Hardware and Software aspects of Parallel Computing

BY

Lesley Bissland

A thesis submitted to the University of Glasgow for the degree of Doctor of Philosophy in the Faculty of Science

Department of Chemistry

January 1996

Abstract

Parallel computing has developed in an attempt to satisfy the constant demand for greater computational power than is available from the fastest processors of the time. This has evolved from parallelism within a single Central Processing Unit to thousands of CPUs working together. The development of both novel hardware and software for parallel multiprocessor systems is presented in this thesis.

A general introduction to parallel computing is given in Chapter 1. This covers the hardware design concepts used in the field such as vector processors, array processors and multiprocessors. The basic principles of software engineering for parallel machines (i.e. decomposition, mapping and tuning) are also discussed.

Part 1 (Chapters 2,3 and 4) is concerned with the development of hardware for multiprocessor systems. Some of the concepts used in digital hardware design are introduced in Chapter 2. These include the fundamentals of digital electronics such as logic gates and flip-flops as well as the more complicated topics of rom and programmable logic.

It is often desirable to change the network topology of a multiprocessor machine to suit a particular application. The third chapter describes a circuit switching scheme that allows the user to alter the network topology prior to computation. To achieve this, crossbar switches are connected to the nodes, and the host processor (a PC) programs the crossbar switches to make the desired connections between the nodes. The hardware and software required for this system is described in detail.

Whilst this design allows the topology of a multiprocessor system to be altered prior to computation, the topology is still fixed during program run-time. Chapter 4 presents a system that allows the topology to be altered during run-time. The nodes send connection requests to a control processor which programs a crossbar switch connected to the nodes. This system allows every node in a parallel computer to communicate directly with every other node. The hardware interface between the nodes and the control processor is discussed in detail, and the software on the control processor is also described.

Part 2 (Chapters 5 and 6) of this thesis is concerned with the parallelisation of a large molecular mechanics program. Chapter 5 describes the fundamentals of molecular mechanics such as the steric energy equation and its components, force field parameterisation and energy minimisation.

The implementation of a novel programming (COMFORT) and hardware (the BB08) environment into a parallel molecular mechanics (MM) program is presented in Chapter 6. The structure of the sequential version of the MM program is detailed, before discussing the implementation of the parallel version using COMFORT and the BB08.

Table of Contents

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Appendix D

List of Tables

Acknowledgements

I would like to gratefully acknowledge the guidance, support and encouragement given to me by my supervisor Dr. David White. His advice and direction was invaluable during my research.

Thanks must also be given to various staff and colleagues of the Chemistry department both past and present. In particular to Noel Ruddock for his guidance with software development and Dr Chris Gilmore for the use of his computers and printers for writing this thesis (and many other reports). My sincere appreciation goes to Stuart Mackay for taking the time to proof read this thesis and for all his suggestions.

A big thank you to Lesley Ann for her friendship, support and lunches at the QM! I would also like to thank Arlene for her friendship and putting up with sharing an office with me.

Thanks is also due to my Mum for taking the photos included in this thesis.

The financial support received from the CEU (Commission of the European Union) to attend a conference in Harrogate is also gratefully acknowledged.

Finally I would like to thank the E.P.S.R.C for funding this research and the attendance at a conference in the U.S.A.

This thesis is dedicated to my parents

for all their support throughout my academic career.

Chapter 1

Introduction to Parallel Computing

Throughout the evolution of computing, parallelism has become more and more significant. Due to the demand for more powerful machines, designers have had to conceive methods of achieving greater speed with the available technology of the day. This has often been achieved by parallelism within a sequential single processor machine or by using several sequential processors working together.

This chapter explains the need for parallel computing and also describes some of the problems associated with it. The main types of parallel architecture and the taxonomies developed to describe parallel systems are detailed. An overview is presented of the issues involved in parallel software engineering and finally the key developments in parallel computing are described.

1.1 What is Parallel Computing?

The basic concept of parallel computing is that a computation is distributed over several processing units, enabling parts of the program to be executed simultaneously. This will potentially speed up the computation compared to executing it on a sequential machine. This approach is analogous to a team of people working on a common task. You would hope to complete the task faster with a team of people than with a single person.

It is not the case however that every program can be speeded up by executing it in parallel. As with people the processors in a parallel computer have to communicate with each other to work effectively and this is one of the main overheads in parallel computing. Also the algorithm must be suited to parallel computing. An algorithm that requires a high ratio of communication to computation would not necessarily be speeded up by parallel computing.

1.2 Why parallel computing?

With the advent of high performance workstations it is often asked why there is a need for parallel computing. The main reason is that even the fastest computers available today are still not powerful enough for the so called Grand Challenges of science. These include applications in weather forecasting, computational fluid dynamics used in the automotive industry and drug design used in the pharmaceutical industry.

Even today's fastest computers are approaching the limits imposed by physics. The propagation delays of signals are restricted by the speed of light. As designers try to shrink architectures to reduce the distance signals require to travel, device physicists are concerned about the impact of atom spacing on their ability to make smaller and, hence, faster transistors. It is therefore probable that designers will have no choice but to rely on parallelism to achieve higher performance.

Another reason for parallel systems is that they can provide a good cost/performance ratio. Many large problems are solved too slowly on a sequential machine to be cost-effective, the reason being that for high-performance single processors the price grows rapidly with speed. It can therefore be less expensive and faster to use several 'off-the-shelf' processors to achieve high performance.

Other advantages of parallel systems include scalability and availability. A well-designed parallel system will allow for the addition of more processors as they become available or as the users computing requirements grow. Also as there is a high availability of components, if one fails the system should be able to continue operation using the remaining components.

1.2.1 Problems in parallel computing

Sequential computers are based on a single underlying model of computation known as the Von Neumann model. This single model has given manufacturers and users a common paradigm on which to construct their software and hardware. This has led to common standards within the sequential market and as a result has given rise to software that is portable between platforms. In parallel computing there is no single model of computation which can lead to problems when porting parallel software from one hardware platform to another.

The primary difference between parallel and sequential computing is that in a parallel computer a program is divided up into processes which maybe on separate processors. These processes will require to communicate with each other in order to produce an overall solution to the problem. At present there is no common standard used to pass messages between processors. Several schemes have been suggested and some of these will be discussed in the course of this thesis.

Other problems that occur in parallel computing and not in sequential computing are deadlock and livelock. Deadlock is where two or more parallel processes can no longer execute any further due to a communication interdependency. Livelock is the state where a process remains active on a processor but does not communicate and acts like an infinite loop. Software engineers need to prevent these situations.

1.3 Sequential models of computation

Sequential models of computation are often used as building blocks for parallel machines. Three of the most common sequential models are discussed below.

1.3.1 Von Neumann model

The von Neumann model of computation is illustrated in Figure 1.1.

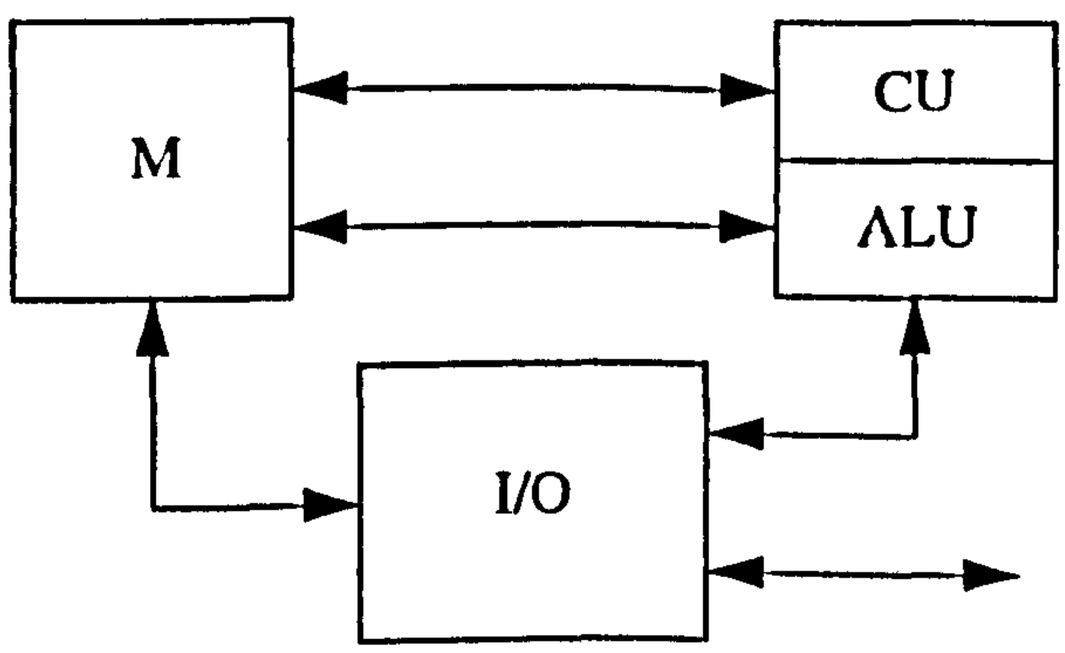


FIGURE 1.1.Von Neumann computer model

A classical von Neumann computer consists of a program control unit (CU), an arithmetic logic unit (ALU), an input/output (I/O) unit, and memory (M). The CU and ALU collectively make up the processing element.

The von Neumann model is based on the following principles:-

- A single processing element separated from memory by a communication bus
- Linear organisation of fixed-size memory cells
- Low-level machine language with instructions performing simple operations on elementary operands
- Sequential centralised control of computations

These principles are simple and well understood and considerable progress has been made with them over the years.

1.3.2 Harvard Architecture

The Harvard Architecture is a variation on the von Neumann model and uses two separate memories for instructions and data instead of the one memory for both as in the von Neumann model. This allows both instructions and data to be accessed simultaneously improving the speed of the machine.

1.3.3 Data-Flow Computations

In a data-flow machine computations take place when operands become available eliminating the need for a program counter. In a von Neumann computer the program counter stores the address of the next instruction in order to process instructions in a sequential manner. It is the data dependencies that constrain the order of computations in a data-flow machine.

The result produced by an instruction is used as a token which passes to the operands of the next instruction. Figure 1.2 overleaf shows a data flow graph for the calculation $z=y(x+1)$. Here the circles represent nodes which are connected by arcs and the dots on the arcs represent tokens. For example purposes x and y are 4 and 5 respectively.

Each node is only permitted to compute when tokens are present on each input arc and there are not tokens on the output arc. In Figure 1.2(a) the "plus" node can compute but the "multiplication" node cannot. In Figure 1.2(b) as the "plus" node has produced a token which enables the "multiplication" node to compute. Finally in Figure 1.2(c) the result $z=25$ is produced

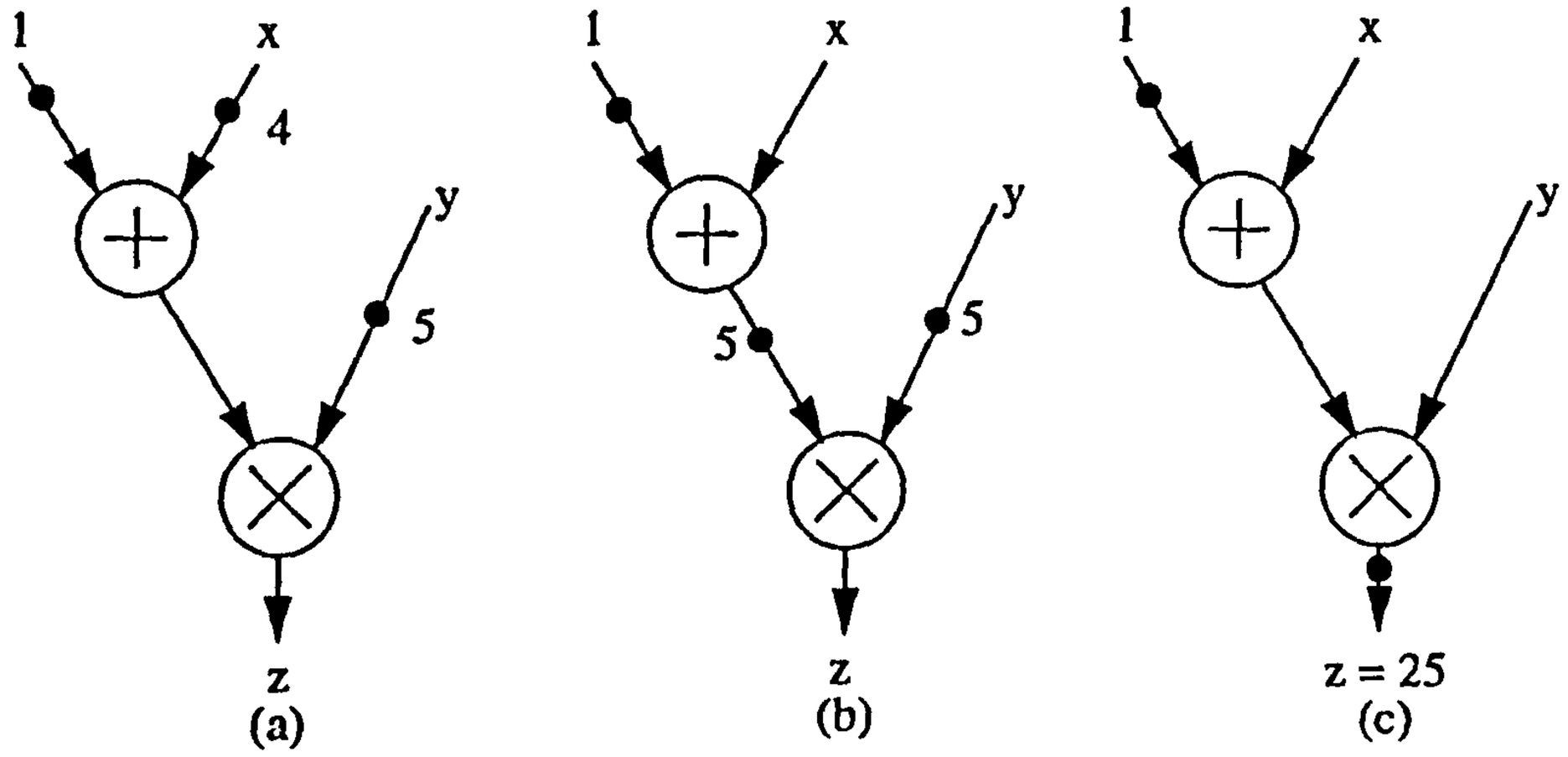


FIGURE 1.2.Snapshots of a data flow diagram for $z = y(x+1)$

1.4 Parallel Concepts

There are various hardware schemes that exploit parallelism in computing[1,2]. Some of these are detailed below.

1.4.1 Pipelining.

Pipelining divides a task T into subtasks T_1, T_2, .., T_k and assigns the subtasks to a chain of processing elements (PEs). Each PE executes a particular subtask and passes its result onto the next PE similar to an assembly line in a factory. Pipelining can be applied at instruction or arithmetic level.

An instruction cycle typically consists of 3 stages. i.e

1) Fetch instruction.

2) Decode instruction.

3) Execute instruction.

In a pipelined processor these functions are carried out in parallel. As one instruction is being decoded the next one will be fetched which means the ALU (arithmetic logic unit) always has an instruction waiting for it. This approach works best with programs

which contain long sections of sequential code, as obviously if an instruction has been prefetched and the previous instruction was a 'JUMP' instruction then the new instruction will have to be discarded. The speed-up obtained from pipelining also depends on the length of the pipe as the longer the pipeline the longer it takes to 'flush' out the pipeline.

In arithmetic pipelining the ALU is arranged as a series of stages, and operations inside the ALU are pipelined. For example when multiplying two floating point numbers A and B, at instant one, stage one calculates the difference between the exponents of A and B. At instant two, stage two aligns the mantissas of A and B at the same time as stage one calculates the difference between the exponents of the next two numbers (C & D).

Most of todays sequential processors use some form of instruction level and arithmetic pipelining. In this way parallelism is present within a single sequential processor.

1.4.2 Vector Processors

Vector processors are specifically designed for computations involving vectors. For example the subtraction of two vectors of n elements can be performed simultaneously on all n elements. This can be achieved by replicating the number of ALUs to the size of the vectors. This requires a considerable amount of hardware and is not particularly flexible. A better approach is to use pipelining.

Vectors are one-dimensional arrays of data and the same sequence of operations is required for each vector element. One or more pipelined ALUs may be used and the vector elements are pushed through the pipeline.(See Figure 1.3)

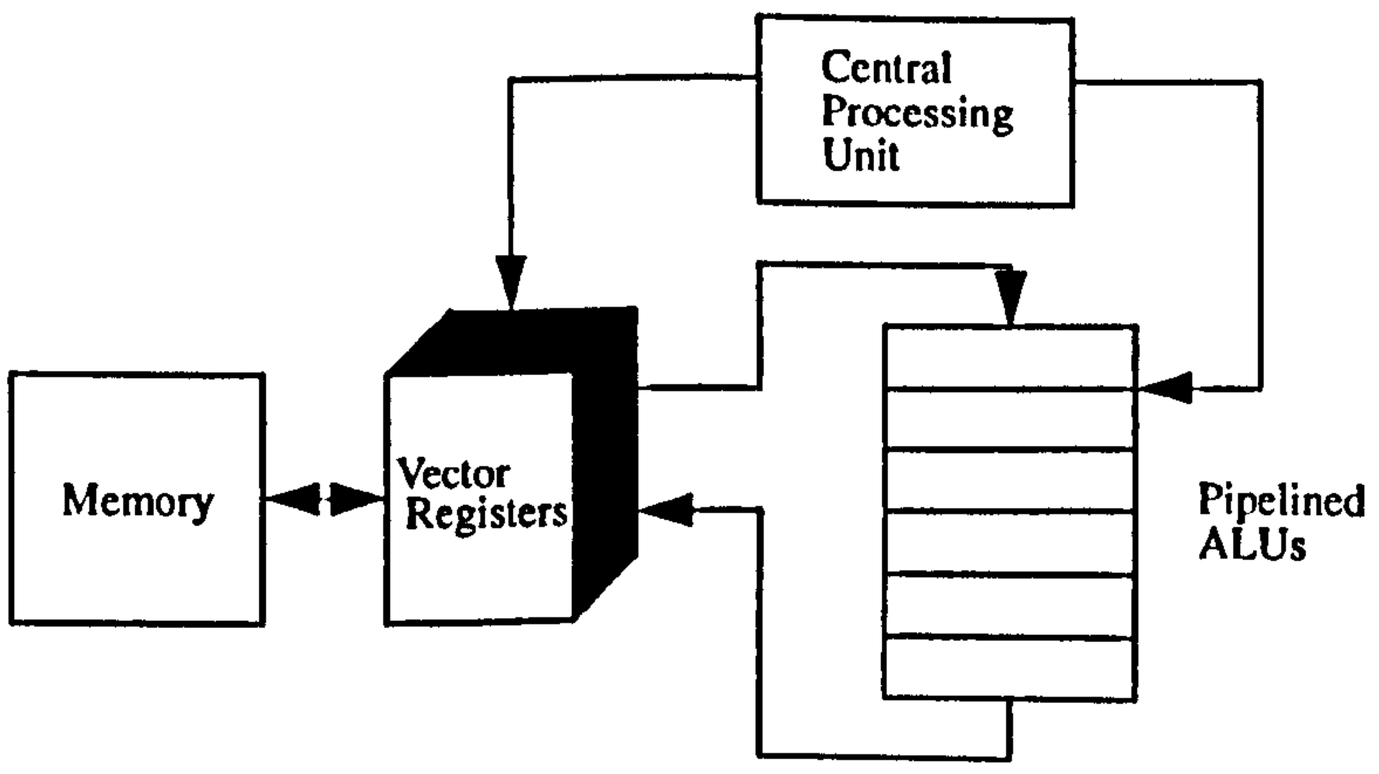


FIGURE 1.3.Vector Processor

1.4.3 Array Processors

An array processor is a synchronous parallel computer which consists of multiple processors under the supervision of a single control unit (See Figure 1.4). The processors each perform the same instruction at the same time but on different data. The control unit synchronises all the processors and collects the results from the processors. This approach is useful for programs with large arrays of data which require the same operation to be executed on each of the elements in the array.

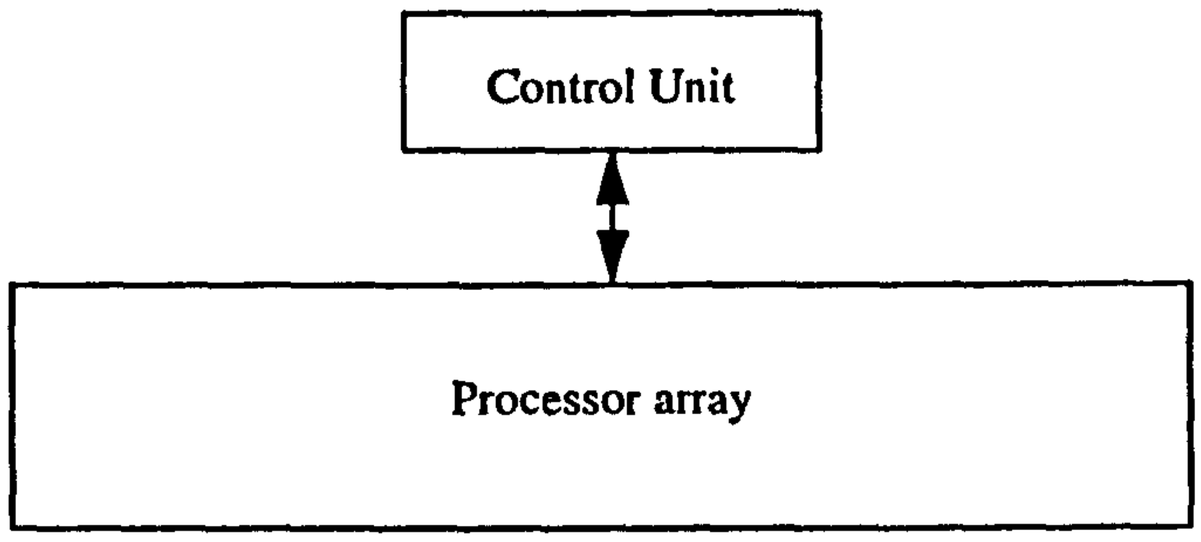


FIGURE 1.4.An array processor

The processors in an array processor usually consist of a bit-serial ALU and some local memory. The processors are arranged in a regular lattice of two or more dimensions with each processor connected to at least its nearest neighbour. In the case of two dimensional problems such as image processing and matrix calculations the data can be mapped easily onto a two dimensional array.

1.4.4 Multiprocessors

Multiprocessing machines consist of multiple complete processors which each contain a CPU, ALU, local memory and an I/O interface (known as a node). This is the ideal approach as it should theoretically allow you to carry out any kind of computation in parallel. However as stated previously, the structure of the computation and the degree of inter-processor communication necessary, must be considered.

1.4.4.1 Shared memory multiprocessors.

This type of multiprocessor exchange data via a shared memory (See Figure 1.5 on page 8). Each node still has its own local memory but uses shared memory for data that is required by other nodes. Since the nodes operate more or less independently of each other, this is an asynchronous architecture. A disadvantage of this architecture is that it

is not easily scalable as if more nodes are added then the shared memory bus becomes a potential bottleneck.

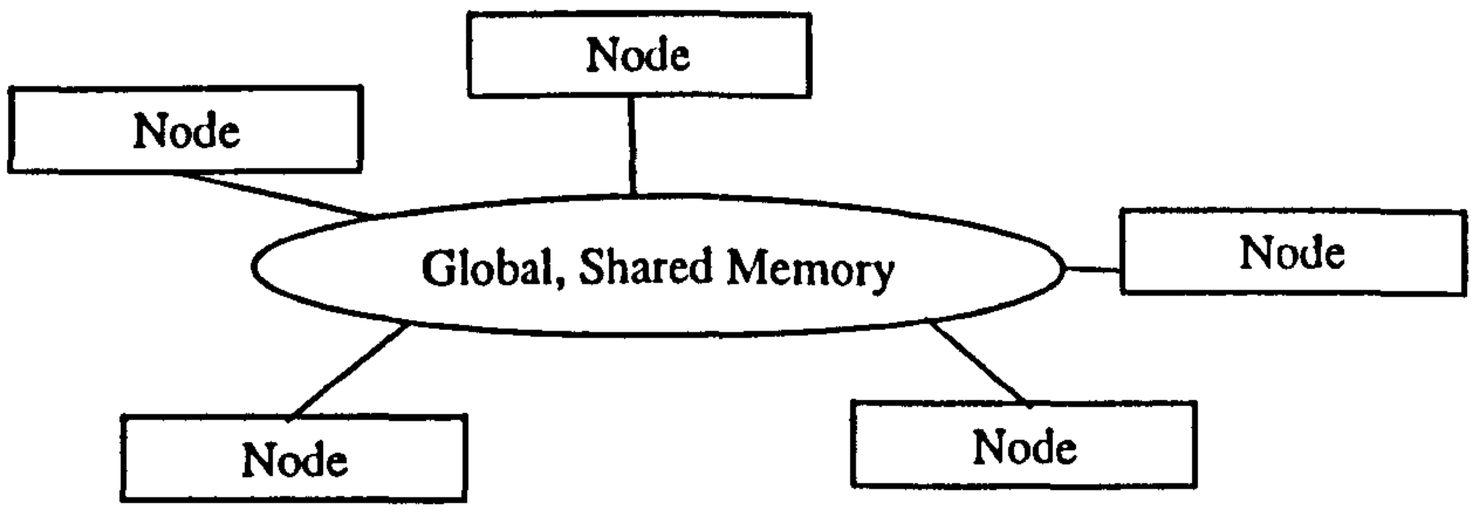


FIGURE 1.5.Shared memory multiprocessor

1.4.4.2 Distributed memory multiprocessors

In these systems each node has its own memory and the nodes communicate via an interconnection network (See Figure 1.6). Ideally every node would be directly connected to every other node but this is usually not feasible especially in multiprocessors with a large number of nodes (1000 nodes requires 1/2 million connections). Usually messages pass via intermediate nodes to reach the destination node (known as message passing). Obviously when designing the network the aim is to minimise the time taken for messages to pass over it.

