057                 pe=10; pe_tags(pe)=3; local_ids(pe)=0; .pe=.pe+1;
0058                 pe=11; pe_tags(pe)=3; local_ids(pe)=0; .pe=.pe+1;
0059                 pe=12; pe_tags(pe)=2; local_ids(pe)=3; .pe=.pe+1;
0060                 pe=5 ; pe_tags(pe)=2; local_ids(pe)=1; .pe=.pe+1;
0061                 pe=6 ; pe_tags(pe)=3; local_ids(pe)=0; .pe=.pe+1;
0062                 pe=7; pe_tags(pe)=3; local_ids(pe)=0;  .pe=.pe+1;
0063                 pe=8; pe_tags(pe)=2; local_ids(pe)=3;  .pe=.pe+1;
0064                 pe=1; pe_tags(pe)=1; local_ids(pe)=1;  .pe=.pe+1;
0065                 pe=2; pe_tags(pe)=2; local_ids(pe)=2;  .pe=.pe+1;
0066                 pe=3; pe_tags(pe)=2; local_ids(pe)=2;  .pe=.pe+1;
0067                 pe=4; pe_tags(pe)=1; local_ids(pe)=2;
0068              end;
0069 endproc;
0070
0071
0072 procedure l_menu:byte x,y,ch2,ii;
0073    pe(1)=5; pe(2)=6; pe(3)=1; pe(4)=2;
0074    clear_menu(0);
0075    x=10; y=16;
0076    move_cursor(x,y); print("LOAD PE WITH A PROGRAM");
0077    move_cursor(x,y+1); print("----------------------");
0078    paket_init;
0079    paket_fill(false,master,0,load_program,null,null);
0080    x=10; y=19;
0081    move_cursor(x,y); print("                        ");
0082    move_cursor(x,y); print("Which program");
0083    move_cursor(x,y+1); print("? ");
0084    /* call filename */
0085    file;
```

```
0086    move_cursor(x,y+3); print("                                    ");
0087 move_cursor(x,y+3);print("How many PEs ");
0088   move_cursor(x,y+4); print("(2 digit HEX) ? ");
0089    /* get dest */
0090    pe_to_load=get_hex_byte;
0091    set_up_which_pes_to_load(pe_to_load, .pe);
0092    /* check its ok */
0093    move_cursor(x,y+3); print("                                    ");
0094    move_cursor(x,y+4); print("                                    ");
0095    move_cursor(x,y+3); print("ok to send (y/n) ");
0096    ch2=getchar;
0097    if ch2='y .or ch2='Y then
0098    begin
0099 ii=1;
0100 repeat
0101 pe_no=pe(ii);
0102    paket_fill(true,master,pe(ii),load_program,null,null);
0103    packet_to_mem(true,master,pe(ii),load_program,null,null);
0104    move_cursor(x,y+3); print("                                    ");
0105    move_cursor(x,y+4); print("                                    ");
0106    move_cursor(x,y+3); print("READY TO SEND (Y/N)");
0107    ch2='y;
0108    if ch2='y .or ch2='Y then
0109    begin
0110       clear_menu(40);
0111       move_cursor(40,17); print("TRANSMITTING ");
0112       hexdump;
0113       if error then report_error(.fcb);
0114       close_file(.fcb);
0115       end_record;
0116    end;
0117 ii=ii+1;
0118 until ii=pe_to_load+1;
0119    end; /* end of ok to send */
0120 endproc;
0121
0122
0123 procedure refresh_display: byte i;
0124    disable_interrupts;
0125    putchar(12);
0126    overseer_title;
0127    i=1;
0128    repeat
0129       status_box_init(i);
0130       i=i+1;
0131    until i=17;
0132    enable_interrupts;
0133 endproc;
0134
0135 procedure stop;
0136 /*     ................................ TO BE WRITTEN */
0137 endproc;
0138
0139 procedure r_menu: byte x,y,ch2: integer address;
0140    clear_menu(0);
0141    x=10; y=16;
0142    move_cursor(x,y);  print("RUN PROGRAM IN A PE");
0143    move_cursor(x,y+1);print("-------------------");
0144    paket_init;
0145    paket_fill(false,master,0,run_program,null,null);
0146    x=10; y=18;
0147    move_cursor(x,y  ); print("                          ");
0148    move_cursor(x,y  ); print("Run program in which PE");
0149    move_cursor(x,y+1); print("? ");
0150    pe_no=get_hex_byte;
0151    move_cursor(x,y+3); print("                          ");