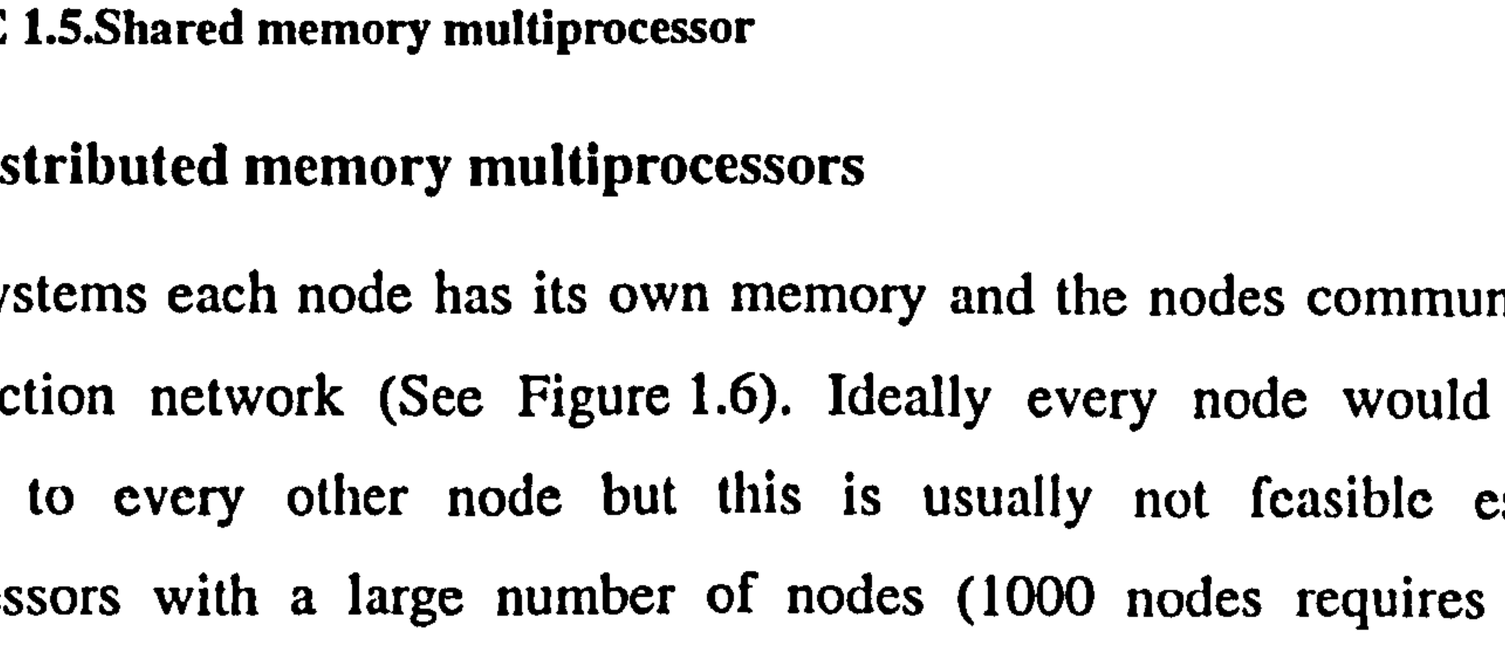
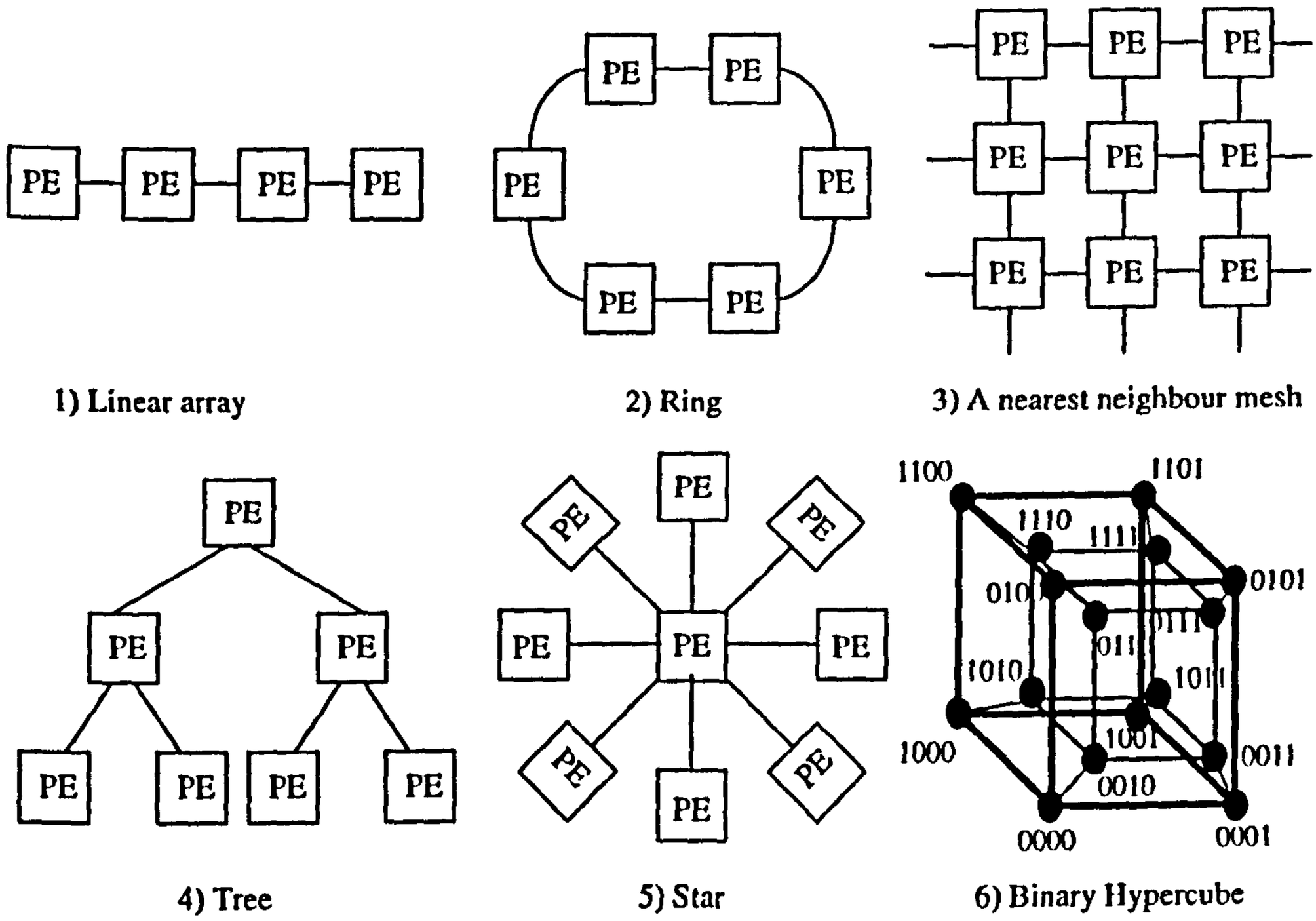


FIGURE 1.6.A distributed memory multiprocessor

The interconnection network can be static or dynamic[2]. A static network topology does not change after the machine has been built whereas a dynamic network can change its topology to suit different computations. The topology can be altered before the computation or dynamically during the computation. Static networks are more appropriate for problems where the communication pattern can be predicted reasonably well, whereas dynamic topologies are suitable for a wider class of problems.

Some of the common static topologies are illustrated in Figure 1.7. In these static networks messages 'hop' from node to node in order to reach the destination. In a simple 1-D linear network the average number of hops is N/3 where N is the number of nodes. The number of hops required can be reduced by increasing the dimensionality of the network. In a ring topology (a 2-D linear network) for example the number of hops is reduce to N/6 (half that of a 1-D linear network). By increasing the dimensionality of the network however, the number of connections required between the nodes increases and hence the cost and complexity increases.



1) Linear array

2) Ring

3) A nearest neighbour mesh

4) Tree

5) Star

6) Binary Hypercube

FIGURE 1.7.Common static network topologies.

A compromise between the number of links and the number of hops is to use higher dimensions and only connect nodes in the same dimension. This is the approach used in the binary hypercube. The hypercube illustrated in Figure 1.7 is a four dimensional hypercube. It is so called as four binary digits are required to specify all the node positions. Each node is connected to every other node whose binary number differs from its own by exactly one digit.

If n is the dimensionality of the hypercube then N(no. of nodes) = 2^n and the maximum number of hops required is $\log_2 N$ which is equivalent to n. The number of connections

at each node is also $\log_2 N$. The hypercube topology has been used in many commercial machines some of which will be discussed at the end of this Chapter.

There are three basic types of dynamic network: bus networks, multistage networks and crossbar networks (listed in order of increasing performance and cost).

A bus network, as the name implies, is a system where all the nodes are connected to a common bus, therefore any node can communicate with any other node. The main advantage of this system is its simplicity. However a major disadvantage is that it can only be used for a limited number of processors because of the limited bandwidth of the bus.

A crossbar switch is an integrated circuit (IC) which when combined can connect any input to any output (See Figure 1.8). Each output is connected to the output of an *n* to 1 multiplexer where *n* is the number of inputs to the crossbar switch. The *n* inputs of each multiplexer are connected to the *n* inputs of the crossbar allowing each output to be connected to any input of the crossbar. Several connections between inputs and outputs can be present at the one time.

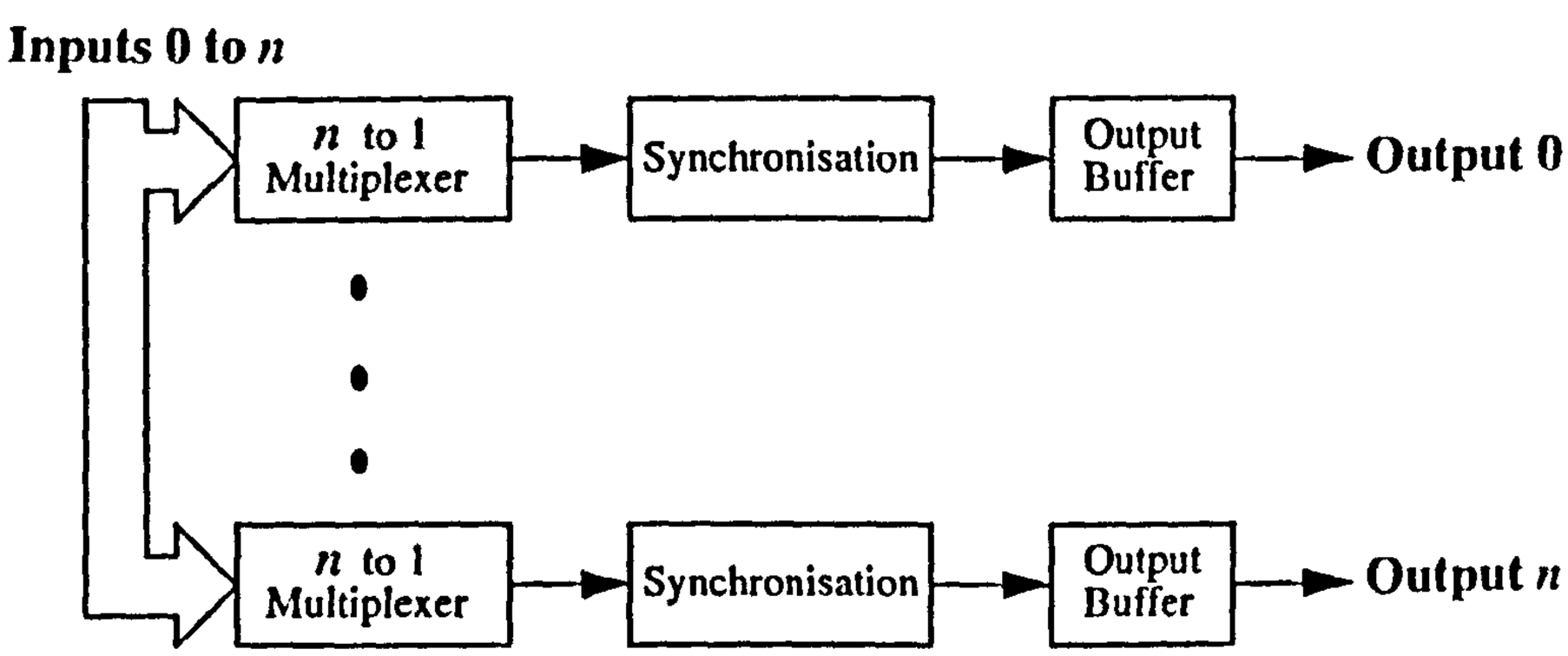


FIGURE 1.8.A crossbar switch

By connecting nodes to a crossbar switch any node can be directly connected to any other node. The crossbar switch can be programmed prior to or during a computation. Crossbar switch systems are only usually suitable for a small number of nodes as the number of logic switches within the crossbar is N^2 where N is the number of processors (usually crossbar switches are 32-to-32 or 64-to-64).

A multistage network attempts to provide the connectivity of a full crossbar by using several 2-to-2 (maybe larger) crossbars connected together. The reason for this is to reduce the number of switching elements required and hence the cost of the system. For an N node system the number of switching elements is $N\log_2 N$ compared to N^2 for a single crossbar switch. However, since a message will need to pass through several switches to reach its destination the latency of such systems is greater than for a single switch.

Figure 1.9 shows an example of a multistage network using several 2 to 2 crossbars. This configuration allows any of the eight inputs to be connected to any of the eight outputs. Since each crossbar has four switching elements the total number of switching elements required is forty eight (4 x 12) compared to the sixty four required by an 8 to 8 crossbar switch. Multistage networks do have the disadvantage however that messages can be blocked as two different routes through the network may require the same connection on one of the crossbars.

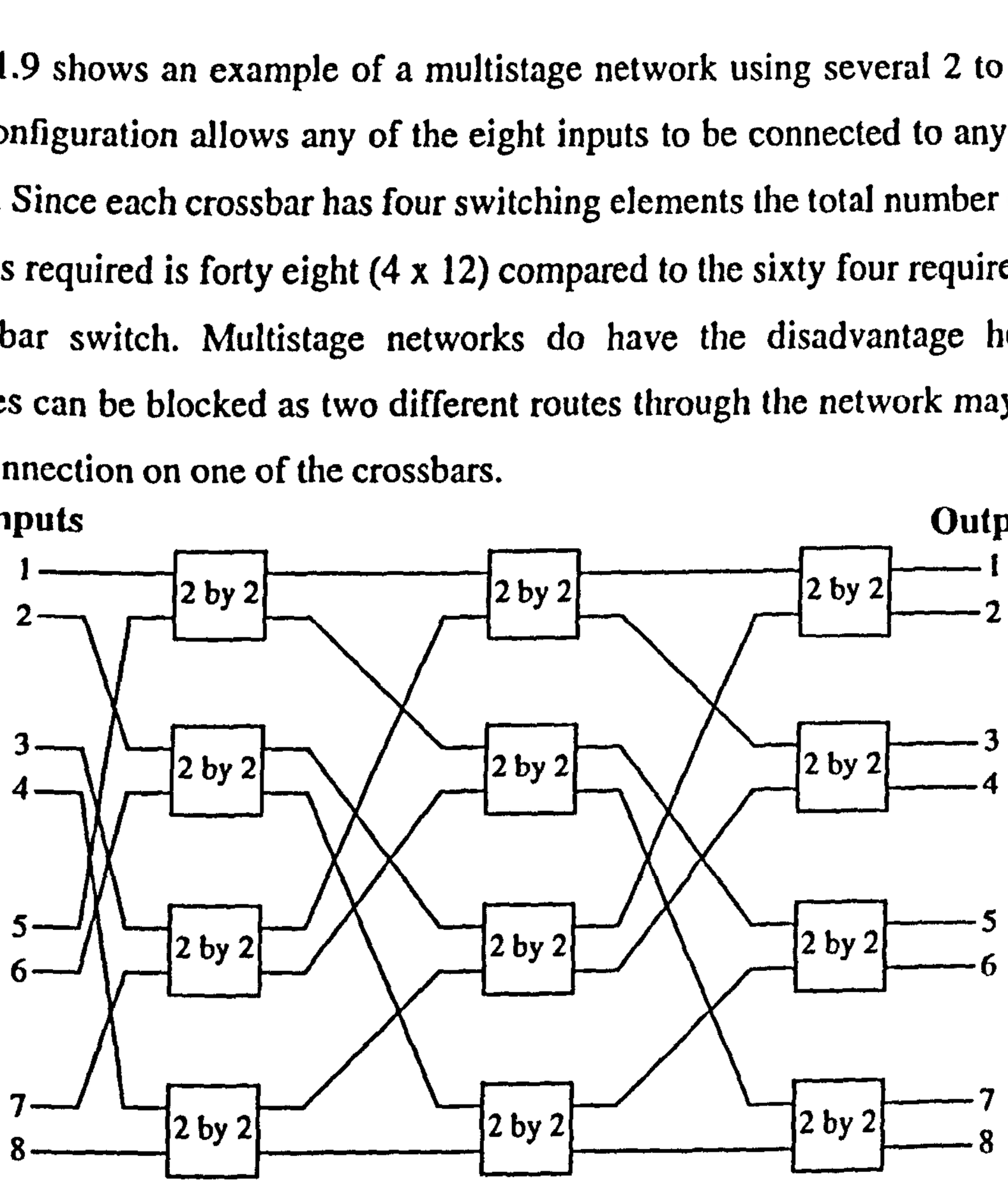


FIGURE 1.9.A multistage network

1.4.5 Multi-Workstations

Multi-workstations in their simplest form are collections of high performance workstations, such as Sun or Silicon Graphics, connected together by ethernet. A program is distributed over the workstations and messages are exchanged between the workstations via ethernet. An advantage of this type of system is that it can utilise existing general purpose hardware. A disadvantage however, is the relative slowness

of ethernet compared to the dedicated high-speed links on a multiprocessor machine. The communication speed between the workstations can be increased by using optical links.

1.4.6 Which parallel methodology?

None of these approaches to parallelism is necessarily the best approach. It is dependent on the type of problem the parallel machine is used for and the cost/ performance ratio required. For example, traditionally supercomputers use pipelined vector processing and rely on the fastest available (expensive) circuit technology to produce high performance. These machines however are only suitable for high speed numeric problems. On the other hand multiprocessors do not require exotic circuit technology or custom processor designs which provides flexibility, familiarity and scalability.

1.5 Taxonomies for parallel computers

Several taxonomies have been developed to classify the various types of parallel computer. The main reasons for their development are:-

- they show what has been achieved to date in the field of architecture.
- they can enable the designer to estimate the suitability of an architecture to solving a given problem.
- there is the potential that such systems may reveal configurations that may not have occurred to designers.
- performance models can be built that cover a wide range of systems with little, or no, modification.

1.5.1 Flynn's Taxonomy

The most widely used taxonomy was developed by Flynn in 1972[3]. This classifies parallel computers into four groups:-

- SISD - Single Instruction Stream Single Data Stream.
- SIMD - Single Instruction Stream Multiple Data Stream
- MIMD - Multiple Instruction Stream Multiple Data Stream
- MISD - Multiple Instruction Stream Single Data Stream

SISD computers are the sequential (Von Neuman) machines where a single stream of instructions acts upon a single stream of data.

A SIMD machine consists of an array of processing elements each carrying out a single instruction simultaneously but on different data sets. An illustration of the principle of a SIMD machine is shown in Figure 1.10. All the people (nodes) are carrying out the same instruction (walking) in lock-step time and are controlled by a leader (master processor).

FIGURE 1.10.Analogy of a SIMD machine

A MIMD machine comprises a number of processing elements all executing their own code simultaneously on different data sets. A diagram illustrating this concept again using the analogy with people is shown in Figure 1.11. In this case every person (node) is carrying out a different task (instruction) using different items (data).

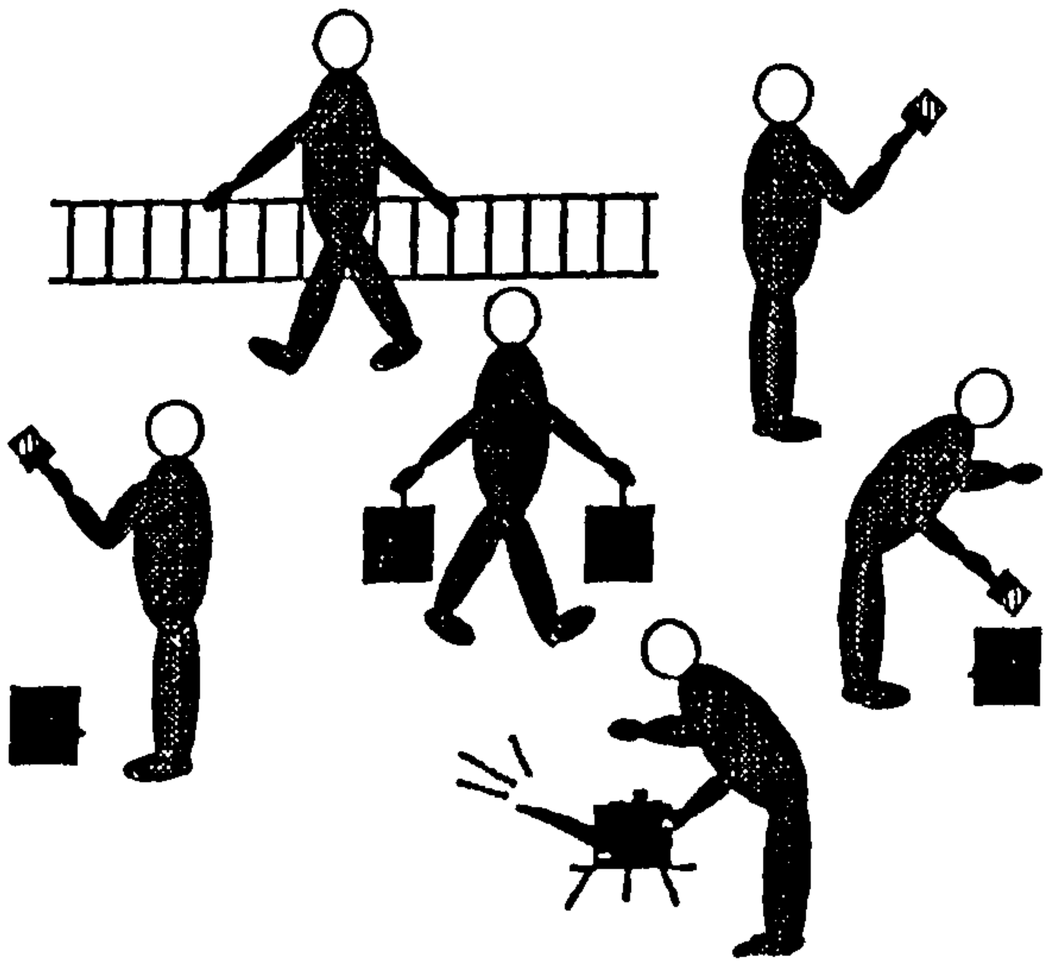


FIGURE 1.11.Analogy of a MIMD machine

MISD computers are theoretically possible but would imply that a set of different instructions would all be performed simultaneously on the same data item which is an unlikely scenario.

Flynn's classification is useful in certain circumstances but it fails to accurately describe some systems. For example a pipelined vector machine can either be described as SISD or SIMD, SISD if considered as processing a single stream of data and SIMD if every element of the vectors is regarded as belonging to an individual stream of data.

Generally though a SIMD machine is taken to be an array of processors operating under central control and an MIMD machine is regarded as an array of processors operating independently of each other executing different instructions on different data streams.

1.5.2 Feng's taxonomy

This is a performance based classification which describes the parallelism of a set of processors in terms of the number of bits than can be processed simultaneously[4]. Parallel machines are defined by the word length of the processing units (n) and the bit slice length (m - a product of the number of pipelines and their depth). This provides the following classification:-

- WSBS - Word Serial, Bit Serial (bit serial processing) - m = 1; n = 1
- WPBS - Word Parallel, Bit Serial (bit slice processing) - m > 1; n = 1
- WSBP - Word Serial, Bit Parallel (word slice processing) - m = 1; n > 1
- WPBP - Word Parallel, Bit Parallel (fully parallel) - m > 1; n > 1

This classification is useful for pipeline and vector processors but would not distinguish between types of multiprocessor architecture.

1.5.3 Händler's Taxonomy

Händler identified three logical levels of parallelism[5]: Program level (multiple processors), Instruction level (multiple ALUs) and the Word level (multiple bits). The Händler classification system therefore uses the triple (K, D, W) to represent a machine, where K is the number of processors, D is the number of ALUs and W is the

wordlength of each ALU. On top of this, pipelining can be included (macro-, instruction- and arithmetic-pipelining respectively), giving rise to (K*K', D*D', W*W'), where the multipliers are the pipeline depth at each level.

The system also enables representations to be combined using the following operators:

+ indicates the existence of more than one structure that operates independently in parallel.

* indicates the existence of sequentially ordered structures where all data is processed through all structures.

v indicates that a certain system may have multiple configurations.

This works well for describing conventional vector processors but it fails to describe the interconnection information in multiprocessor systems.

1.5.4 Skillicorn's taxonomy

Skillicorn introduced the idea of modelling the possible interconnection networks within a system[6]. The networks include the processor to memory, processor to ALU, and processor to processor subsystems. The system is therefore represented by the following:

1) no.of instruction processors (IP).

2) no. of instruction memories (IM).

3) the IP to IM network.

4) no. of ALUs (DP)

5) DP to data memory network.

6) IP to DP network.

7) DP to DP network.

The networks are described by abstract switches which connect the functional units together. These abstract switches can be implemented in different ways: by buses,

dynamic switches, or static interconnection networks. Four different forms of abstract switch connect functional units together:-

- 1-to-1 : a single functional unit of one type connects to a single functional unit of another
- n-to-n : the ith unit of one set of functional units connects to the ith unit of another. This type of switch is a 1-to-1 connection replicated n times.
- 1-to-n : in this configuration, one functional unit connects to all n devices of another set of functional units.
- n-by-n : in this configuration, each device of one set of functional units can communicate with any device of a second set and vice versa.

Further discriminations can be made by describing whether or not each of the processors is pipelined and by giving its internal functional structure by a state diagram.

This system is very detailed and flexible, and is capable of describing most current systems. However it is slightly complex and is probably best used in combination with Flynn's system so that only the departures from the base class need to be specified.

These are only some of the taxonomies that have been proposed. Skillicorn's comes closest to the ideal as it includes the interconnection topology of nodes. However it still does not cover all the topologies available as it only uses simple one to one or all to all models to describe the interconnection networks. Depending on the type of system in use (i.e. pipelined, vector etc.) the best approach is to classify the system using a combination of Flynn's taxonomy and one of the others (i.e Händler's for a vector processor).

1.6 Parallel Software Engineering

The development of parallel software is governed chiefly by the target hardware and the nature of the application. Hardware can vary from a small pipeline to a large multiprocessor machine containing thousands of nodes. The application can vary from a large numerical problem such as weather forecasting to a small real time embedded system. Depending on the hardware and the application different requirements are demanded of the software.

The main aims of the software engineer are to balance the computational load and to minimize the communication to computation ratio. It is not advantageous to have one node very busy while the others are idle or to have so much communication that the nodes spend most of their time communicating rather than computing.

Software engineers also may be required to consider issues such as portability and scalability. For some applications the hardware setup will not alter during the lifetime of the software (i.e embedded and process control systems) but for the majority of systems it is desirable to allow for the possible implementation on other parallel systems, and also to provide for the scaling up of the existing target hardware.

The operating system on a parallel machine provides the same services as on a sequential machine (i.e memory management, device I/O), as well as managing interprocess communication and synchronisation. The operating system may also be responsible for the allocation of processes onto nodes.

1.6.1 The basic principles of software engineering for parallel machines.

There are three major steps to producing a parallel program[7]:-

1) Decomposition

2) Mapping

3) Tuning.

Figure 1.12 overleaf illustrates these processes[8]. Decomposition is the partition of the application into a set of parallel processes and data. Mapping is the distribution of the processes onto the nodes. Tuning is the alteration of the working application to balance the load and to optimise performance.

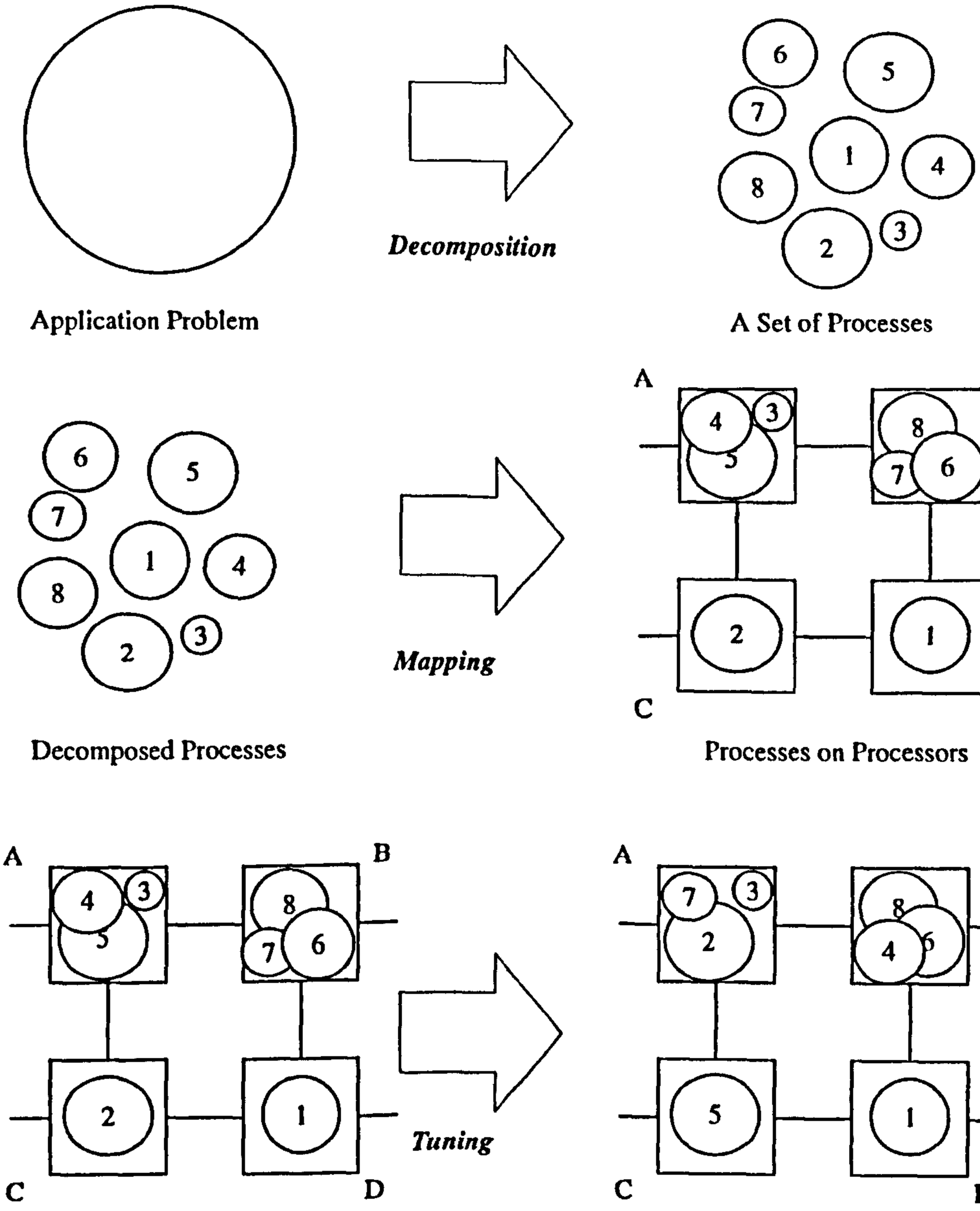


FIGURE 1.12.The main stages in producing a parallel program.

1.6.1.1 Decomposition

Decomposition is the first and most important step to producing a parallel program. It guides the whole programming process. The decomposition of an application must break up the program into a set of well defined processes that can be linked together logically to provide a finite solution to a computation.

In order to choose the best decomposition method for an application an understanding of the application problem, the data domain, the algorithms used and the flow of control in the application are required.

There are three general decomposition methods:-

- Perfectly parallel decomposition
- Domain Decomposition
- Control Decomposition

1.6.1.2 Perfectly Parallel Decomposition

Perfectly parallel applications can be divided up into a set of processes that require little or no communication with one another. Application of this type are usually the easiest to decompose.

An obvious way to implement perfect parallelism is to run equivalent sequential programs on several nodes but on different data sets. If this type of algorithm was executed on a single processor then each data set would have to be considered one at a time whereas by using several nodes almost linear speed-up can be achieved with little effort required by the programmer.

Examples of perfectly parallel applications can be found in most disciplines. An example from physics is the use of the Monte Carlo technique to determine atomic structure. Physicists analyse thousands of random electron distributions around an atomic nucleus to define a probability distribution that points to the probable atomic structure. Each random electron distribution can be calculated independently in parallel making this a perfectly parallel application.

1.6.1.3 Domain Decomposition

Problems subject to domain composition are usually characterised by large, discrete static data structures. It is the fundamental data structure that controls how the program is parallelised. For example calculations involving matrices could be parallelised by dividing the matrix into columns and separate nodes could execute different sets of instructions on different columns as required in a Gauss Elimination for example.

1.6.1.4 Control Decomposition

Control decomposition is for applications where no static or fixed domain is identified but instead it is the flow of control or operations that is used as the guideline for parallelism. As the development progresses, the data structures are also distributed but the focus of the parallelisation still remains the flow of control.

Functional decomposition is a method of control decomposition. Here, the problem is regarded as a set of operations (in terms of its functions) and the processes for the nodes are based on those operations. Figure 1.13 illustrates a functional decomposition model of an algorithm.

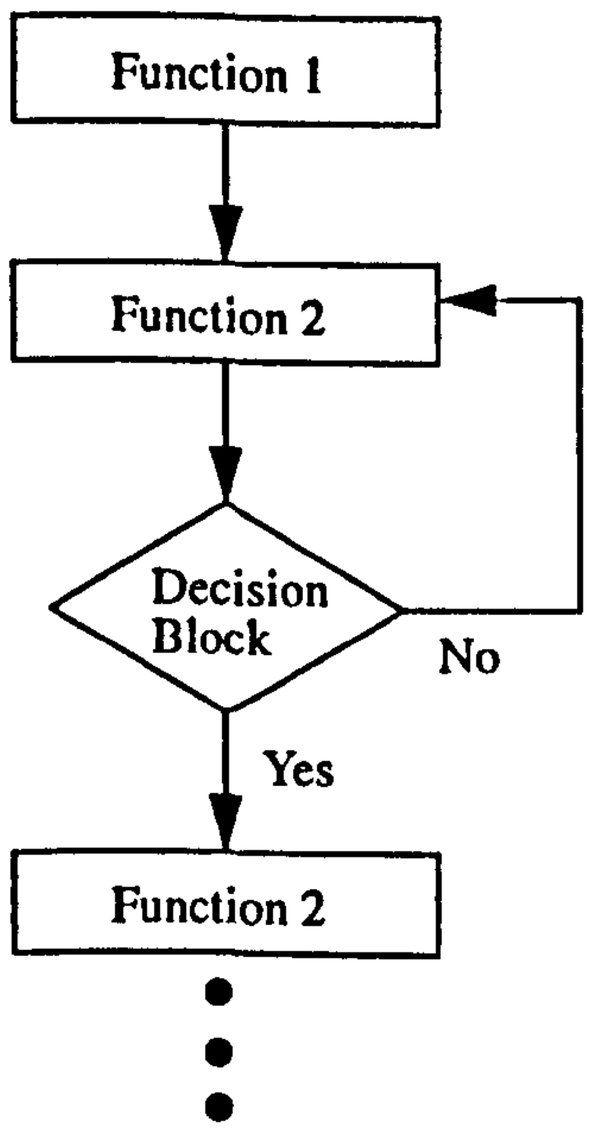


FIGURE 1.13.Functional Decomposition Model

The flow of control is indicated by the lines between the boxes. For small problems the functions are usually required to be executed sequentially therefore a parallel application is not produced easily. However, large problems usually have a large degree of overlap between functions so it is possible to extract some sort of parallelism.

The most common type of functional parallelism is where the data is pipelined from one module to another creating what is called a large-grain pipeline. An example of this method can be found in image recognition. The traditional approach to image recognition includes the following steps:

1) Preprocessing to reduce noise.

2) Edge and region detection.

3) Object Recognition.

4) Object grouping.

5) Screen interpretation.

By dedicating a node (or more likely a group of nodes) to each step, the stream of input frames could be pipelined through the above five steps. The number of nodes assigned to each step would be determined by analysis and experimentation.

Another method of control decomposition is the manager/worker approach. This involves dividing the application into tasks (without attempting to make the tasks of equal size) and then using one of the nodes (the manager) to distribute the tasks to the other nodes (the workers) as they become available. The manger's job is to assist or create the pool of jobs to be done, and then to keep the workers busy by assigning jobs to workers. The manger also usually returns the final results based on the full results of the individual workers.

1.6.1.5 Granularity

Granularity is the level of parallelism which is a measure of the degree to which tasks are partitioned into subtasks (i.e effectively the degree of decomposition). Parallel systems can be fine-grained, medium grained or coarse grained. The "grain" of a computation can be measured by the amount of computation between tasks. An example of fine-grained parallelism would be the execution of a DO loop in parallel whereas course-grained parallelism is where large sections of code are executed in parallel.

The granularity of a system relies on the number of processors to be used and the nature of the problem decomposition. Often there can be abundant parallelism at fine granularity which is not exploited as working with fine granularity increases the amount of data communication between processes. It also increases the software complexity.

1.6.1.6 Mapping

Decomposition is followed by the distribution of the processes onto the nodes which is known as mapping. Ideally the processes should be allocated to the nodes in a manner which keeps all the nodes busy during the entire time the computation is running. Processes can be allocated dynamically during program execution or statically before the execution of the program. The less equal the loads on the nodes the more the computing resources of the system are wasted. Well balanced mapping relies on the modularity acquired from the problem decomposition.

1.6.1.7 Tuning

Once an application has been mapped to the nodes of a system and it is running properly, the next step is to tune it to enhance the performance. Tuning usually involves attempting to the reduce the communication to computation ratio as this is one the main overheads in parallel computing. This could involve altering the mapping of the processes onto the nodes or altering the decomposition of the application.

1.6.2 Operating Systems

In addition to providing the services of a normal OS (operating system) on a sequential machine, the OS on a parallel machine must provide such services as program scheduling and interprocess communication and synchronisation. Some of the operating systems developed for parallel systems are simply extensions of uniprocessor OSs such as UNIX whereas some OSs have been developed especially for multiprocessors such as Helios developed for transputers.

There are four basic designs that have been used for multiprocessor operating systems[1]:-

- master/slave
- separate executive for each processor
- symmetric treatment of each processor
- distributed operating systems

In the master/slave approach the OS is permanently assigned to one particular processor and always operates in that processor. If a slave processor requires service, that service can only be provided by the executive. The slave must interrupt the

executive and request service. It must then wait until the program currently being executed is interrupted and the executive is dispatched to the slave processor.

The main advantage of this type of system is that interprocessor communication and sychronisation can be very simple and well defined. A major disadvantage however is that the system is subject to catastrophic failure in the case of a failure in the master processor or at least severe degradation in the case of a failure in a slave processor.

A separate executive system, is where every processor has a copy of the OS. In this configuration each processor can service its own needs. Therefore, no service requests or service from a single executive are required. As each processor has its own copy of the OS, the system is much less sensitive to catastrophic failure. A failure of one or more processors will cause a proportional loss of system capability, but will not bring down the entire multiprocessor system.

A symmetric system maybe thought of as a master/slave type system where the master floats form one processor to another. This is the most difficult method of operation both from a design and from an operating viewpoint. It does however have the advantage that it provides the most efficient use of available system resources (e.g. I/O devices and any central memory).

In a distributed OS the various OS utilities and functions are distributed among the various processors. Each processor is dedicated to a particular utility or function and together they implement all OS functions.

Helios[9,10] is an example of a distributed operating system which uses the client/server model for operating systems. A client process wishing to access a system resource, such as opening a file, sends a message to a server process requesting this action to be performed on its behalf. The client and server processes may reside on different processors whereas in a single CPU machine the client and server would of course be on the same processor.

Each processor node contains a Helios kernel, which handles memory management and message passing. Each node also contains two servers: the processor manager and the loader. The processor manager is responsible for process creation within that

processor and other housekeeping jobs. The loader handles the loading and unloading of both program modules and resident libraries which are loaded on demand.

Other servers run on one or several processing nodes. Some servers must run on nodes with particular hardware attached. For example the file system needs the disc device connected while a window manager must run on a processor with video memory attached. Servers with no specific hardware requirements are distributed to share the load evenly amongst the processors.

An I/O server is provided by Helios which runs on the host machine of a parallel system. This causes the host machine to appear to the network of nodes just like another node running Helios. The I/O server communicates with the host operating system to provide such things as access to the file system and serial ports.

Helios provides a task, called the Helios Shell, that acts as a command line interface to the operating system. The shell commands are similar to Unix shell commands. The standard Unix-like file manipulation commands such as ls, mv, rm and so on are supported by Helios.

The Task Force Manager (TFM) is a distributed server used by Helios. This consists of a number identical servers distributed throughout a network of nodes, each controlling a different area of the network. The TFM processes all client level task force (the programs to be distributed over the network) execution requests. It analyses the current state of the network and distributes the component tasks of the task force to the most suitable processing elements. The criteria for the distribution include the resource requirements of particular component tasks, connectivity of the task force, and the current status of the network.

The prime means of communication under Helios is through message passing implemented by the kernel. In order to provide transparency the semantics of message passing require to be the same regardless of whether the destination is in the same processor or in another. The user callable routines **PutMsg** and **GetMsg** are responsible for the sending and receiving of messages whether they are on the same processor or not.

Helios was specifically designed to run on a network of transputers (a single chip processor designed for multiprocessing). This makes programs written using the Helios environment less portable to other architectures. However, the Unix like command line interface makes the OS easier to use for Unix users.

1.6.3 Parallel Development Tools

As parallel computing has become more popular and accessible the evolution of software tools for parallel computing has accelerated. Debuggers, compilers, and languages are available for parallel systems.

1.6.3.1 Parallel Languages

Parallel versions of sequential languages such as Fortran and C have been developed. These can be helpful when converting existing sequential code onto a parallel system, as usually large sections of the code will remain unchanged and it is only the parallel constructs that require to be added.

Special purpose parallel languages such as Occam, which was developed for the transputer, also exist. These can be combined with parallel Fortran and C to produce mixed language programming which is also useful when porting an application from a sequential platform to a parallel system.

1.6.3.2 Compilers

Automatic parallelisation compilers attempt to identify the elements in existing sequential code that are candidates for parallel computation, and produce compiled code for the specific multiprocessor machine. This approach however, usually gives inefficient code which produces disappointing speed-ups in programs. These compilers can be useful though to give the software engineer an idea of the parts of the program that can be executed in parallel.

1.6.3.3 Parallel Programming Environments

Parallel programming environments consist of a set of tools for parallel program development. The tools may include a subroutine library supporting parallel programming, debuggers, and performance analysis tools. The subroutine libraries

provide some of the same services as operating systems do for parallel machines. The difference however is that with an operating system its services can be accessed from the command line whereas the subroutine libraries are only accessed from calls within a program.

An example of such a programming environment is Express developed by the Parasoft Corporation[11]. This supports two basic models of parallel programming: the host/node model and the cubix model. In the host/node model the application program is divided up into two parts, one for the host machine and one for the parallel machine (See Figure 1.14). In the Cubix model the entire application is executed on the parallel machine (See Figure 1.15).

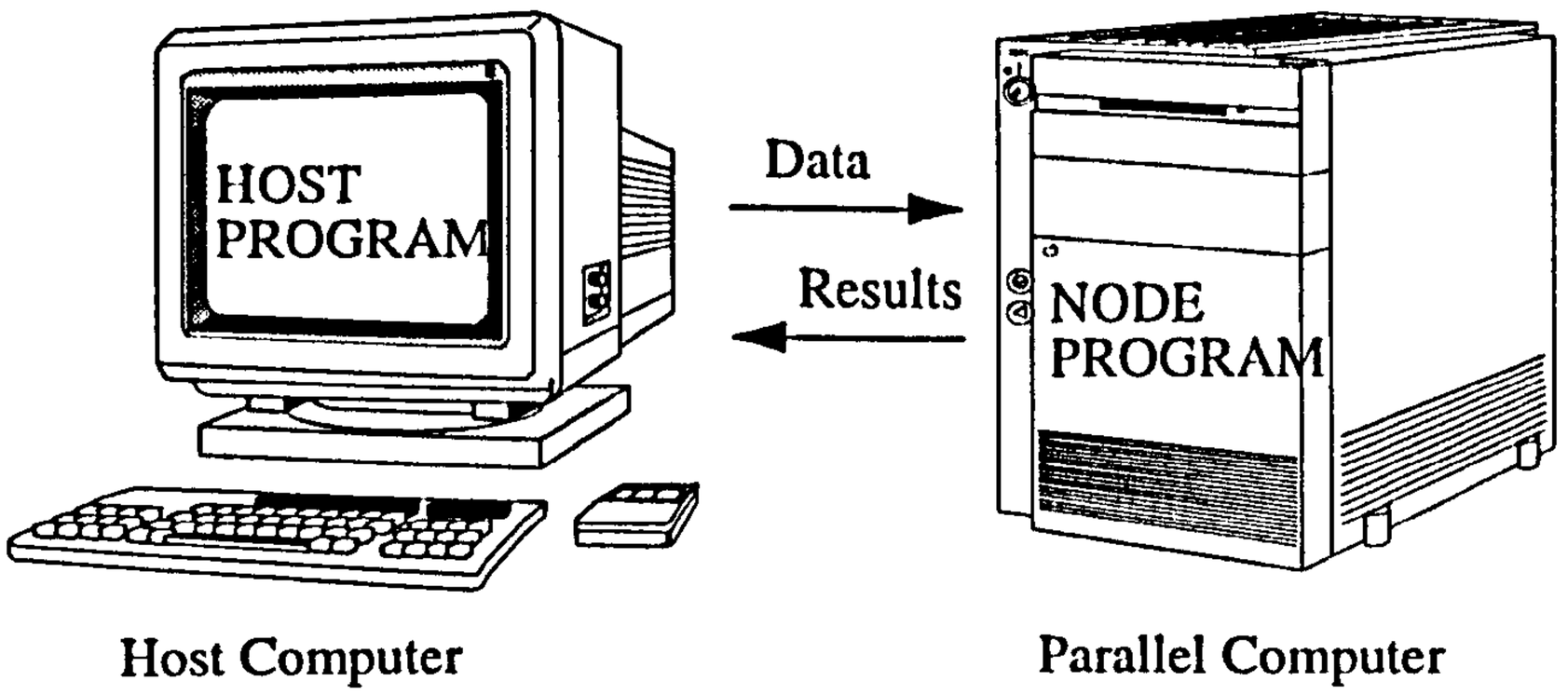


FIGURE 1.14.Host/Node programming model

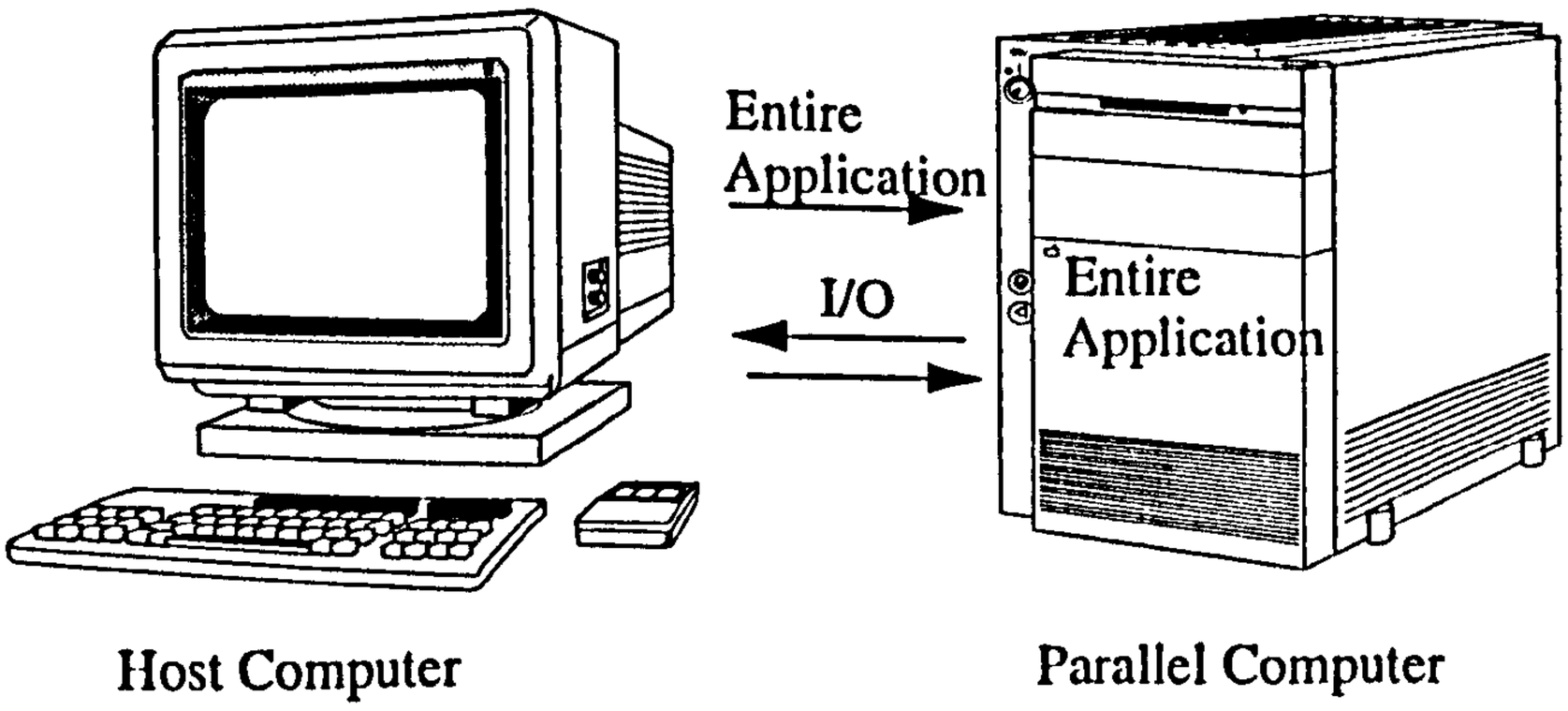


FIGURE 1.15.Cubix programming model

Cubix is the name of the I/O server which loads a program onto the parallel machine and starts it running. It also performs the system services requested by the nodes. However it only provides basic operating system facilities to the node programs. If the program needs to have direct, low level access to a peripheral device then the host/ node model is required.