0152    move_cursor(x,y+4); print("                          ");
0153    move_cursor(x,y+3); print("Start address");
0154    move_cursor(x,y+4); print("? ");
0155    address=get_hex_address;
0156    paket_fill(true,master,pe_no,run_program,address,null);
0157    move_cursor(x,y+3); print("                          ");
0158    move_cursor(x,y+4); print("                          ");
0159    move_cursor(x,y+3); print("READY TO SEND (Y/N) ?");
0160    ch2=getchar;
0161    if ch2='Y .or ch2='y then
0162    begin
0163       disable_interrupts;
0164       packet_to_mem(true,master,pe_no,run_program,address,null);
0165       interrupt( direction_of_pe_no(pe_no) );
0166       synch;
0167       enable_interrupts;
0168       disable_interrupts;
0169    end;
0170    enable_interrupts;
0171 endproc;
0172
```

```
procedure compass: byte y,x,ch;
   putchar(12);
   print("COMPASS DIRECTIONS OF PEs")
   crlf;
   print("-------------------------");
   y=5;
   x=40;
   repeat
      move_cursor(x,y); print(":");
      y=y+1;
   until y=12;
   y=8;
   x=20;
   repeat
      move_cursor(x,y); print("-");
      x=x+1;
   until x=50;
   move_cursor(40,3); print("E");
   move_cursor(40,13); print("W");
   move_cursor(18,8); print("N");
   move_cursor(52,8); print("S");
endproc;
/EOF
#
```

THE OVERSEER

THE FOLLOWING GIVES A BRIEF DESCRIPTION OF THE FUNCTION OF THE MAIN PROCEDURES OF THE MULTIPROCESSOR OPERATING SYSTEM:

PROCEDURE	INPUTS	ACTION
direction_of_pe_no	pe_number	outputs a value which corresponds to the direction in which a message needs to be passed (either 1,2,3 or 4) EXIT no alt
interrupt	direction	sends interrupt in direction of 'direction' variable (1,2,3 or 4) EXIT no alt
resirq	direction	sends a reset interrupt signal in the direction of the 'direction' variable EXIT no alt
set_sm_pointer_top	pe_no	sm_pointer is set to appropriate chunk of sm to enable communication with pe

		pe_no
		EXIT sm_pointer alt
set_sm_pointer_bot	pe_no	sm_pointer and its RAM counterpart pointer_value are set to the appropriate value to enable communication with pe pe_no
		EXIT sm_pointer alt pointer_value alt
reset_sm_pointer	-	after a set_sm_pointer_top the pointer can be reset by exuating it to its RAM counterpart
		EXIT sm_pointer alt (to original value before call to set sm pointer top)
move_cursor	x,y	moves cursor to x,y on screen
		EXIT no alt
overseer_title	-	prints program title
		EXIT no alt

status_box_init	pe_no	sets up area on screen in which the status of pe pe_no is displayed EXIT no alt
status_box_update	pe_no,mess	updates information in status box pe_no with the message corresponding to mess EXIT no alt
clear_memu	-	clears menu from screen EXIT no alt
p_menu		prints main menu on screen EXIT no alt
packet_init	-	displays on screen the packet headings of a message EXIT no alt
paket_fill	irq_id,from message,dp,ds	puts the information received into a packet displayed on the screen

		EXIT no alt
irq	-	deals with interrupts arising from global communication to/from other PE and takes the appropriate action
		EXIT no alt
r_menu	-	displays the run menu. enables a program to be run in an arbitrary pe starting at an arbitary location
		EXIT no alt
hexdump	pe_no	reads in a file and sends down appropriate direction with full handshaking
		EXIT no alt
file	-	inputs a filename with binary defaults
		EXIT no alt
put_mem	-	displays the top 16 bytes of a sm block previously defined

		EXIT no alt
put_a	add,byte	displays a byte held in address add
		EXIT no alt
packet_to_mem	irq_id,from to,mess,dp,ds	places packet into memory
		EXIT no alt
main	-	scans keyboard for inputs, while being enabled for any interrupts