In the host/node model the computationally intensive aspects of an application are extracted and executed on the parallel machine. The interface and control portions of the code remain on the host machine. All communication between host and nodes and among the nodes is done with Express system calls. The node programs are loaded onto nodes by function calls from the host machine.

Express provides a library of subroutines which supports low level communication primitives for sending messages between processors, peripherals and other system components. Utilities are also included which provide such facilities as broadcasting code/data onto the nodes and data redistribution. Figure 1.16 shows some of the routines available[12].

```
KXINIT      Start up Express and initialise XPRESS common block
KXLOAD      Load program onto all nodes
KXOPEN      Allocate a group of processors
KXSTAR      Start execution of a node program
KXREAD      Read a message
KXWRIT      Write a message
KXTEST      Test for an incoming message - non-blocking
KXBROD      Interprocessor broadcast
KXHAND      Install asynchronous message handler
KXRECV      Read a message - non-blocking
KXSEND      Send a message - non-blocking
```

FIGURE 1.16.A sample of the Express routines

The Express environment also provides a parallel graphics library, a debugger and a system for analysing such matters as subroutine usage, communication overheads, load balancing, interprocessor timing differences etc. Express is available in both Fortran and C for many hardware platforms (INTEL iPSC2, iPSC/i860, CRAY X-MP etc.).

Another programming environment is PVM (Parallel Virtual Machine)[13]. This is a an integrated set of software tools and libraries, designed to link separate host machines to form a "virtual machine" which gives an illusion of a single manageable computing resource. The virtual machine can be composed of hosts of varying types, in physically remote locations. The system is portable to a wide variety of architectures, including workstations, multiprocessors, supercomputers and PCs.

The PVM computing model divides an application into several tasks. Each task is responsible for a part of the application's workload. The tasks may be performing the same operations on different data sets or performing completely different operations on separate data sets. The user views the complete application as a set of communicating tasks and it does not matter where the tasks are executed.

The application's computational tasks execute on a set of machines (the host-pool) that are selected by the user for a given run of the PVM program. Both single-CPU machines and hardware multiprocessors may be part of the host pool. The host pool may be altered by adding and deleting machines during operation.

The PVM system is composed of two parts: a daemon and a utilities library. The daemon (called pvmd3) is a program which resides on all the computers making up the virtual machine. A user wishing to run a PVM application creates a virtual machine by starting up PVM. The PVM application can be started from a command line prompt on any of the computers in the system.

The PVM library contains user-callable routines for message passing, spawning processes, coordinating tasks and modifying the virtual machine. Typically a user writes one or more sequential programs in C, C^{++}, or Fortran 77 that contain embedded calls to the PVM library. Each program corresponds to a task making up the application.

These programs are compiled for each architecture in the host pool, and the resulting object files are placed at a location accessible from machines in the host pool. To execute an application, a user typically starts one copy of one task by hand from a machine within the host pool. This process subsequently starts other PVM tasks, eventually resulting in a collection of active tasks that then compute locally and

exchange messages to solve the problem. Figure 1.17 and Figure 1.18 show two communicating PVM tasks.

```
main()
{
    int cc, tid, msgtag;
    char buf[100];

    printf("i'm t%x\n", pvm_mytid());

    cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
    if (cc == 1) {
        msgtag = 1;
      pvm_recv(tid, msgtag);
      pvm_upkstr(buf);
      printf("from t%x: %s\n", tid, buf);
    } else
      printf("can't start hello_other\n");

    pvm_exit();
}
```

FIGURE 1.17.PVM program *hello.c*

```
#include "pvm3.h"
main()
{
    int ptid, msgtag;
    char buf[100];

    ptid = pvm_parent();

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
     msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);

    pvm_exit();
}
```

FIGURE 1.18.PVM program *hello_other.c*

This program *hello.c* is intended to be invoked manually. After printing its task id (supplied by the daemon pvmd3 and received from the function `pvm_mytid()`), it initiates a copy of the program *hello_other* (Figure 1.18) using the `pvm_spawn()` function. A successful spawn causes the program to execute a blocking receive using `pvm_recv()`. After receiving the message sent by *hello.other*, the program prints the message as well as the task id of *hello_other*. The buffer is extracted from the message

using `pvm_upkstr`. The final `pvm_exit()`call dissociates the program from the PVM system

The program *hello_other.c* is the "slave" or spawned program. Its first PVM action is to obtain the task id of the "master" using the `pvm_parent()`call. The program then obtains its hostname and transmits the complete string to the host: `pvm_initsend` initialises the send buffer, `pvm_pkstr(buf)` places a string into the send buffer and `pvm_send` transmits the contents of the send buffer to the task specified by `ptid`.The message is tagged with the number 1 by `msgtag`.

PVM is public domain software and is available via the internet. PVM libraries are available for C, C++, and Fortran. It has also been used with other languages, such as Lisp. The most common PVM platform is a Unix machine, however it is relatively simple to port it to other platforms such as Intel iPSC/860, iPSC/2 etc.

In general though, for software engineers porting an application onto a parallel system is still much more cumbersome than doing so onto an established system. This is partly due to the lack of standardisation in architecture, operating systems, languages etc. It is also inherent in our teaching that we think of code in a sequential manner and it is not natural to think of code in a parallel manner. These problems will only be overcome with the general acceptance of parallel computers.

1.7 Key Developments in Parallel Computing.

1.7.1 The earliest parallel machines

The concept of parallelism in computing began as early as 1953 with the advent of bit-parallel arithmetic rather then bit-serial as had been the case. The IBM 704 was the first commercial machine with floating-point hardware and was capable of 5kFLOPS (FLoating point Operations Per Second)[14].

Functional parallelism increased throughout the 50's and early 60's with the release of such computers as the IBM STRETCH which included two parallel memory banks and instruction execution pipelining. One of the best known pipelined computers was the CRAY-1 developed by Seymour Cray. It was a vector computer operating on 64-bit floating point numbers with a listed peak performance of 160 MFLOPS.

1.7.2 The first SIMD machines.

As the limits were reached in what could be achieved in parallel on a sequential machine, the idea of multiprocessing surfaced. This began with array processing where several processing elements were under the control of a single control unit. This was the approach used in the ICL DAP (Distributed Array processor) which consisted of a 64x64 array of bit-serial processors, each with 4 Kbits of memory.

One of the earliest SIMD machines was the ILLIAC IV designed in 1968[2]. This contained 64 processing elements arranged as an 8-by-8 array with each PE connected to its four nearest neighbours. Each PE was capable of 4MFLOPS giving a theoretical maximum performance for the whole machine of 1000MFLOPS (of course this was never obtained). The machine contained many pioneering design concepts which are still relevant today. One of the lessons learned from the ILLIAC IV was that it assumed too much regularity in communication (i.e an 8x8 array) than was present in most problems.

A SIMD machine which allowed greater flexibility in communication than the ILLIAC IV was the CM-1 Connection Machine manufactured by Thinking Machines Corporation in 1986. This consisted of 65,536 1-bit processors connected in a 256x256 grid; in addition, clusters of 16 processors were also interconnected in a 12-dimensional hypercube network for routing messages, and the 16 processors within a cluster were linked in a daisy chain fashion.

1.7.3 The first MIMD machines

The idea of multiprocessor systems where each processor would have it's own instruction stream began to emerge in the early 1970s. One of the major designs was the C.mmp machine developed at Carnegie Mellon University. This used 16 DEC PDP-11s (a minicomputer) connected through a circuit-switched crossbar network to 16 memory modules, forming a shared-memory MIMD design.

A prototype distributed memory MIMD machines was the Cosmic Cube developed at the California Institute of Technology in the early 80s. This contained 64 processing nodes each with a direct point-to-point connection to six other nodes forming a six dimensional hypercube.

The first commercial hypercube was the iPSC/1 (Intel Personal Supercomputer) which comprised between 32 and 128 nodes. Each node consisted of an Intel 80286/7 processor/coprocessor, 512 KBytes of memory, and a 10Mbit/second communication link. The peak performance of a 32-node model is about 2MFLOPS. Intel went on to develop a series of iPSC computers based on the 8086 and i860 series of microprocessors.

Another commercial hypercube is the nCube/10 produced in 1985. This consists of up to 1024 32-bit single-chip custom processors. Each node consists of this chip plus six 256-Kbit memory chips.

A key development in the 80s was the arrival of the INMOS transputer. The transputer is a microprocessor with special on-chip serial links for communicating with other transputers. This allows many transputers to be connected together to form a MIMD machine. Transputers are relatively inexpensive which allows even individuals access to parallel computing.

1.7.4 GFLOP parallel machines

The late 80s and early 90s saw the emergence of parallel systems capable of Giga FLOP peak performance. Intel produced the Touchstone Delta (a prototype for the Paragon) in 1991. This contained 528 i860 processors arranged in a mesh pattern and was capable of 10GFlops.

Parallel systems based on the fast RISC processors used in high performance workstations began to emerge in the 90s. Thinking Machines produced the CM-5 in 1992 which contained up to 1024 Sparc microprocessors connected in what is known as a fat tree topology. This machine was capable of a peak performance of 40GFLOPS. Meiko also use Sparc microprocessors in their machines.

More recently multiprocessor machines have emerged on based on the DEC Alpha processor. Cray have produced the T3D which contains up to 256 DEC Alpha chips arranged as a 3-D torus (a 3-D mesh with wraparound wires in the rows and columns)[15]. The 32 processor version has peak performance of 4GFLOPS and costs ~$2 million.

Convex produce the Exempler SPP1000/XA system which is a massively parallel processor using Hewlett - Packard's PA-RISC 7100 processors. The SPP1000/XA can have up to 128 processors giving a peak performance of 25GFLOPS. The system also claims to provide scalability to TFLOPS (T=tera=10^{12})of performance and TBytes of storage.

1.8 Conclusions

The key points in hardware and software development on parallel machines have been described. This has shown that a wide variety of architectures exist for parallel systems and the most suitable architecture is dependent on the algorithm being implemented. It has also been shown that the development of parallel software is a complicated matter which lacks standardisation.

The technologies of the future such as virtual reality and video conferencing will require a large amount of computational power to achieve the predicted performance and this will surely involve parallel computing. The computations involved in the so called Grand Challenges of science are also still not fast enough even on the most powerful supercomputers. If parallel computing is to provide the computational power required in the future more research has to be done to provide efficient parallel systems.

Most systems are basically extensions of the Von Neumann model of computation used on sequential processors. It would be helpful to develop a model (or models) of computation specific to parallel systems. This would hopefully lead to more standardisation in parallel systems.

More research is also required into interconnection networks. The study of the suitability of networks to particular problems is necessary to produce acceptable gains on parallel systems. This can be achieved by modelling parallel systems in order to study their features and predict their performance.

In the area of parallel software, techniques and tools have to be developed for mapping algorithms onto nodes. At the moment, mapping is usually left to the programmer and usually a heuristic approach is used. The development of new parallel languages designed specifically to handle the problems associated with parallel processing (i.e

communication protocols, parallel I/O etc.) would help to produce more efficient parallel code.

Methods of interprocessor communication both in hardware and software require standards, to enable applications to be portable. Research into producing message passing standards is underway with projects such as the MPI (message passing interface) forum[16]. The aim of the forum is to discuss and define a set of library interface standards for message passing.

This thesis is concerned with the design and implementation of both novel hardware and software for use on distributed multiprocessor machines. The hardware involves the design of two forms of dynamic interconnection network. The first method allows the topology of the network to be altered prior to computation and the second method permits the network topology to adapt as required during the computation. This work is covered in Part 1 of the thesis.

The software development is concerned with the parallelisation of a sequential FORTRAN molecular mechanics program to run on novel hardware, where each node processor has a dedicated high speed link to the host processor. This allows the host processor to broadcast code/data to all the nodes simultaneously. The parallelisation of the sequential code, involved the implementation of the COMFORT message passing subroutine library. This work is described in Part 2 of the thesis.

References

[1] Moldovan, Dan I. *"Parallel Processing: From Applications to Systems."* Morgan Kaufmann Publishers, San Mateo, California, 1993 (ISBN 1 55860 254 2)

[2] Almasi, George S. and Gottlieb, Allan. *"Highly Parallel Computing."* The Benjamin/Cummings Publishing Company, Redwood City, California, 1989 (ISBN 0 8053 0177 1)

[3] Flynn, M.J. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers*, C-21, **No.9**, Sept. 1972, pp. 948-960

[4] Feng, T.Y. Some Characteristics of Associative/Parallel Processing. *Proc. 1972 Sagamore Computing Conf.*, Aug. 1972, pp 5-16

[5] Händler, W. The Impact of Classification Schemes on Computer Architecture. *Proc. Int'l Conf. on Parallel Processing*, Aug. 1977, pp.277-300

[6] Skillicorn, David B. A Taxonomy for Computer Architectures. *Computer*, Nov. 1988, pp.46-57

[7] Parallel Programming Primer. *Intel® Corporation*, 1990

[8] Hazdra, T. and Singh B. Programming Transputers Major Issues. *A Workshop Presentation to the Transputer Research and Applications Conference*. Oct. 1994

[9] King, T. "Helios - A Distributed Operating System". *Technical Report No.2*. Perhelion Software Ltd. Dec. 1988

[10] *"The Helios Operating System"*. Perhelion Software Ltd. Prentice Hall International (UK) Ltd., 1989 (ISBN 0 13 386004 3)

[11] Express User's Guide. *Parasoft Corporation*, 1990

[12] Express Reference Manual. *Parasoft Corporation*, 1990

[13] World Wide Web. *http://www.netlib.org/pvm3/book/node17.html*

[14] Sharp, John A. *"An Introduction to Distributed and Parallel Processing"*. Blackwell Scientific Publications, Oxford, 1987 (ISBN 0 632 01462 8)

[15] *BYTE Magazine*, Feb. 1995, pp. 65-72

[16] Walker, D., Dongarra, J. (Convener & Meeting Chair). MPI: A Message-Passing Interface Standard. March 1993, University of Tennesse, Knoxville, Tenessee

Part 1

Chapter 2

Concepts in Digital Electronics

Part 1 of this thesis covers the design of various hardware systems. This chapter describes some of techniques used in the designs[1,2]. First of all the basics of digital electronics such as logic levels and gates are described and then a detailed description of the operation of programmable logic devices and programmable read only memory is presented.

2.1 Basic Digital Electronics

2.1.1 Logic Levels

Whereas analogue electronics involves quantities with continuous values, digital electronics involves quantities with discrete values. In digital electronics there are two different voltage levels: a logic high and a logic low. These two values can be represented by the binary digits 1 and 0 (a binary digit is a bit). In positive logic system (which is used in most cases) a 0 is logic low and a 1 is logic high and in a negative logic system the opposite is true. A group of several bits represents a piece of binary information such as a number or a letter (8 bits = byte, 16 bits = word).

In a digital circuit a logic high is a voltage between a specified minimum value and specified maximum. Likewise, a logic low can be any voltage between a specified minimum value and a specified maximum value (See Figure 2.1). For the purposes of this thesis a logic high will be taken to be +5V and a logic low will be taken to be 0V.

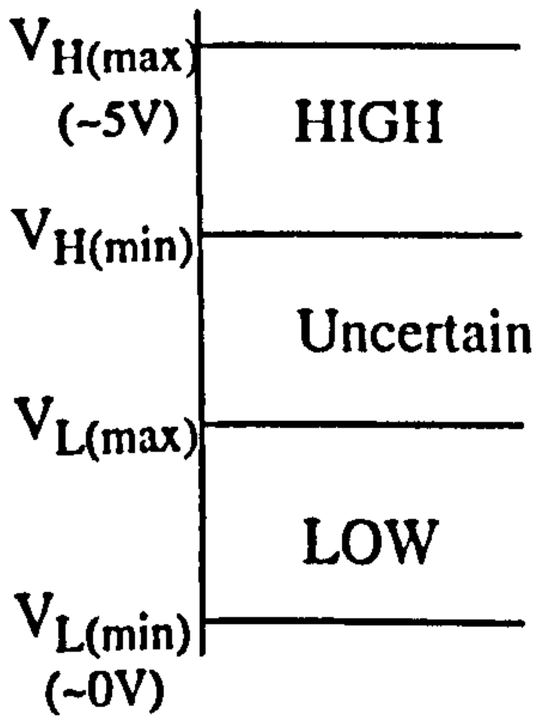
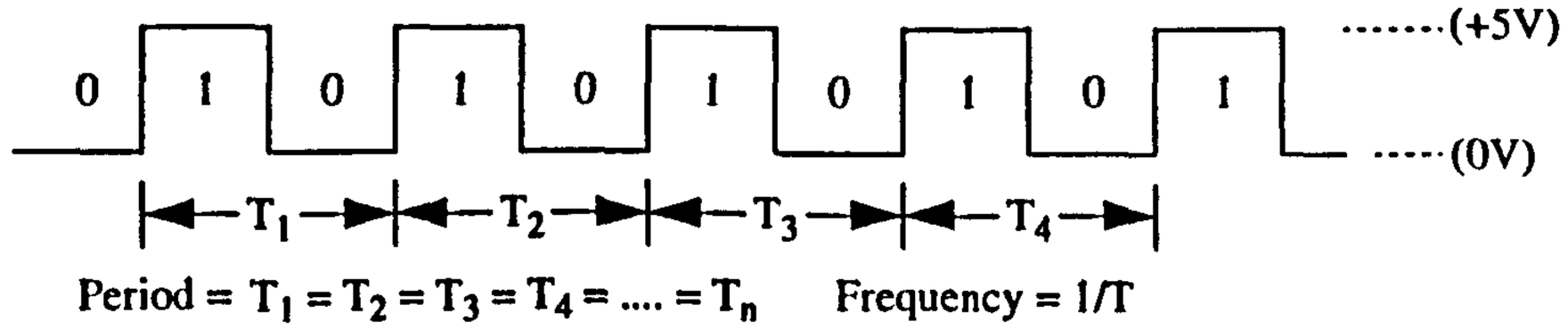
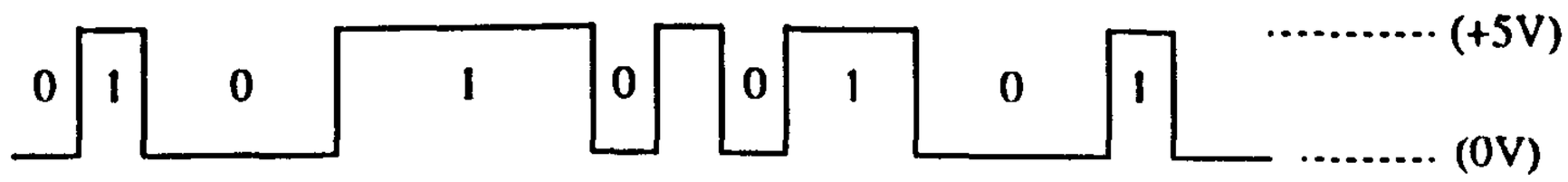


FIGURE 2.1. Logic level ranges for a digital circuit

Most binary information handled by digital systems appears as a pulse waveform (See Figure 2.2). All pulse waveforms are derived from and related to a basic timing waveform called the clock (See Figure 2.3). The clock is a periodic waveform in which each pulse interval (period) is one bit time. Figure 2.3 shows that each change in level of waveform A corresponds to a leading edge on the clock waveform. In some cases changes can occur on the trailing edges of the clock.



(a) Periodic (square wave)



(b) Nonperiodic

FIGURE 2.2. Examples of Pulse Waveforms

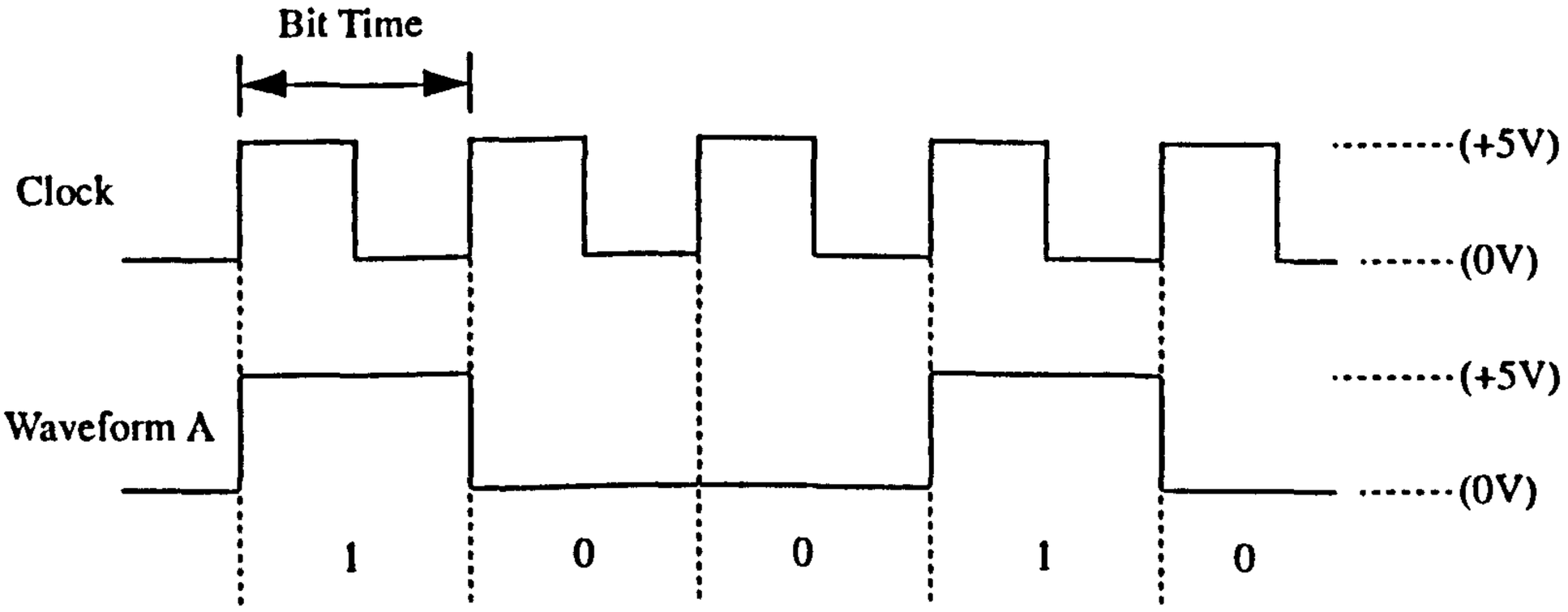


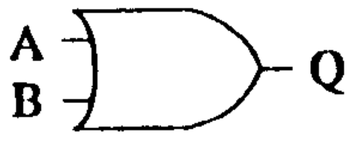
FIGURE 2.3. A timing diagram

2.1.2 Logic Gates

Complex digital systems such as microcomputers require to combine digital inputs to produce digital outputs. For example a FPU requires circuits that can add, divide and multiply numbers together. The basic elements (logic gates), and their truth tables, used

in combinatorial logic are shown in Figure 2.4. These gates are constructed from transistors. A small circle at an input or output on a gate indicates the signal is negated. The gates only have the capacity to combine inputs to produce an output and cannot memorise logic levels after the input conditions have been removed.

OR Gate

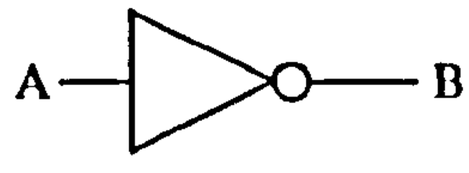


Inputs		Outputs
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

AND Gate

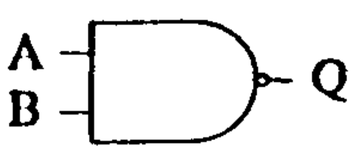
Inputs		Outputs
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

Inverter (the NOT function)



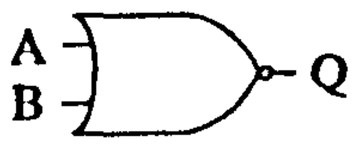
A	Q
0	1
1	0

NAND Gate



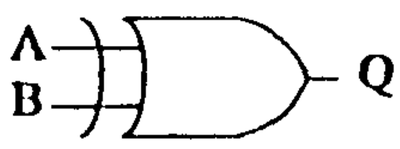
Inputs		Outputs
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate



Inputs		Outputs
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate



Inputs		Outputs
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

FIGURE 2.4. Basic logic gates used in digital design

Circuits which contain memory are known as sequential circuits. The fundamental element of memory used in digital circuits is called the flip-flop (see Figure 2.5). This is the basic type of flip-flop and it is constructed by combining two OR gates with negative inputs.

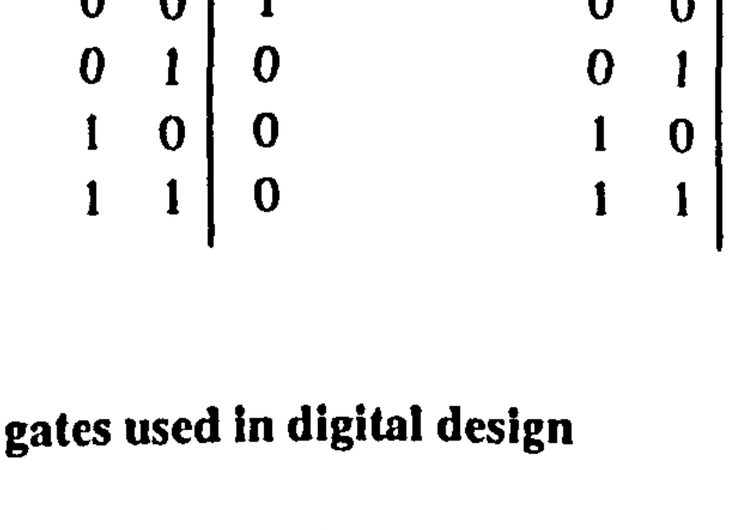


FIGURE 2.5. Flip-flop (set-reset)

The two stable states of the flip-flop with both inputs (A and B) logic high are shown in Figure 2.6 (it is not possible to have both outputs in the same logic state). If the input A is pulled low momentarily in both stable states the flip-flop is guaranteed to go into the state X = HIGH, Y=LOW. When the input A is returned to logic high the flip-flop remains in this state so the outputs are dependent on the previous state of the inputs and therefore the flip-flop has memory.

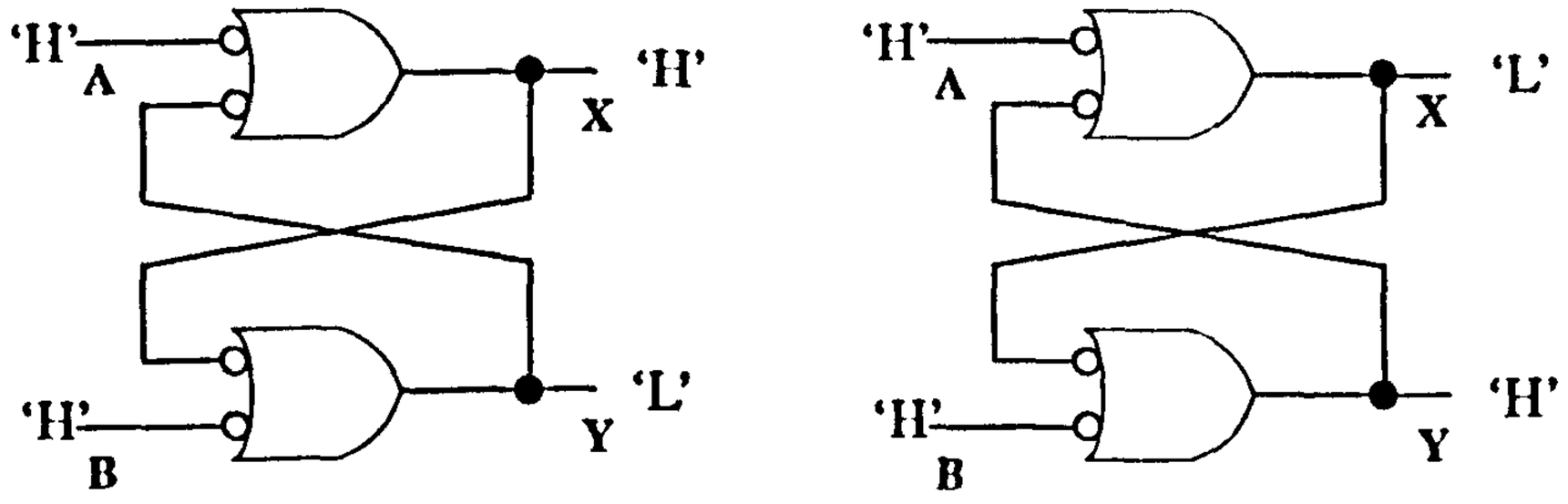


FIGURE 2.6. Stable states of flip-flop

Flip-flops that are made from two gates are generally known as SR (set-reset), or jam-loaded, flip-flops. They are forced into one state or the other by generating the correct input signal. The most widely used form of flip-flop however, looks slightly different. Instead of a pair of jam inputs, it has one or two data inputs and a single clock input. The outputs either change state or stay the same, depending on the levels at the data inputs when the clock pulse arrives.

The simplest form of clocked flip-flop is illustrated in Figure 2.7. It is basically the same as an SR flip-flop, with a pair of gates (controlled by the clock) to enable the SET and RESET inputs.

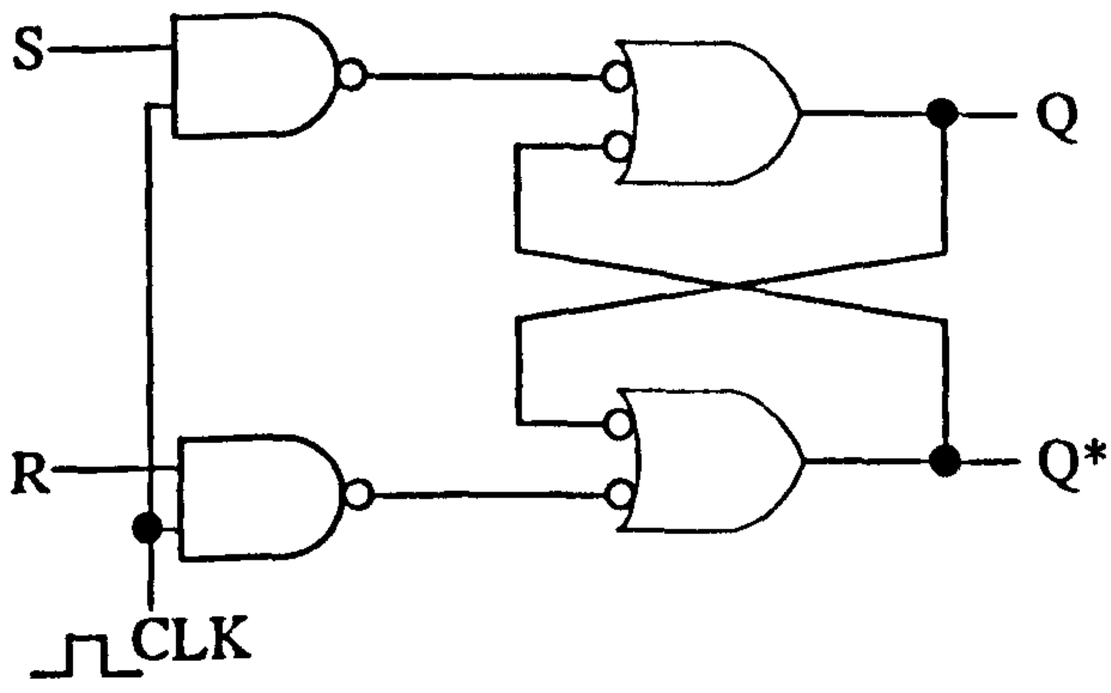


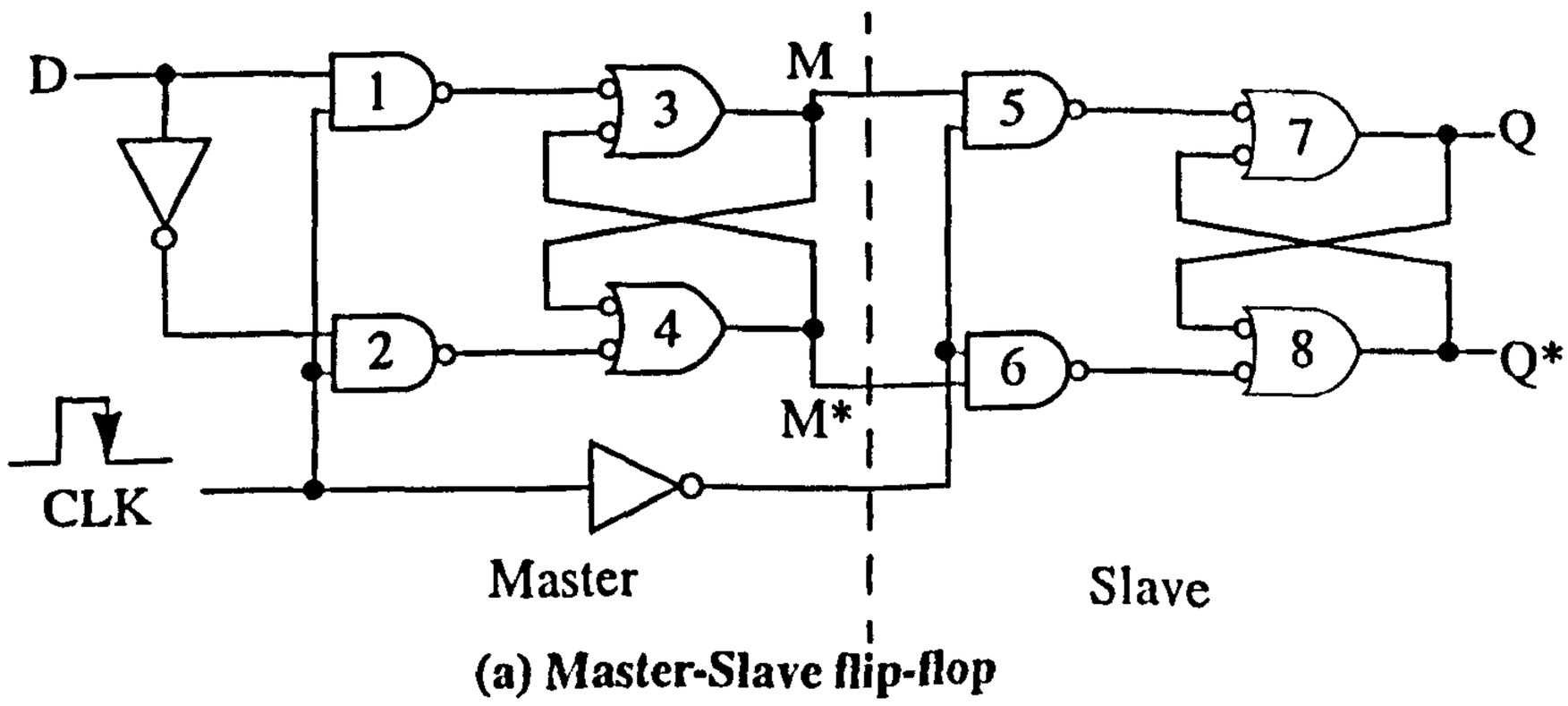
FIGURE 2.7. Clocked flip-flop

The truth table for this type of flip-flop is illustrated below:-

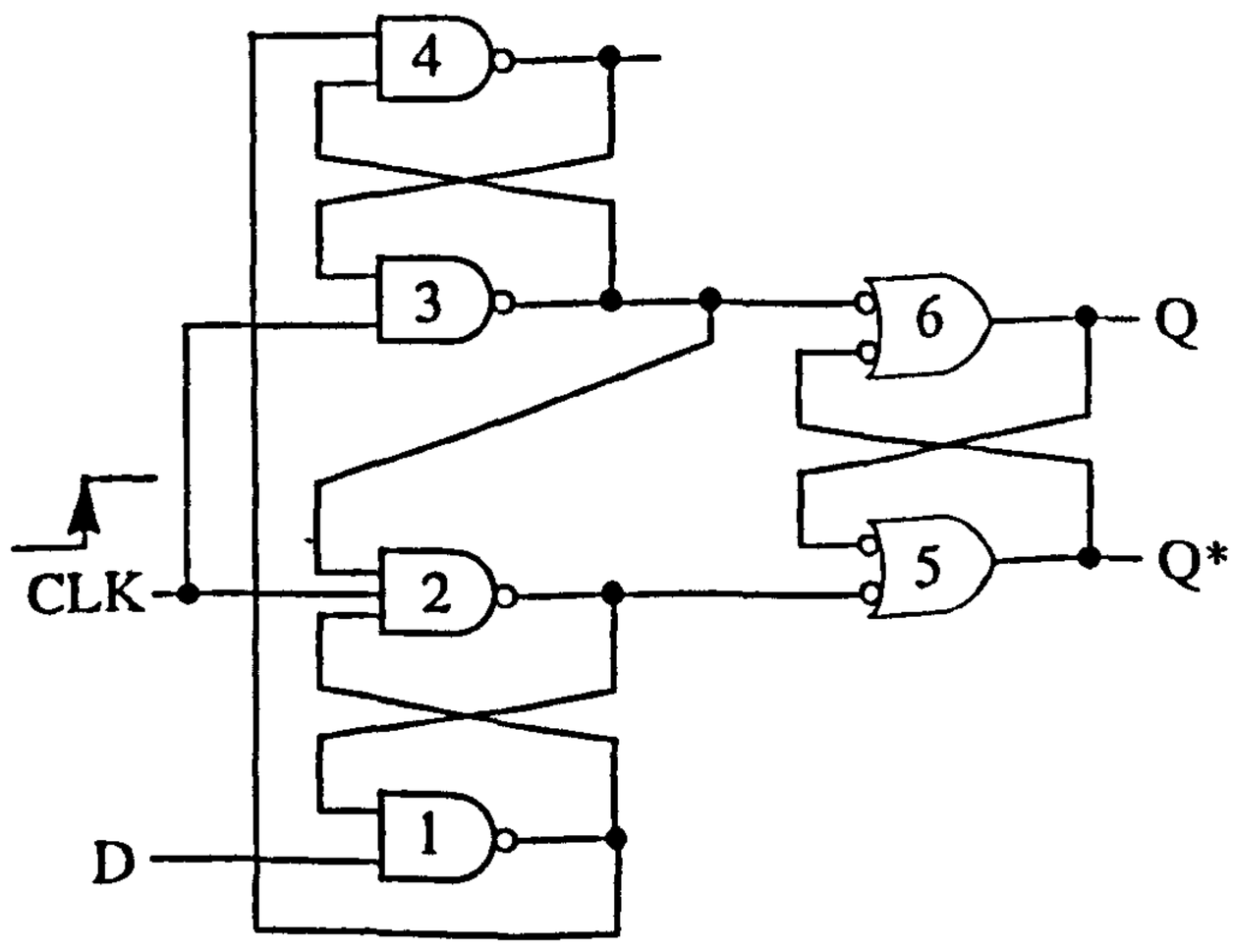
S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	indeterminate

where Q_{n+1} is the Q output after the clock pulse and Q_n is the output before the clock pulse. The basic difference between this and the previous type of flip-flop is that R and S can now be thought of as data inputs. What is present on R and S when a clock pulse arrives determines the logic level on Q.

A problem with this type of flip-flop however, is that the output can change in response to the inputs during the time the clock is logic high. This problem is solved with the use of the master-slave flip-flop and the edge-triggered flip-flop (See Figure 2.8).



(a) Master-Slave flip-flop



(b) Positive edge-triggered flip-flops

FIGURE 2.8. Master-slave and positive edge triggered flip-flops

These are the most popular type of flip-flop. The data present on the input lines just before a clock transition, or "edge" determines the output state after the clock has changed. They are both known as D-type flip-flops. Data present on the D input is transferred to the Q output after a clock pulse.

The master-slave flip-flop is basically two of the clocked SR flip-flops joined together. While the clock is logic high, gates 1 and 2 are enabled, forcing the master flip-flop (gates 3 and 4) into the same state as the D-input (i.e. M=D, M´=D´). Gates 5 and 6 are disabled, therefore the slave flip-flop (gates 7 and 8) retains its previous state.

When the clock returns to logic low, the inputs to the master are disconnected from the D input, while the inputs of the slave are simultaneously coupled to the outputs of the master. The master thus transfers its state to the slave and no further changes can occur at the output as the master is now stuck. At the next rising edge of the clock, the slave will be decoupled from the master and will retain its state, while the master will once again follow the input.

The edge-triggered circuit behaves the same externally as the master-slave circuit although the inner workings are different. In this case when the clock is low gates 2 and 3 are disabled and therefore the SR flip-flop (gates 5 and 6) retains its previous state. On the next rising edge of the clock gates 2 and 3 are enabled forcing the SR flip-flop into the same state as the D input (i.e Q=D, Q´=D´).

These type of flip-flops are known as D-type flip-flops. They are available with either positive or negative edge triggering (i.e. change state either on the rising or falling edge of clock). In addition, most flip-flops also have SET and CLEAR jam-type inputs. They may be set and cleared on HIGH or on LOW, depending on the type of flip-flop.

Figure 2.9 on page 43 shows a few popular flip-flops in IC form (explained later). The wedge means edge triggered and the small circle means "negation" or complement. The ´74 is a dual type D positive-edge-triggered flip-flop with active low jam-type SET and CLEAR inputs. The 4013 is a CMOS dual type D positive-edge-triggered flip-flop with active HIGH jam-type SET and CLEAR inputs.

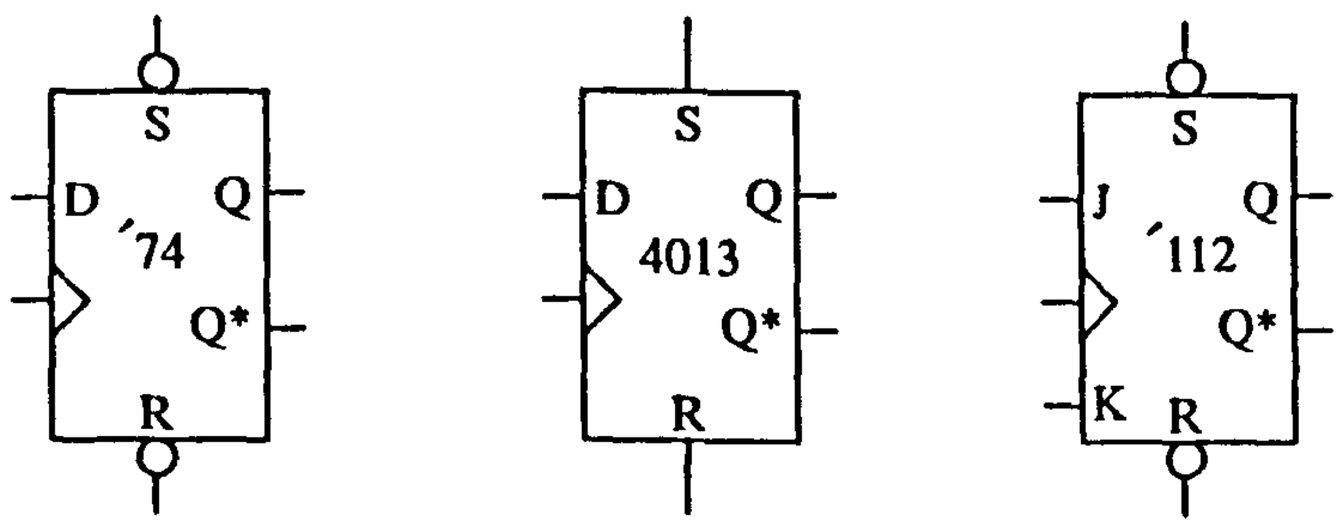


FIGURE 2.9. D-type and JK flip-flops

The JK flip-flop works on principles similar to those of the D-type flip-flop, but it has two data inputs. Figure 2.10 shows the truth table for a JK type flip-flop. If J and K are complements, Q will go to the value of the J input at the next clock edge. If J and K are both LOW, the output will not change. If J and K are both HIGH, the output will "toggle" (reverse its state after each clock pulse).

J	K	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	Q_n'

FIGURE 2.10. Truth table for JK type flip-flop

Logic gates and flip-flops are combined to construct more complex logic circuits, such as counters, registers, decoders, multiplexers and memories. These circuits are available in small packages called integrated circuits (ICs) made from silicon. The two most widely used type of IC are TTL (transistor-transistor logic) and CMOS (complementary metal oxide semiconductor). The difference between the two is in the types of transistor used in their construction; TTL uses bipolar transistors whereas CMOS uses field effect transistors.

Although an AND gate, for instance, performs identical operations in both TTL and CMOS versions, the logic levels and other characteristics (speed, power, input current, etc.) are quite different. Within any one logic family, outputs are designed to drive other inputs easily so the designer does not often have to worry about thresholds, input current etc. However when interfacing between logic families care has to be taken to ensure the correct operation of the circuit.

2.1.2.1 Buses and tri-state logic

In a computer system several functional units have to exchange data. The CPU, memory, and various peripherals all need to be able to send and receive 16-bit or 8-bit words. It would be awkward to have separate 16 or 8-wire cables connecting each device to all others. The solution is the so-called data bus, a single set of 16 or 8-wire cables connecting each device to all others. Only one device at a time may assert data but all may receive data at the same time (See Figure 2.11).

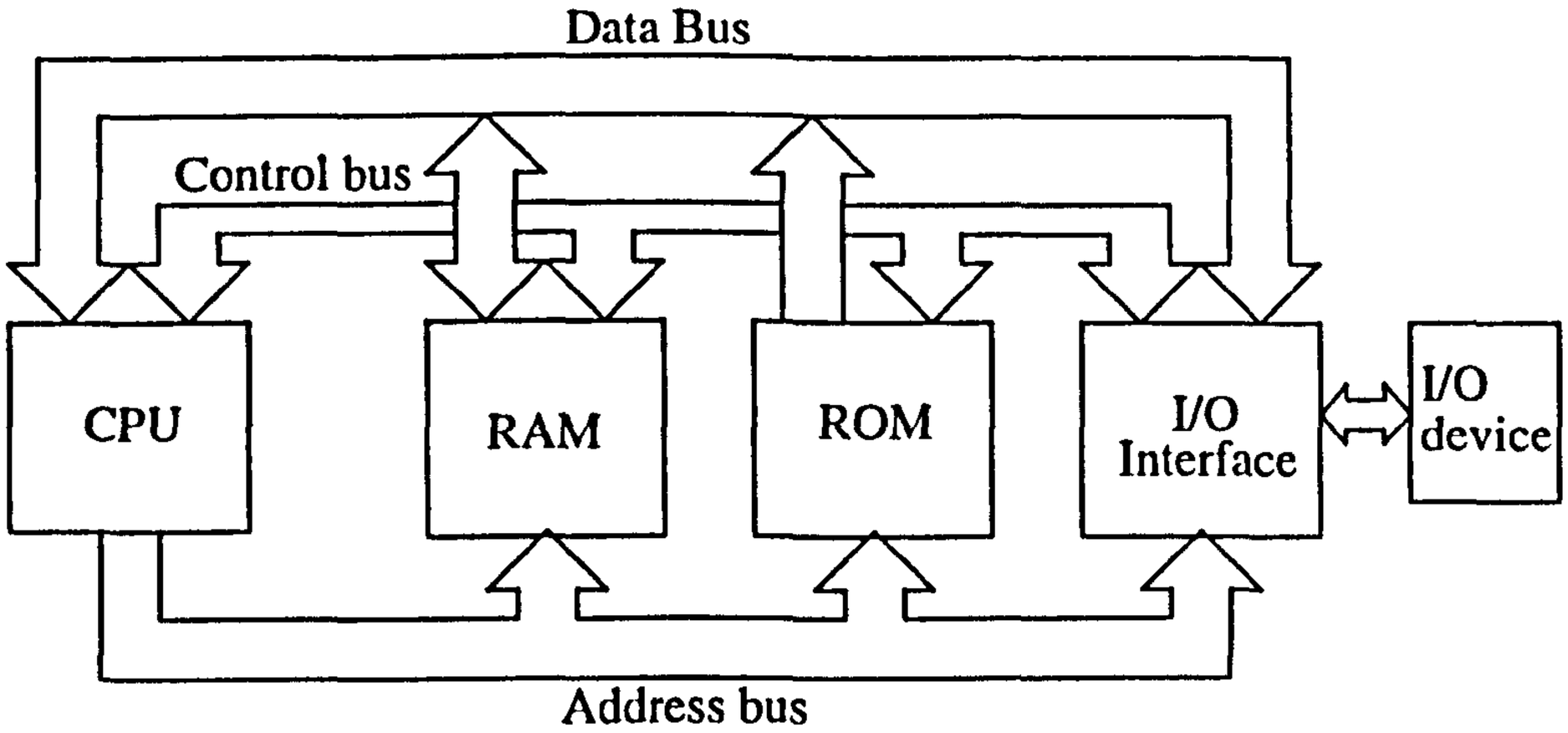


FIGURE 2.11. Basic bus structure in a microcomputer

As well as a data bus there are also address and control buses. Each device external to the CPU has an address or range of addresses corresponding to it. It can only send or receive data when it is addressed correctly. The control bus is for control signals such as **read** or **write** which specify whether the CPU is sending or receiving data.

There needs to be some way of isolating outputs from a shared data or address bus. This is achieved by what is called tri-state logic levels. The name is misleading; it is not digital logic with three voltage logic levels. It is just ordinary logic, with a third output state: open circuit (See Figure 2.12). A separate enable input determines whether the output behaves like an ordinary active pull-up output or goes into the "third" state (also known as the high impedance state), regardless of the logic levels present at the other inputs.

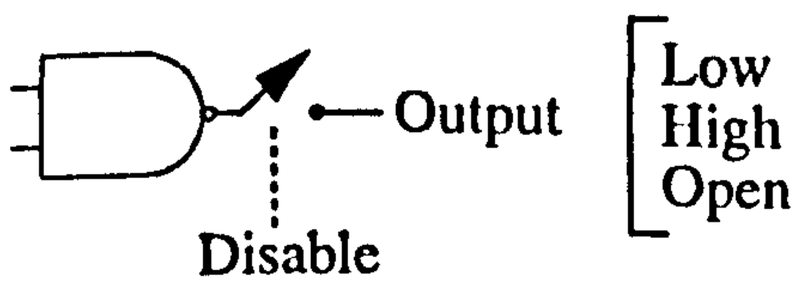


FIGURE 2.12. Conceptual diagram of a tri-state NAND gate

2.2 Rom and Programmable Logic Devices

Most ICs have a specific purpose (i.e. adder, comparator etc.) but in some the internal connections can be programmed for the required purpose. This is the case in PROMs (programmable read-only memory) and PLDs (programmable logic devices).

2.2.1 ROM

A ROM (read-only memory) holds a byte for each distinct address applied to its inputs. For example a 1K x 8 ROM gives eight output bits for each of 1024 input states, specified by a 10-bit input address (See Figure 2.13). A ROM can be programmed to produce a particular output from a particular input address. ROMs are often used to store finished programs and data tables.

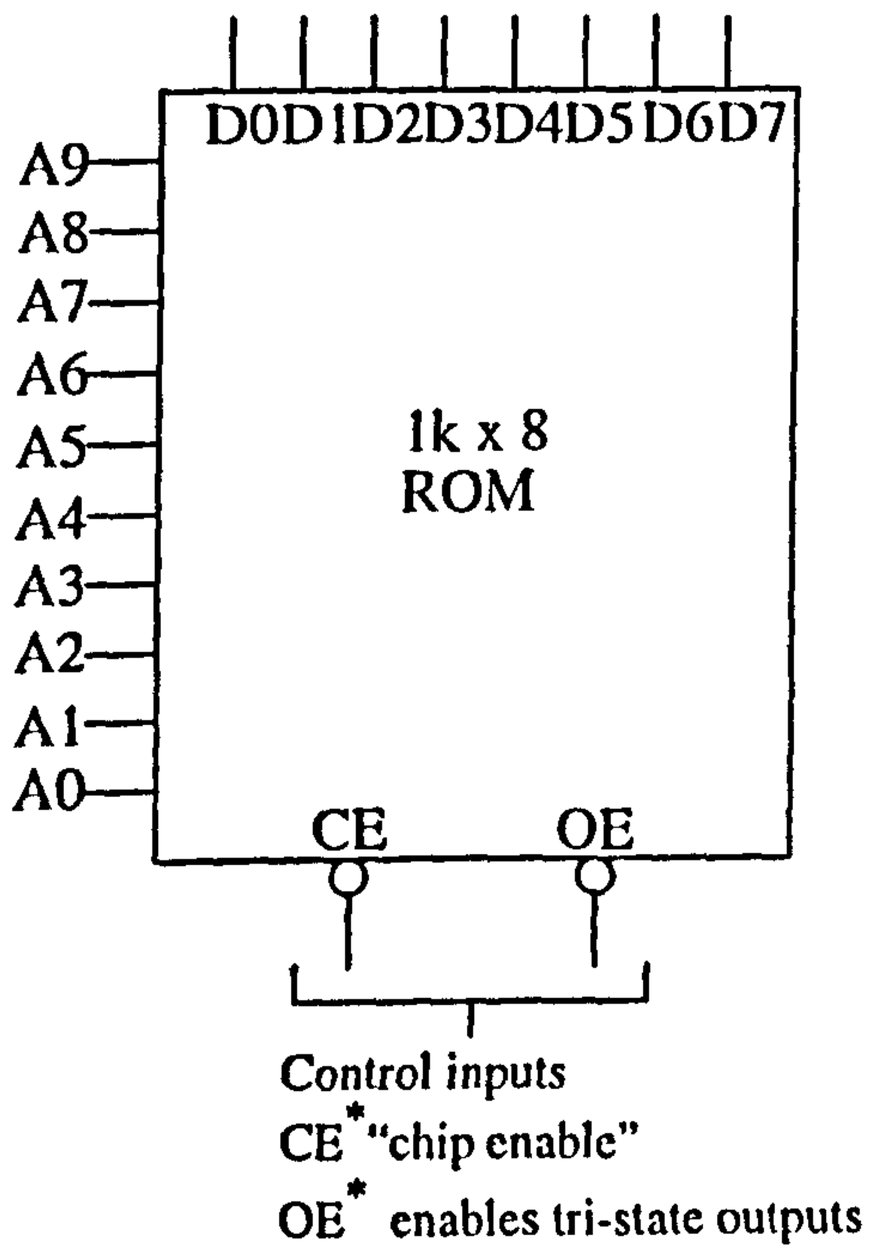


FIGURE 2.13. A 1K x 8K ROM

Most ROMs use the presence or absence of a transistor connection at a ROW/ COLUMN junction to represent a logic 1 or logic 0 (See Figure 2.14 on page 46). A connection from a ROW line to the base of a transistor represents a logic 1 at that location. When the ROW line is pulled HIGH (i.e. that row is addressed), all transistors with a base connection to that ROW line turn on and connect the HIGH to the

associated COLUMN lines. At ROW/COLUMN junctions where there are no base connections, the COLUMN lines remain LOW when the ROW is addressed.

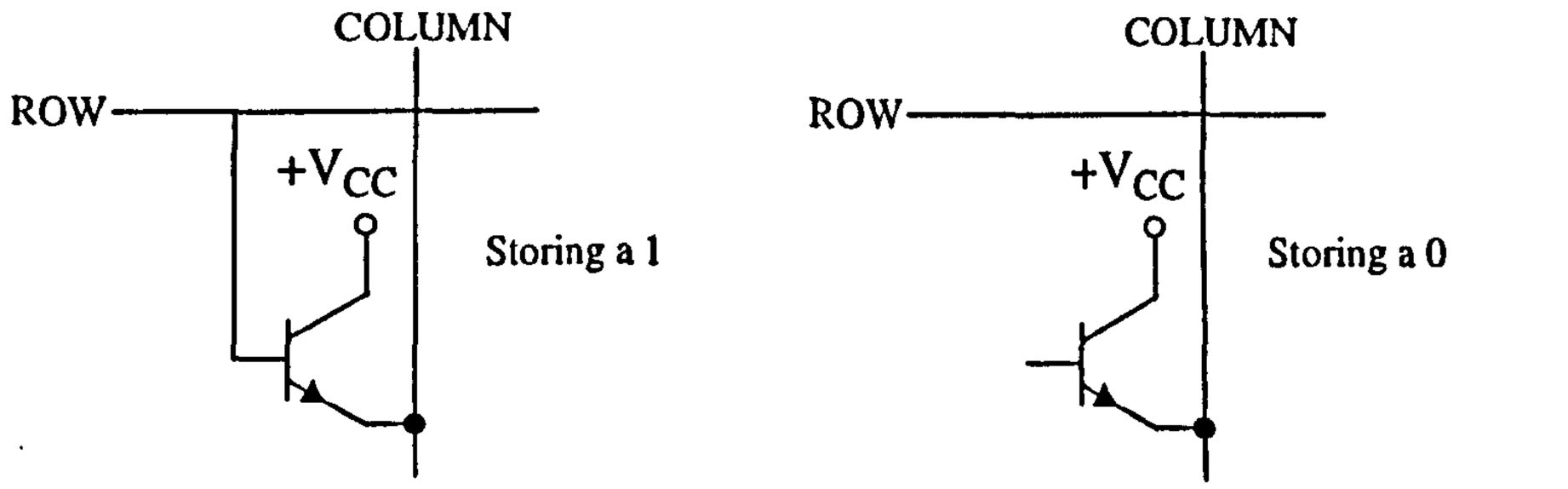


FIGURE 2.14. Bipolar ROM cells

ROMs are also available in CMOS technology using MOSFETs (metal oxide semiconductor field-effect transistors) rather than bipolar transistors as in TTL. The same principles apply however: in this case it is the presence or absence of a gate connection at a junction that permanently stores a logic 1 or 0.

Figure 2.15 shows a very simple ROM array. To read a byte of data from this ROM, first of all an address in applied to the address lines. The address decoder decodes the address and then sets the corresponding row to logic high. This high is connected to the column lines through the transistors at each junction (cell) where a 1 is stored. At the cells where a logic 0 is stored, the column line stays logic low due to the terminating resistor. Since the column lines form the data output, the eight data bits stored in the selected ROW appear on the output lines.

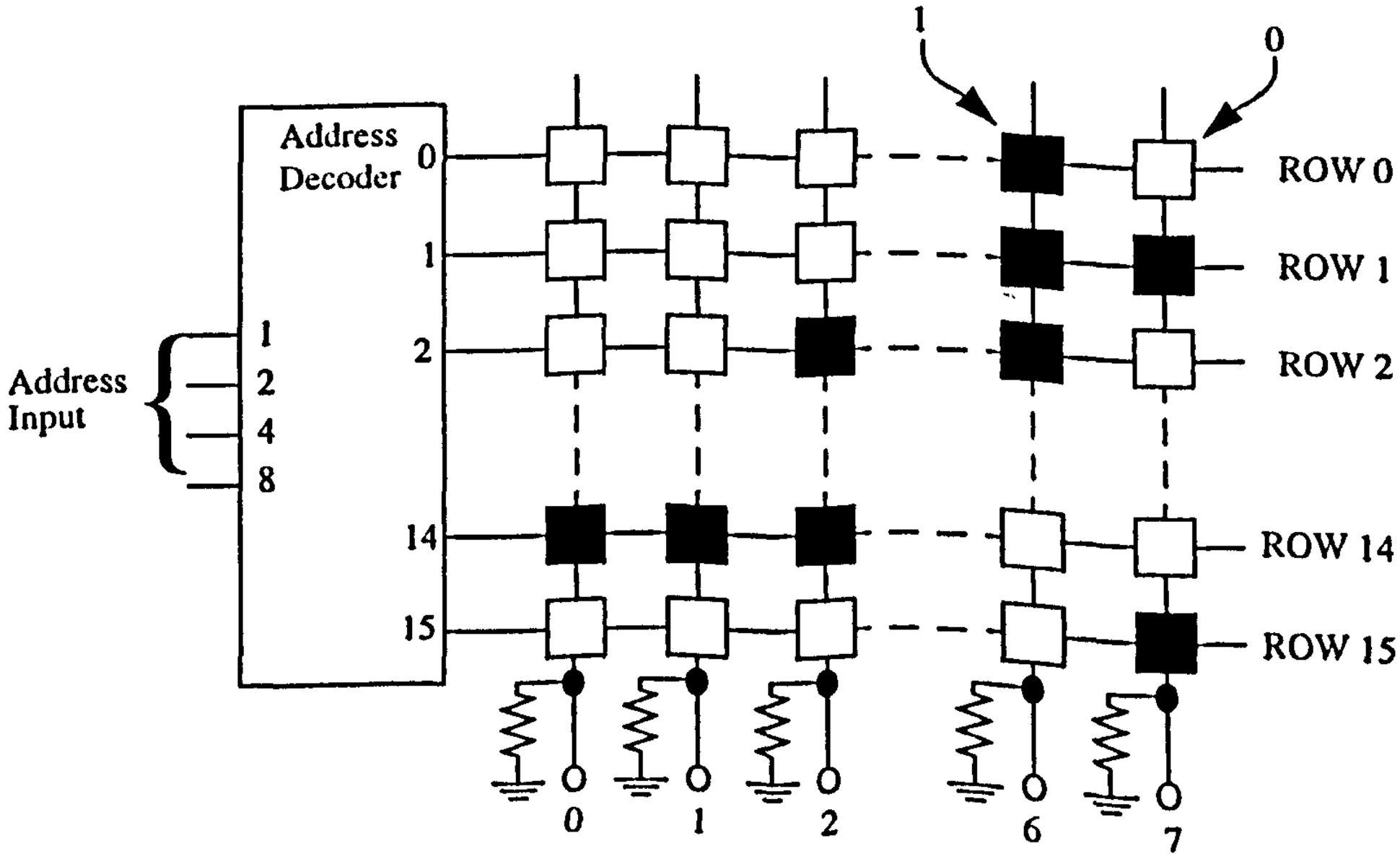


FIGURE 2.15. A 16x8-bit ROM array

This is an example of a very simple 16x8 ROM. In practice ROMs are usually more complicated but the same principles of rows and columns usually apply.

Mask-programmable ROMs have their bit pattern built in at the time of manufacture whereas programmable ROMs (PROMs) are programmable by the user. PROMs usually employ some type of fusing process to store bits, whereby a memory link is fused open or left intact to represent a 0 or 1. To program the connections an elevated voltage (usually 12.5V or 21V) is applied to the device while asserting the desired bytes at the appropriate addresses.

ROMs are nonvolatile, meaning that the stored information is retained even when power is removed. The information can however be erased in PROMs. Erasable programmable ROMs (EPROMs) can be erased by exposing them to intense ultraviolet light. Electrically erasable programmable ROMs (EEPROMs) behave like EPROMs, but can be programmed and erased electrically, while in the circuit, with the standard supply voltage (+5V). Internal circuitry generates the higher programming voltage required.

Both EPROMs and EEPROMs use an MOSFET array of transistors with an isolated-gate structure. The isolated gate has no electrical connections and can store an electrical charge for indefinite periods of time. The data bits in this type of array are represented by the presence or absence of a stored gate charge.

2.2.2 Programmable Logic

Programmable logic devices (PLDs) are similar to PROMs as they are fuse-programmable. However, they are different from PROMs in their applications. A PLD is used to implement combinatorial logic (some also have memory (registers)) and can replace individual gate or flip-flop ICs in many situations.

The most popular types of PLD are PALs (programmable array logic) and PLAs (programmable logic arrays). They both are single ICs which contain many gates whose interconnections can be programmed to form the desired logic functions. Obviously it is not possible to program any required logic function on a PLD, the functions available are limited by the gates inside the PLD.

Figures 2.16 and 2.17 show the basic designs of combinatorial (no registers) PALs and PLAs. To keep the figure simple, the AND and OR gates, though drawn with a single input line, are in fact multiple-input gates, with an input at every crossing (See Figure 2.18 on page 49)

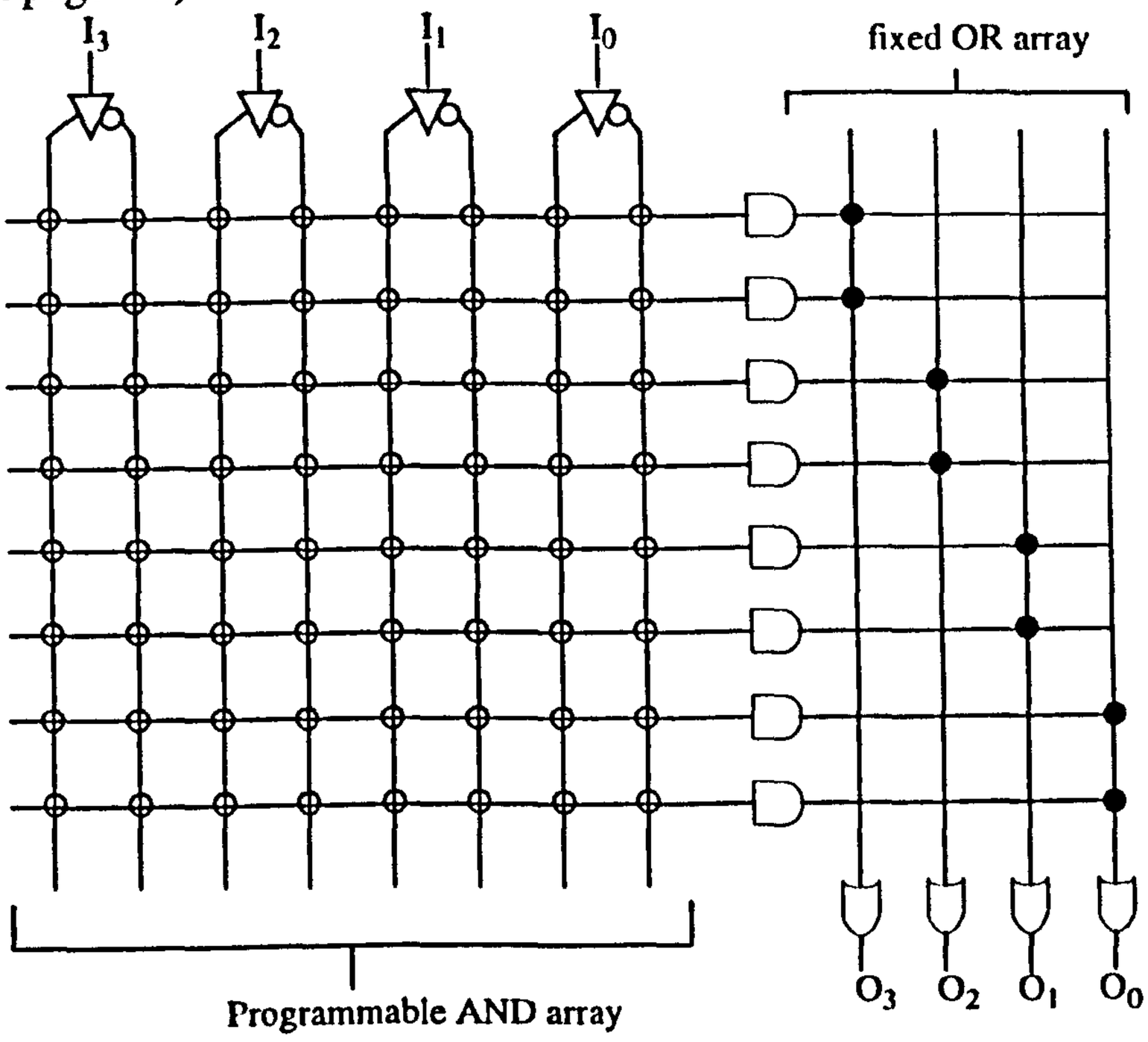


FIGURE 2.16. A PAL

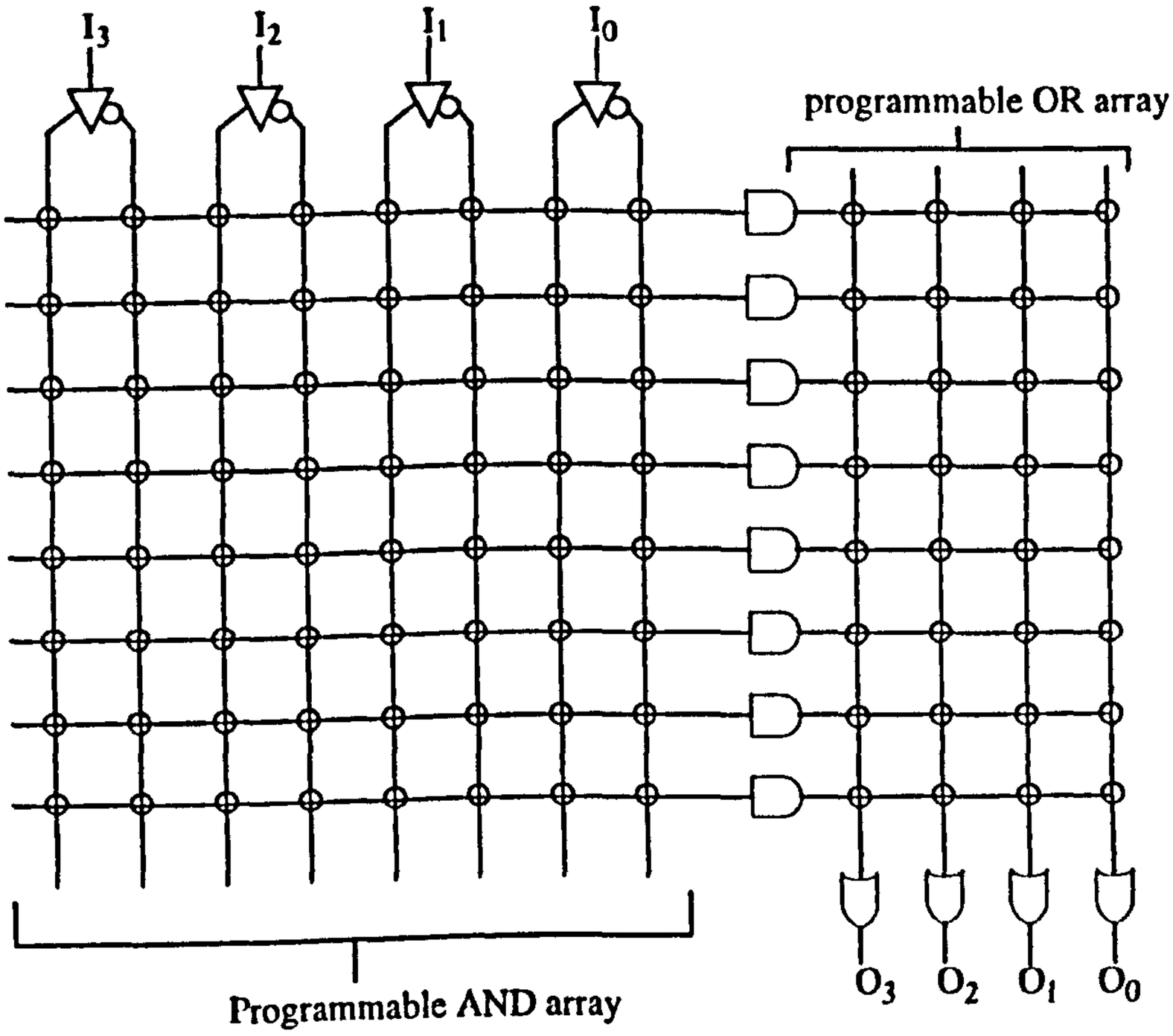


FIGURE 2.17. A PLA

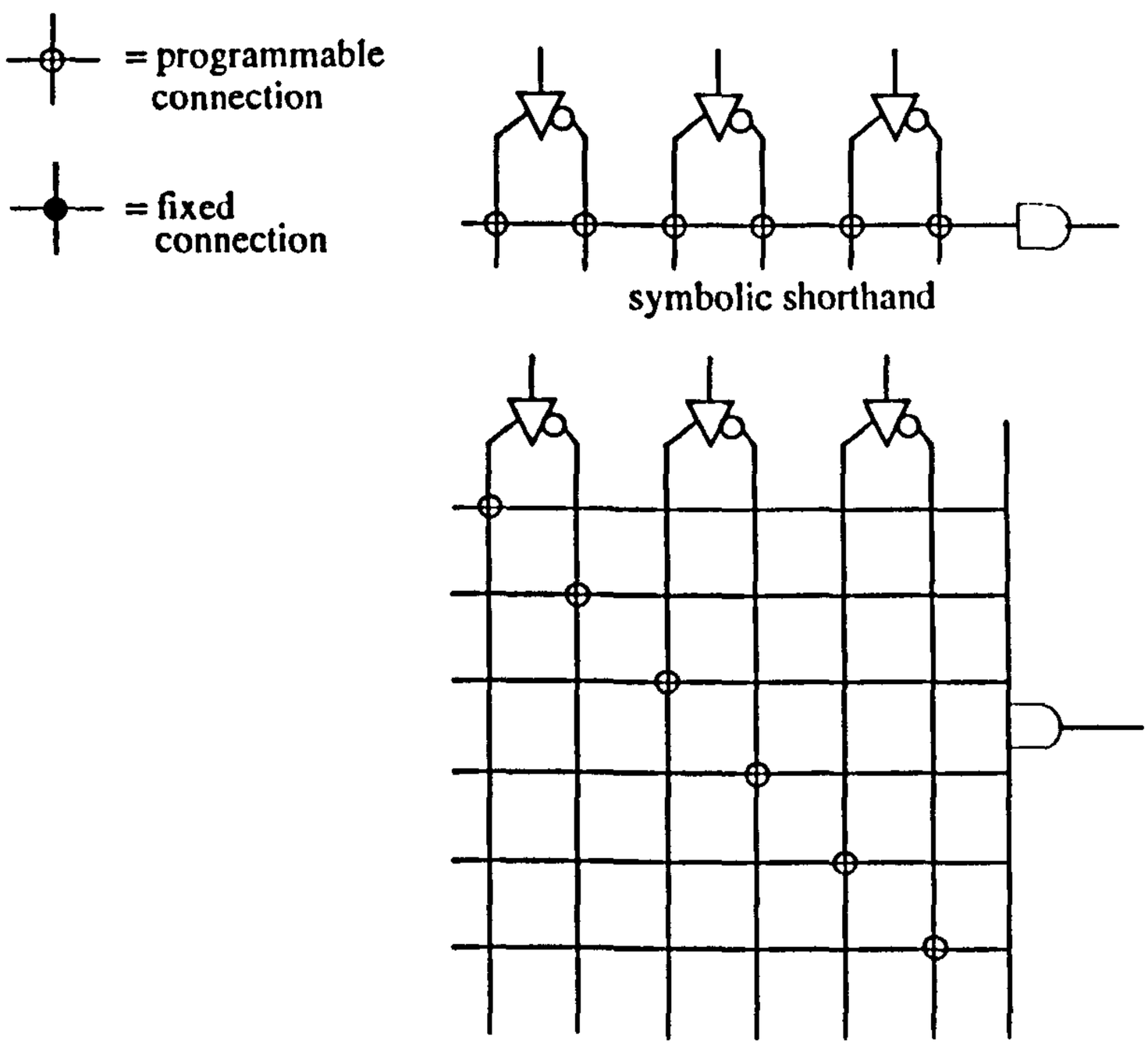


FIGURE 2.18. Details of shorthand used to describe PLDs

Each (tri-state) output of a combinational PAL comes from an OR gate, each of whose inputs is prewired to an AND gate with several inputs. PLAs are similar to PALs, but they have the added flexibility that the AND gate outputs can be connected to the OR-gate inputs in any combination (i.e the OR array is programmable), rather than being fixed as in a PAL.

The PALs and PLAs described previously are combinational (i.e. only contain gates) but they are also available with sequential logic (i.e. contain registers - a piece of memory composed of flip-flops). In general the outputs of the OR array in a PAL or PLA generate the inputs for clocked D-type registers (See Figure 2.19 on page 50) with tri-state outputs.

PLDs provide a flexible and compact alternative to fixed-function ICs. Sometimes designers are not quite certain how they want a circuit work, and PLDs allow the designer to experiment with different programming without the rewiring that would be

required with several ICs. Also PLDs can generally get the design job done more quickly once the designer had learned how to use them.

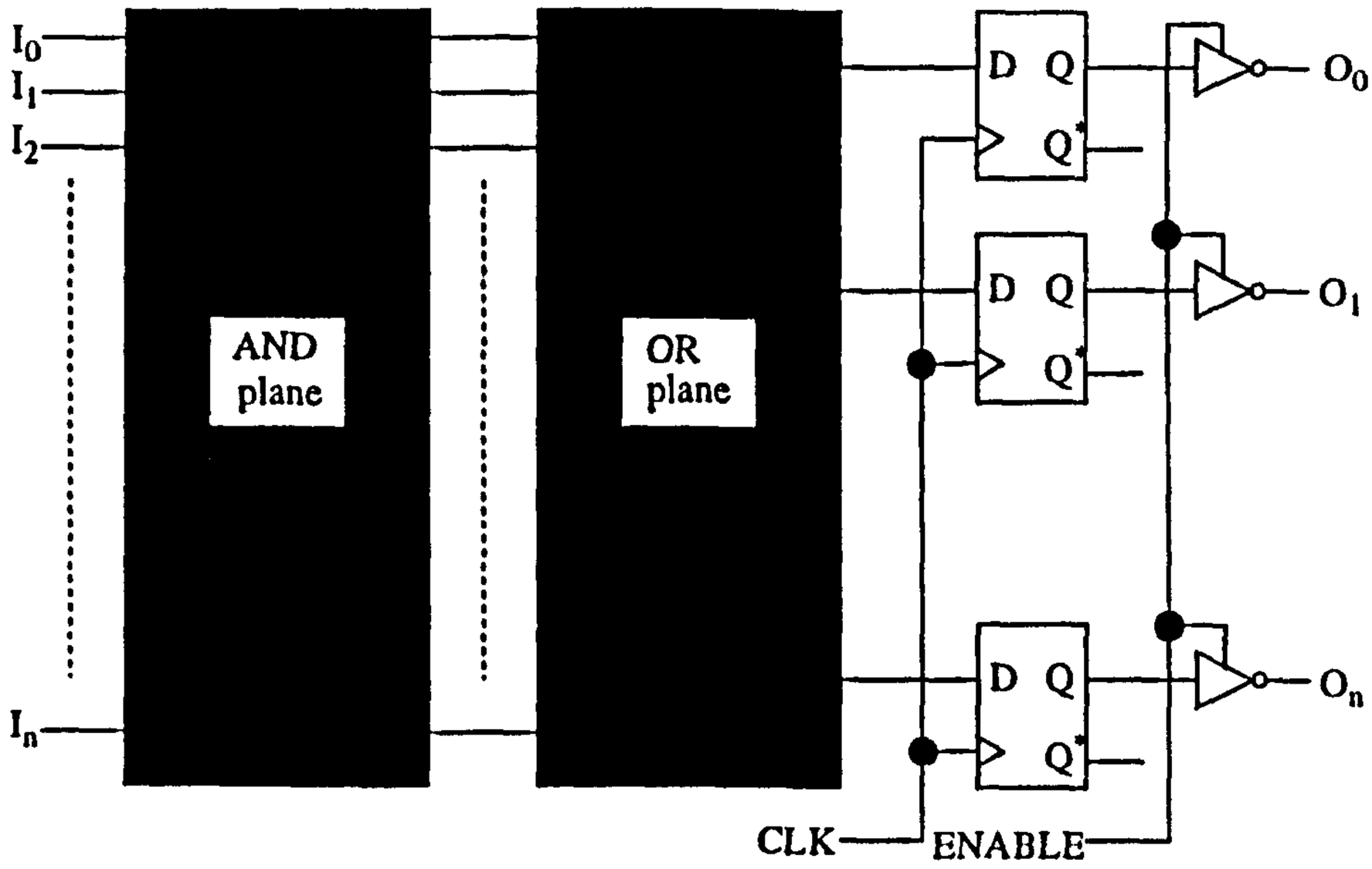


FIGURE 2.19. A PLD with registered outputs

2.2.3 Programming PLDs and PROMs

In order to program a PLD or PROM a device called a programmer is required which burns the fuses in the device and verifies the finished product. Most programmers connect via the serial port to a computer (usually a PC), on which some form of the programmer software runs.

The most basic kind of software simply lets you select the fuses to burn. The user decides what logic is required at the gate level, then lists (or marks on a graphics display) the fuses. Most programmers however let the user specify logic expressions and the software does the rest, including minimisation, simulation, and programming.

2.2.4 CUPL programming language

One of the programming languages used to program PLDs is CUPL[3]. It is a high level language which allows you to define arrays (for a set of signals, e.g. an address bus), expressions, and intermediate values, then use them in later expressions. It also produces a standard JEDEC download file which is compatible with any device programmer that uses JEDEC files.

2.2.4.1 CUPL source code

An example of a CUPL source file which shows how simple NOT, AND, OR, and XOR gates can be constructed using a PLD is shown in Figure 2.20 on page 52. The / * and */ constructs mark the beginning and end of comments. To mark the end of a statement a semi-colon is required.

The inputs to the PLD are **a** and **b** (pins 1 and 2 of the PLD) and the outputs of the PLD are **inva, invb, and, nand, or, nor, xor** and **xnor** (pins 12-19). A description of the logic operators used in CUPL is given in Table 2.1. From this it should be clear how the outputs in Figure 2.20 are constructed.

TABLE 2.1. Logical Operators

Operator	Example	Description
!	!A	NOT
&	A & B	AND
#	A # B	OR
$	A $ B	XOR

A variable preceded by a **!** has different meanings in pin assignments and logic expressions. In a pin assignment it identifies that an input or output is active low (i.e when it is low it is logic true (i.e. active)). A pin assignment variable not preceded by **!** identifies an active high input or output. In a logic expression when a variable is preceded by a **!** this inverts the signal (i.e. it makes the signal logic false (inactive) regardless of whether it is active high or low in the pin assignments).

Figure 2.21 on page 53 shows a more complicated example of CUPL source code **(waitgen.pld)**. In this scenario the PLD is acting as an interface between a CPU, ROM and RAM (See Figure 2.22 on page 54). The PLD performs address decoding and timing control functions. The PAL used is a 16R8 which has eight external inputs, eight outputs (four of which are registered (using D-type registers)), a clock, and a tri-state control line.

```
Name            Gates;
Partno          CA0001;
Revision        04;
Date            9/12/89;
Designer        G. Woolhiser;
Company         Logical Devices, Inc.;
Location        None;
Assembly        None;
Device          G16V8;
/*************************************************************/
/*                                                           */
/*      This is a example to demonstrate how CUPL            */
/*      compiles simple gates.                               */
/*                                                           */
/*************************************************************/
/*      Target Devices: P16L8, P16LD8, P16P8, EP300, and 82S153 */
/*************************************************************/

/* Inputs:  define inputs to build simple gates from*/

 Pin 1 =  a;
 Pin 2 =  b;

/*
 * Outputs:  define outputs as active HI levels
 *
 * Note: For PAL16L8 and PAL16LD8, DeMorgan's Theorem is applied to
 *  invert all outputs due to fixed inverting buffer in the device.
 */

Pin 12 = inva;
Pin 13 = invb;
Pin 14 = and;
Pin 15 = nand;
Pin 16 = or;
Pin 17 = nor;
Pin 18 = xor;
Pin 19 = xnor;

/* Logic:  examples of simple gates expressed in CUPL*/

inva = !a;                  /* inverters */
invb = !b;
and  = a & b;               /* and gate */
nand = !(a & b);            /* nand gate */
or   = a # b;               /* or gate */
nor  = !(a # b);            /* nor gate */
xor  = a $ b;               /* exclusive or gate */
xnor = !(a $ b);            /* exclusive nor gate */
```

FIGURE 2.20. CUPL source code for simple gates

```
Name       Waitgen;
Partno     P9000183;
Date       03/14/85;
Revision   02;
Designer   Osann;
Company    ATI;
Assembly   PC Memory;
Location   U106;
Device     F155;

/*************************************************************/
/* This device generates chip select signals for one         */
/* 8Kx8 ROM and two 2Kx8 static RAMs.  It also drives        */
/* the system READY line to  insert a wait-state of at       */
/* least one CPU clock for ROM accesses.                     */
/*************************************************************/
/** Allowable Target Device Types : PAL16R4, 82S155         **/
/*************************************************************/

/**  Inputs  **/

PIN 1          = cpu_clk    ;     /* CPU clock                */
PIN [2..6]     = [a15..11] ;      /* CPU Address Bus          */
PIN [7,8]      = ![memw,memr] ;   /* Memory Data Strobes      */
PIN 9          = reset      ;     /* System Reset             */
PIN 11         = !oe        ;     /* Output Enable            */

/**  Outputs  **/

PIN 19         = !rom_cs    ;     /* ROM Chip Select          */
PIN 18         = ready      ;     /* CPU ready signal         */
PIN 15         = wait1      ;     /* Start Wait State         */
PIN 14         = wait2      ;     /* End Wait State           */
PIN [13,12]    = ![ram_cs1..0] ;  /* RAM Chip Selects         */


/** Declarations and Intermediate Variable Definitions **/
Field memadr = [a15..11] ; /* Give The Address Bus         */
                           /* the Name "memadr"            */
memreq = memw # memr ; /* Create The Intermediate         */
                       /* Variable "memreq"               */
select_rom = memr & memadr:[0000..1FFF] ;   /* = rom_cs */

/**  Logic Equations  **/
rom_cs = select_rom ;
ram_cs0 = memreq & memadr:[2000..27FF] ;
ram_cs1 = memreq & memadr:[2800..2FFF] ;
wait1.d = select_rom /* = rom_cs */ & !reset ;/* Synchronous Reset
wait2.d = select_rom &  wait1 ;    /* wait1 delayed      */
ready.oe = select_rom; /*turn buffer on*/
ready = wait2; /* End Wait */
```

FIGURE 2.21. CUPL source code for interface between memory and CPU

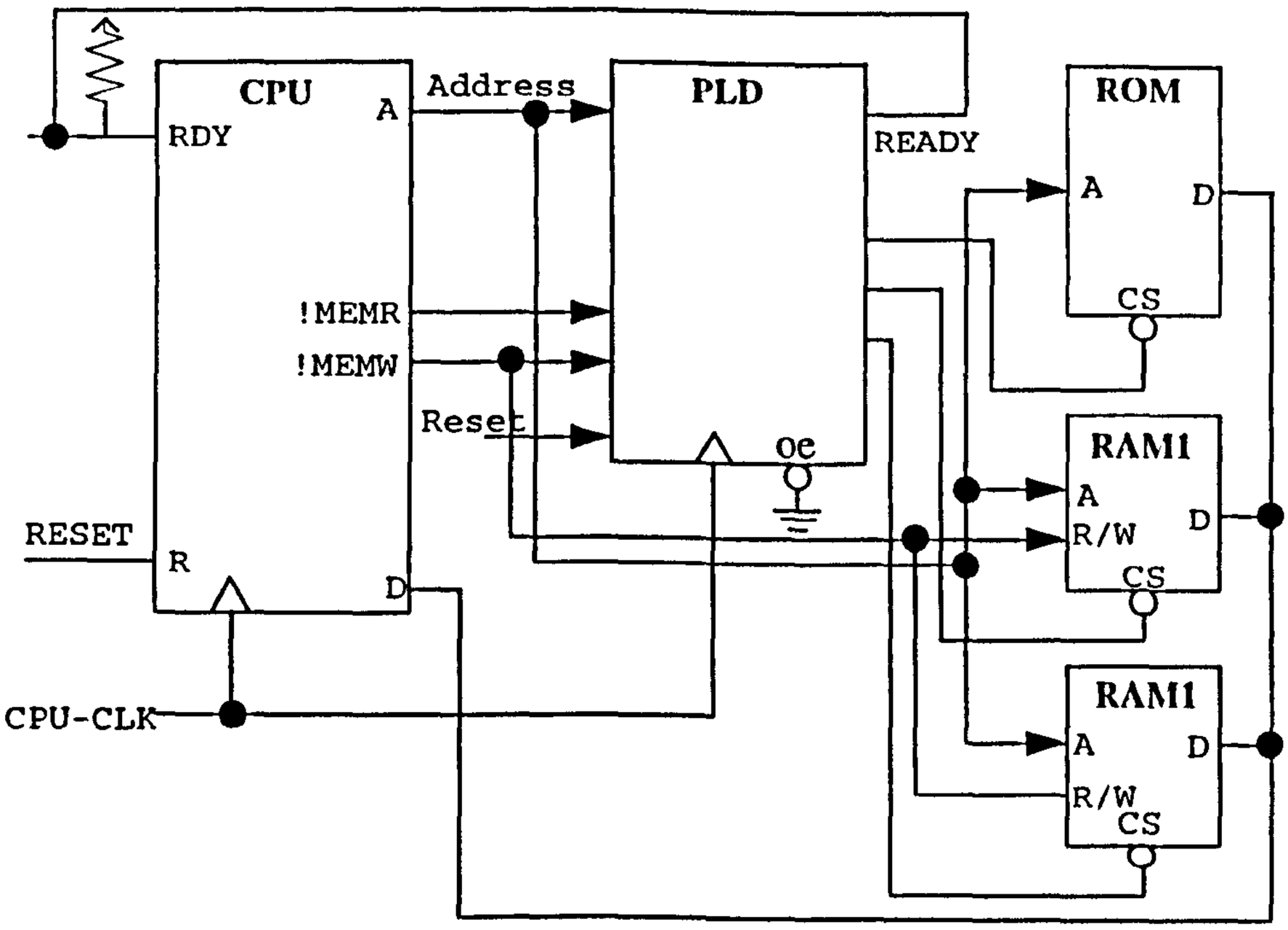


FIGURE 2.22. Microprocessor-based system

The !oe pin on the PLD is pulled to ground permanently which means the tri-state outputs of all four pins connected to registers are always enabled.

The functions of the inputs and outputs of the PLD are given in the comments following each pin assignment. In order to define the address bus an intermediate variable **memadr** is defined using the FIELD statement. This statement assigns a single variable name to a group of bits. When the variable name is used in an expression, the operation specified in the expression is applied to each bit in the group.

To check for specific values on the address bus the equality operation (See Figure 2.23 on page 55) is used. This checks for a bit-wise equality between a set of variables and a constant. If both quantities are equal then the result is set logic true otherwise it is set logic false.

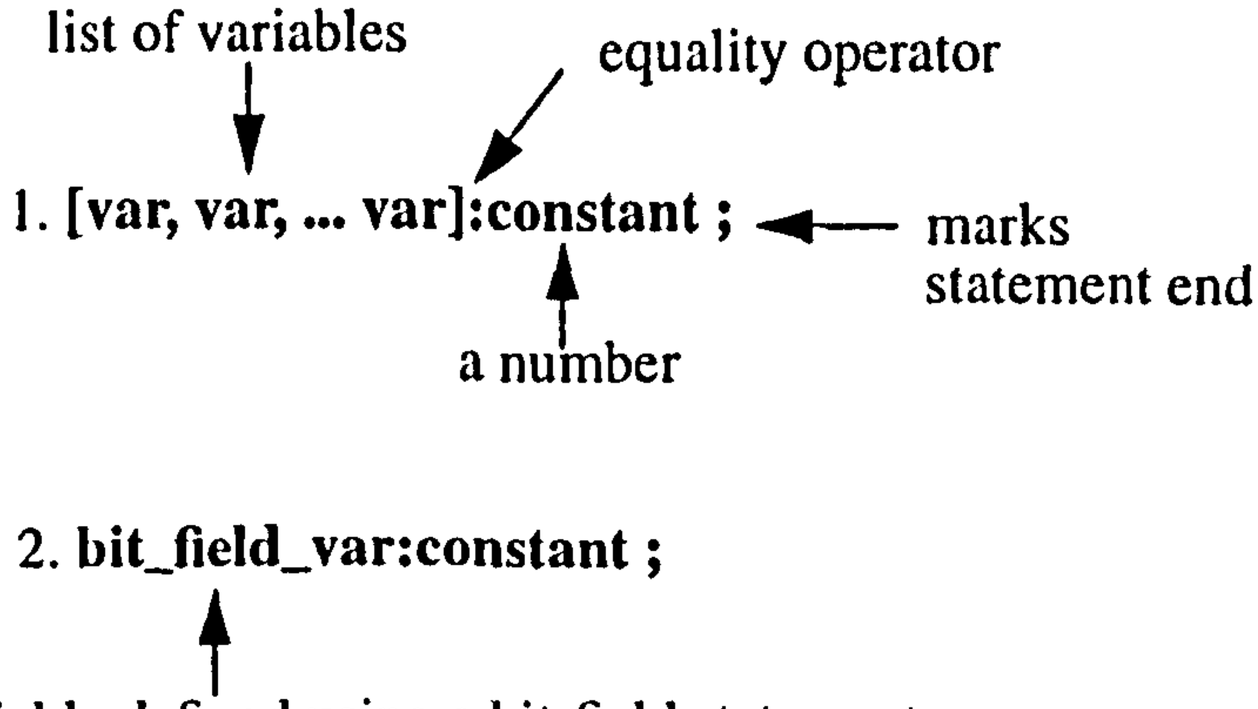


FIGURE 2.23. The equality operator

Since both RAMs require to be selected when read from and written to, an intermediate variable **memreq** is declared which is logic true when either the **memr** or **memw** signal is true. Whenever **memreq** is used in other equations, CUPL substitutes **memw # memr** at compile time. The RAM chip select signals therefore only become logic true when the **memreq** signal is true and the addresses are within the specified ranges for the RAMs.

Another intermediate variable **select_rom** is declared which is used as the chip select for the ROM and is also used in the generation of wait states. Since the ROM is only read from and not written to, the **select_rom** signal only becomes logic true when **memr** is logic true and the address is in the specified range.

The **wait** and **ready** signals are required as the ROM chip is slow; at least one CPU clock period is required to be added to the ROM access time (i.e. a wait state - holds address valid and read or write signal logic true longer than normal). The **RDY** input to the CPU is used to insert wait states. A timing diagram for the signals necessary to create the wait state is shown in Figure 2.24.

When the **!memr** signal becomes logic true (actually logic low as the signal is active low) for an address corresponding to the ROM, the **!rom_cs** signal is asserted (logic low). This also turns on the ready signal which is the output of a tri-state buffer (i.e. **ready.oe = select_rom**). The **ready** signal is logic low which indicates to the CPU to insert a wait state.

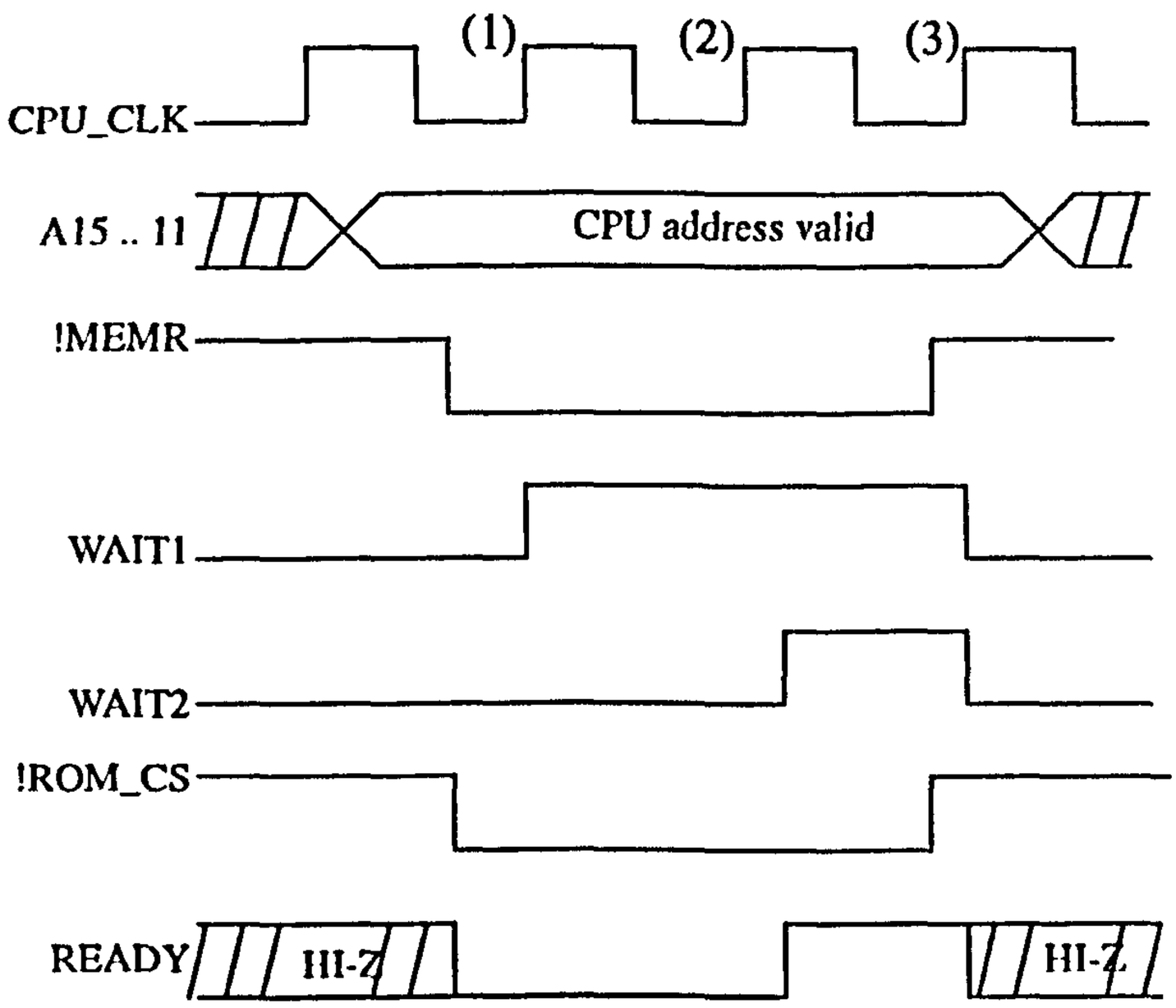


FIGURE 2.24. Wait state generator timing diagram

Since the **wait** signals are outputs of D-type flip-flops they only become set on a rising clock edge. The **wait1** signal therefore only becomes logic true on the rising edge of the CPU clock (1) (i.e. **wait1.d = select_rom**). After one CPU clock period has passed, the **wait2** signal is asserted and therefore at this point the wait state period of one clock cycle has been completed. This causes the **ready** signal to be pulled logic high which causes the CPU to continue its read cycle and remove the **!memr** signal at the appropriate time.

The **!memr** returning to logic false causes the **!rom_cs** to become logic false which disables the tri-state buffer driving the **ready** signal. At the next rising edge of the CPU clock (3) the **wait1** and **wait2** signals are pulled logic false ready for the CPU to assert another cycle.

This example illustrates the use of a PLD as an interface between a CPU and various components. The use of PLDs for this purpose is implemented throughout this thesis. PLDs are also in the design of state machines, however this will be described in Chapter 4.

2.2.4.2 CUPL simulator

CUPL also has a simulator (CSIM) to test logic expressions. Test vectors are specified for the inputs to the PLD and the expected output vectors from these test vectors are written into a file. Test vectors can also be downloaded to a device programmer in order that the actual PLD is tested.

The test specification source file (**waitgen.si**) for the previous example is shown in Figure 2.25 on page 58. This file contains three major parts; header information and title block, an ORDER statement, and a vectors statement.

This file has the same header information as the CUPL source code to ensure that the proper files, including current revision level, are being compared against each other.

The ORDER statement lists the input and output variables that are included in test vectors and also defines how they are displayed in the output file. The variables are listed in the order they are to be displayed. They are separated by a comma and the % symbol indicates the number of spaces between the variables.

Following the ORDER statement is the VECTORS statement that creates a function table containing eleven test vectors. To make the vectors easier to understand the $MSG command is used to create a heading for the function table. The variable names are listed in vertical columns in the same order and with the same spacing as specified in the ORDER statement.

The test vectors are entered underneath the appropriate variable names. These vectors are created by assigning a value to each of the input variables and an expected value to each of the output variables. Table 2.2. on page 59 shows the allowable values to use for the test vectors.

The $REPEAT directive in the test vectors causes the eighth vector to be repeated twice. The asterisks in the eighth vector for WAIT1, WAIT2, and READY tell CSIM to compute the output based on the inputs and place the results in the output file.

```
Name      Waitgen;
Partno    P9000183;
Date      03/14/85;
Revision  02;
Designer  Osann;
Company   ATI;
Assembly  PC Memory;
Location  U106;
Device    F155;
/*************************************************/
/* This device generates chip select signals for one    */
/* 8Kx8 ROM and two 2Kx8 static RAMs.  It also drives   */
/* the system READY line to  insert a wait-state of at  */
/* least one CPU clock for ROM accesses.                */
/*************************************************/
ORDER:
        cpu_clk , %2, a15, %2, a14, %2,
        a13, %2, a12, %2, a11, %2,
        !memw, %2, !memr, %2, reset, %2, !oe,
        %4, !ram_cs1, %2, !ram_cs0, %2, !rom_cs, %2,
        wait1, %2, wait2, %2, ready;
VECTORS:
   /* 123456-leave six blanks to allow for numbers in .SO file */
$msg "                                        !  !           ";
$msg "      c                                 r  r  !        ";
$msg "      p                                 a  a  r        ";
$msg "      u                    !  !  r      m  m  o  w  w  r";
$msg "      _                    m  m  e      _  _  m  a  a  e";
$msg "      c  a  a  a  a  a  e  e  s  !      c  c  _  i  i  a";
$msg "      1  1  1  1  1  1  m  m  e  o      s  s  c  t  t  d";
$msg "      k  5  4  3  2  1  w  r  t  e      1  0  s  1  2  y";
$msg "      ____________________________      ________________";
$msg "    Power On Reset                                     ";
          0  X  X  X  X  X  1  1  1  0      H  H  H  *  *  Z
$msg "    Reset Flip Flops                                   ";
          C  X  X  X  X  X  1  1  0  0      H  H  H  L  L  Z
$msg "    Write RAM0                                         ";
          0  0  0  1  0  0  0  1  0  0      H  L  H  L  L  Z
$msg "    Read  RAM0                                         ";
          0  0  0  1  0  0  1  0  0  0      H  L  H  L  L  Z
$msg "    Write RAM1                                         ";
          0  0  0  1  0  1  0  1  0  0      L  H  H  L  L  Z
$msg "    Read  RAM1                                         ";
          0  0  0  1  0  1  1  0  0  0      L  H  H  L  L  Z
$msg "    Begin ROM Read                                     ";
          0  0  0  0  0  0  1  0  0  0      H  H  L  L  L  L
$msg "  Two Clocks For Wait State, Then Drive READY High     ";
$repeat 2;
          C  0  0  0  0  0  1  0  0  0      H  H  L  *  *  *
$msg "    End ROM Read                                       ";
          0  0  0  0  0  0  1  1  0  0      H  H  H  H  H  Z
$msg "    End ROM Read                                       ";
          C  0  0  0  0  0  1  1  0  0      H  H  H  L  L  Z
```

FIGURE 2.25. CSIM (.SI) file for interface between CPU and memory

TABLE 2.2. Table of Test Conditions

Input	Definition
0	Drive input LO (0 volts)
1	Drive input HI (+5V)
C	Drive input LO,HI,LO
K	Drive input HI,LO.HI
L	Test output LO (0 volts)
H	Test output HI (+5V)
Z	Test output for high impedence
X	Input undefined, Output not tested
N	Power pins and Outputs not tested
P	Preload registers

The value of the clock variable, CPU_CLK is 0 in some vectors and C in others. A value of 0 causes no clocking to occur. A value of C causes CSIM to examine the input values in the vector for any registered outputs that would be fed back internally prior to the clock. Then after a clock is applied, CSIM computes the appropriate expected outputs for registered and nonregistered variables.

When CSIM is run a file is created (**waitgen.so**) which contains the result of the simulation (See Figure 2.26 on page 60). Comparison of the **.si** with the output file shows the vectors 8 and 9 were created as a result of the $REPEAT directive, and also CSIM has replaced the asterisks from the **.si** file with the appropriate logic levels (H and L) for the WAIT1, WAIT2 and READY signals.

If the any of the output tests had failed they would have been flagged with the actual output value displayed. Each variable that is incorrect is listed along with the expected (user-supplied) value. Any invalid or unexpected test values are recorded along with an appropriate error message.

2.2.4.3 JEDEC format

Once the CUPL source code has been written and tested, it is compiled into the JEDEC format, which is downloaded to the device programmer. Figure 2.27 on page 61 shows the JEDEC file for **waitgen.si**. It consists of an ASCII Start-of-Text (STX) character, followed by various fields of information, then an ASCII End-of-Text (ETX) character, and a transmission checksum.

```
 1: Name       Waitgen;
 2: Partno     P9000183;
 3: Date       03/14/85;
 4: Revision   02;
 5: Designer   Osann;
 6: Company    ATI;
 7: Assembly   PC Memory;
 8: Location   U106;
 9: Device     F155;
10:
11: /*************************************************************/
12: /* This device generates chip select signals for one         */
13: /* 8Kx8 ROM and two 2Kx8 static RAMs.  It also drives        */
14: /* the system READY line to  insert a wait-state of at       */
15: /* least one CPU clock for ROM accesses.                     */
16: /*************************************************************/
17:
18: ORDER:
19:          cpu_clk , %2, a15, %2, a14, %2,
20:          a13, %2, a12, %2, a11, %2,
21:          !memw, %2, !memr, %2, reset, %2, !oe,
22:          %4, !ram_cs1, %2, !ram_cs0, %2, !rom_cs, %2,
23:          wait1, %2, wait2, %2, ready;
24:
```

```
=============================================================
                      Simulation Results
=============================================================
     c                                  r  r  !
     p                                  a  a  r
     u                          !  !  r  m  m  o  w  w  r
     _                          m  m  e  _  _  m  a  a  e
     c  a  a  a  a  a  e  e  s  !  c  c  _  i  i  a
     l  1  1  1  1  1  m  m  e  o  s  s  c  t  t  d
     k  5  4  3  2  1  w  r  t  e  1  0  s  1  2  y
     ------------------------------   ------------------
     Power On Reset
0001: 0  X  X  X  X  X  1  1  1  0   H  H  H  H  H  Z
     Reset Flip Flops
0002: C  X  X  X  X  X  1  1  0  0   H  H  H  L  L  Z
     Write RAM0
0003: 0  0  0  1  0  0  0  1  0  0   H  L  H  L  L  Z
     Read  RAM0
0004: 0  0  0  1  0  0  1  0  0  0   H  L  H  L  L  Z
     Write RAM1
0005: 0  0  0  1  0  1  0  1  0  0   L  H  H  L  L  Z
     Read  RAM1
0006: 0  0  0  1  0  1  1  0  0  0   L  H  H  L  L  Z
     Begin ROM Read
0007: 0  0  0  0  0  0  1  0  0  0   H  H  L  L  L  L
  Two Clocks For Wait State, Then Drive READY High
0008: C  0  0  0  0  0  1  0  0  0   H  H  L  H  L  L
0009: C  0  0  0  0  0  1  0  0  0   H  H  L  H  H  H
     End ROM Read
0010: 0  0  0  0  0  0  1  1  0  0   H  H  H  H  H  Z
     End ROM Read
0011: C  0  0  0  0  0  1  1  0  0   H  H  H  L  L  Z
```

FIGURE 2.26. Output file (.SO) from simulator

```
<STX>
CUPL            3.2b  Serial# MD-32B-7769
Device          f155  Library DLIB-h-25-14
Created         Sun Jan 25 00:05:45 2065
Name            Waitgen
Partno          P9000183
Revision        02
Date            03/14/85
Designer        Osann
Company         ATI
Assembly        PC Memory
Location        U106
*QP20
*QF2108
*QV11
*G0
*F0
*L00000 10100110111111111010111111111111
*L00032 11111011111111111111111010011011
.
.
.
.
*L02080 00000000000101000000000000101
*C5AE5
*V0001 CXXXXX110N0HHLLZXXHN
.
.
.
*V0011 C0000011ON0HHLLXXZHN
*<ETX>74FF
```

FIGURE 2.27. JEDEC file for interface between CPU and memory

The design specification is the first field in the format (i.e. all information between the STX and the first asterisk). This information is for documentation purposes only, and consists of the header information from the CUPL source file along with version number of the compiler and device library.

At the start of the fields with asterisks there are characters that identify the type of information in the field. The Q character indicates a value. For example, the value QP describes the number of pins for the device. Another value field, QF, describes the total number of programmable fuses in the device. Both values are decimal numbers.

To enable the security fuse to be programmed on devices that have such an option, the security fuse field (G) instructs the device programmer to disable (G0) or enable (G1) the programming of the security fuse.

The default fuse state field (F) defines the state of the fuses that are not explicitly defined in the L field. It is the fuse link field (L) that contains the actual data. Each device fuse link is assigned a decimal number, starting with 0000. Each numbered fuse has two possible states: binary 0 specifies a low resistance link (FUSE INTACT) and binary 1 specifies a high resistance link (FUSE BLOWN).

The L identifier begins the field and is followed by the number of the first fuse being defined in the field. When more than one binary value is specified, the additional values are assigned to fuses numbered consecutively from the first fuse number. All the L fields are not shown in the listing for simplicity.

The next field is a fuse checksum (C) field. The checksum is a 16-bit hexadecimal value which is computed by adding 8-bit words from the specified state of each fuse link in the device. Link number 0 is the least-significant bit (lsb) and link number 7 is the most-significant bit (msb) of word 0. Unspecified bits in the final 8-bit word are set to zero before computing the checksum. In Figure 2.28 the first thirty-two fuses generate four 8-bit words.

word 00	1 0 1 0 1 1 0 1	→	AD
word 01	1 1 1 1 1 0 1 1	→	FB
word 02	0 1 1 1 0 0 1 1	→	73
word 03	1 1 1 0 1 1 0 1	→	EC
	Checksum	→	0307

FIGURE 2.28. Example of a Checksum

In order to allow the test vectors to be applied on the device rather than just simulated, a test vector field (V) can be created by running CSIM with the -j option flag (this is a flag added to the command line when CSIM is run). The test vector fields in the JEDEC code contain functional test information for each device being tested.

The test conditions, as they appear in the vector, are applied to the device pins in numerical order from left to right (the first condition is applied to pin 1 and the last to pin 20 of a 20 pin device). Signals C and K which drive the clock are presented after all the other inputs are stable. The L,H, and Z conditions are tested after all inputs have stabilised, including C and K.

The results of the test vectors are again presented in the .so output file.

The end of transmission is signified with a non-printing ASCII ETX character followed immediately by a transmission checksum (sum-check) of four ASCII hex characters. This checksum is the 16-bit sum of the ASCII values of all the transmitted characters between, and including, the starting STX and ending ETX characters.

2.3 Summary

This chapter described some of the fundamental principles of digital electronics. The basic logic gates used to construct digital circuits were detailed, along with how they are combined to form the basic element of memory, the flip-flop. It is these basic elements that are combined to form complex digital systems such as microcomputers, CD players etc.

Part 1 of this thesis is concerned with design and construction of various pieces of hardware for parallel computers. The PLDs described in this chapter are used frequently in these designs.

References

[1] Floyd, Thomas L. *"Digital Fundamentals"*, Macmillan Publishing Company, 1990, ISBN 0 02 946106 5

[2] Horowitz & Hill. *"The Art of Electronics - Second Edition"*, Cambridge University Press, 1991, ISBN 0 521 37095 7

[3] CUPL PAL Programming Manual

Chapter 3

Design of a Programmable Circuit Switched Network

The design and implementation of a programmable interconnection network which allows the user to alter the topology of the network prior to computation is described. This is achieved by connecting the high speed links on the nodes to two crossbar switches.

The various methods of interprocessor communication are detailed before describing some of the ICs utilised in the network. Design of the hardware boards is described and then the software required to program the crossbar switches is detailed.

3.1 Interprocessor Communication

There are basically three mechanisms used in interprocessor communication: packet switching, circuit switching[1] and wormhole routing[2,3].

3.1.1 Packet Switching

Packet switching is a form of message passing in which a message is split into smaller parts called packets. In a static interconnection network these packets are transmitted from the source node to the destination node via intermediate nodes. The packet at the head of the complete message has a header attached which defines the route to be taken at each crosspoint in the network. The packets are passed in a store and forward manner (i.e. the entire message has to arrive at one node before it is passed on to the next node.)

The message transmission time (i.e. the interval between the time when the beginning of a message leaves the source node and the time when the end of the message reaches the destination) for a packet switched scheme is given by: -

- message length x number of hops required to reach the destination.

The result of this is that as the message length is increased, the message transmission time increases rapidly. Packet switching schemes also require additional software on each node to manage the passing of the packets.

3.1.2 Circuit Switching

A more efficient method for large volumes of data which does not require additional software on each node is circuit switching. Circuit switching mechanisms establish a dedicated direct communication link between the two communicating nodes and this link is held until the message is completely transferred (like a telephone system). No dedicated communication software on each node is required only on the node which is setting up the communication links. The dedicated communication links can be set up before program execution or dynamically on demand during the program run-time.

3.1.3 Wormhole Routing

Wormhole routing is effectively a combination of packet switching and circuit switching. The packets are handled by special switches (routers) rather than by node software.

A message is divided up into a number of flow control digits or "flits" that are pipelined through the network. It is only the header flits of a message that are stored. The destination address in the header flits is decoded and, if the required link is free, the message body is transmitted as a stream from input to output without being stored at all.

As flits are forwarded, the message becomes spread out across the channels between the source and the destination. Message flits may not be interleaved with the flits of other messages as most flits do not contain routing information.

As each flit is forwarded to the next node as soon as it arrives (known as cut-through routing), the message transmission time is proportional to the sum of the message length and number of hops to reach the destination (i.e. it is faster than packet switching). This routing technique reduces the amount of node storage required compared to packet switching. If fast routers are used this technique can be more efficient than circuit switching.

The work presented in Part 1 of this thesis is concerned with the development of circuit switched schemes for interprocessor communication. This chapter describes the design and implementation of a circuit switched system which sets up dedicated communication links prior to computation.

3.2 INMOS products

This circuit switched network uses various products designed by INMOS. These are described in this section.

3.2.1 INMOS C004

The INMOS C004 is a 32 way crossbar switch which can be used to set up direct physical links between communicating nodes. A block diagram of the IMS C004 is illustrated below.

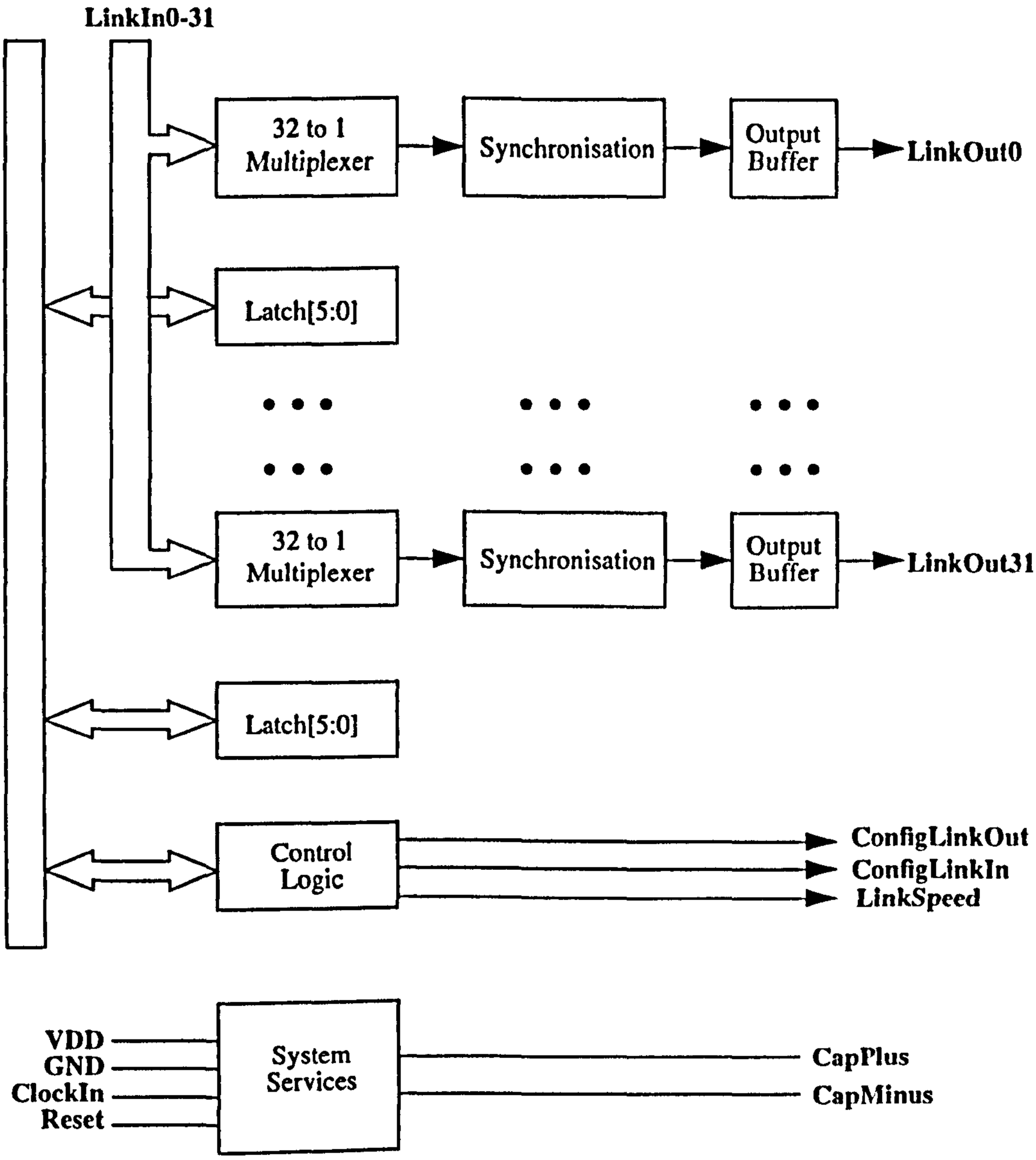


FIGURE 3.1. IMS C004 block diagram

3.2.1.1 Switch Implementation

The switch is internally organised as a set of thirty two 32-to-1 multiplexers. Each multiplexer has associated with it a six bit latch, five bits of which select one input as a corresponding source of data for the corresponding output. The sixth bit is used to connect and disconnect the output.

These latches are read and written to via the **ConfigLinkIn** and **ConfigLinkOut** pins. The user sends configuration messages to the switch (consisting of one, two or three bytes) via the **ConfigLinkIn** pin and receives any data sent back from the switch via the **ConfigLinkOut** pin (See Table 3.1.). Each input and output is identified by a number in the range 0 to 31.

Configuration Message	Function
[0] [input] [output]	Connects **input** to **output**
[1] [link1][link2]	Connects **link1** to **link2**
[2] [output]	Enquires which input the **output** is connected to. The IMS C004 responds with the input.
[3]	This command byte must be sent at the end of every configuration sequence which sets up a connection.
[4]	Resets the switch. All outputs are disconnected and held low.
[5] [output]	Output **output** is disconnected and held low.
[6] [link1][link2]	Disconnects the output of **link1** and the output of **link2**.

TABLE 3.1. IMS C004 configuration messages

3.2.1.2 INMOS OSLinks

The INMOS C004 uses INMOS OS Links (i.e **LinkIn0-31** and **LinkOut0-31**). These bi-directional serial links provide synchronised communication between INMOS products and the outside world. Each link comprises an input and output channel (i.e.**LinkIn** and **LinkOut**). A link between two devices is implemented by connecting input to output and output to input.

Every byte of data sent on a link is acknowledged on the input of the same link. A receiver can transmit an acknowledge as soon as it starts to receive a data byte. This allows the transmission of an acknowledge byte to be overlapped with the receipt of a data byte to provide continuous transmission of data.

The quiescent state of a link output is low. Data bytes are transmitted as a two high (+5V) start bits followed by eight data bits (the least significant bit of data is transmitted first) followed by a low (0V) stop bit (See Figure 3.2). After transmitting a data byte the sender waits for an acknowledge which comprises of a high start bit followed by a low stop bit. This acknowledge signifies to the sender that the receiver is ready to receive another byte of data.

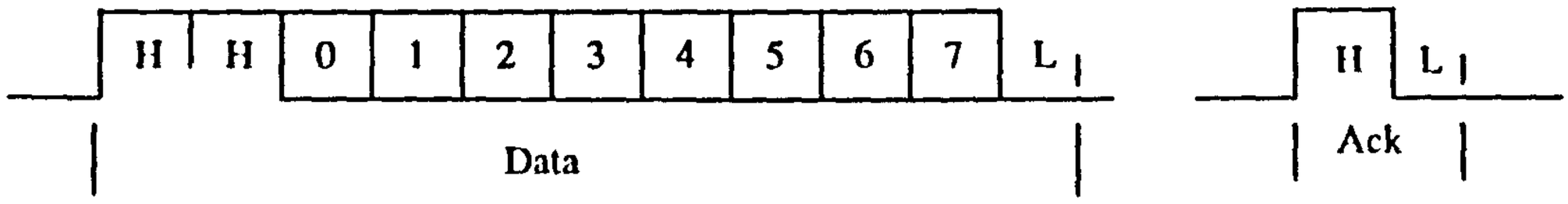


FIGURE 3.2. IMS C004 link data and acknowledge packets

INMOS OS Links run at speeds of 10Mbits/s and 20 Mbits/s. When the **LinkSpeed** pin on the C004 is logic low all links operate at the 10Mbits/s and when this pin is pulled logic high the links operate at 20Mbits/s. Links are not synchronised with **ClockIn** (a 5MHz crystal oscillator), enabling links from independently clocked systems to communicate, providing only that the clocks are nominally identical and within specification.

3.2.1.3 System Services

Descriptions of the function of the system services pins on a C004 are shown in Table 3.2. below.

Pin	In/Out	Function
VDD, GND		Power supply and return
CapPlus, CapMinus		External capacitor (1μF) for internal power supply
ClockIn	in	Input clock
Reset	in	System reset

TABLE 3.2. IMS C004 system services

3.2.2 INMOS T-800 transputer

The T-800 transputer[4] is a 32-bit microprocessor designed specifically to be used in a distributed memory multiprocessor environment. It has four on-chip high speed serial

data links for communication between processors as well as a peripheral interface. A block diagram of the T-800 transputer is illustrated in Figure 3.3.

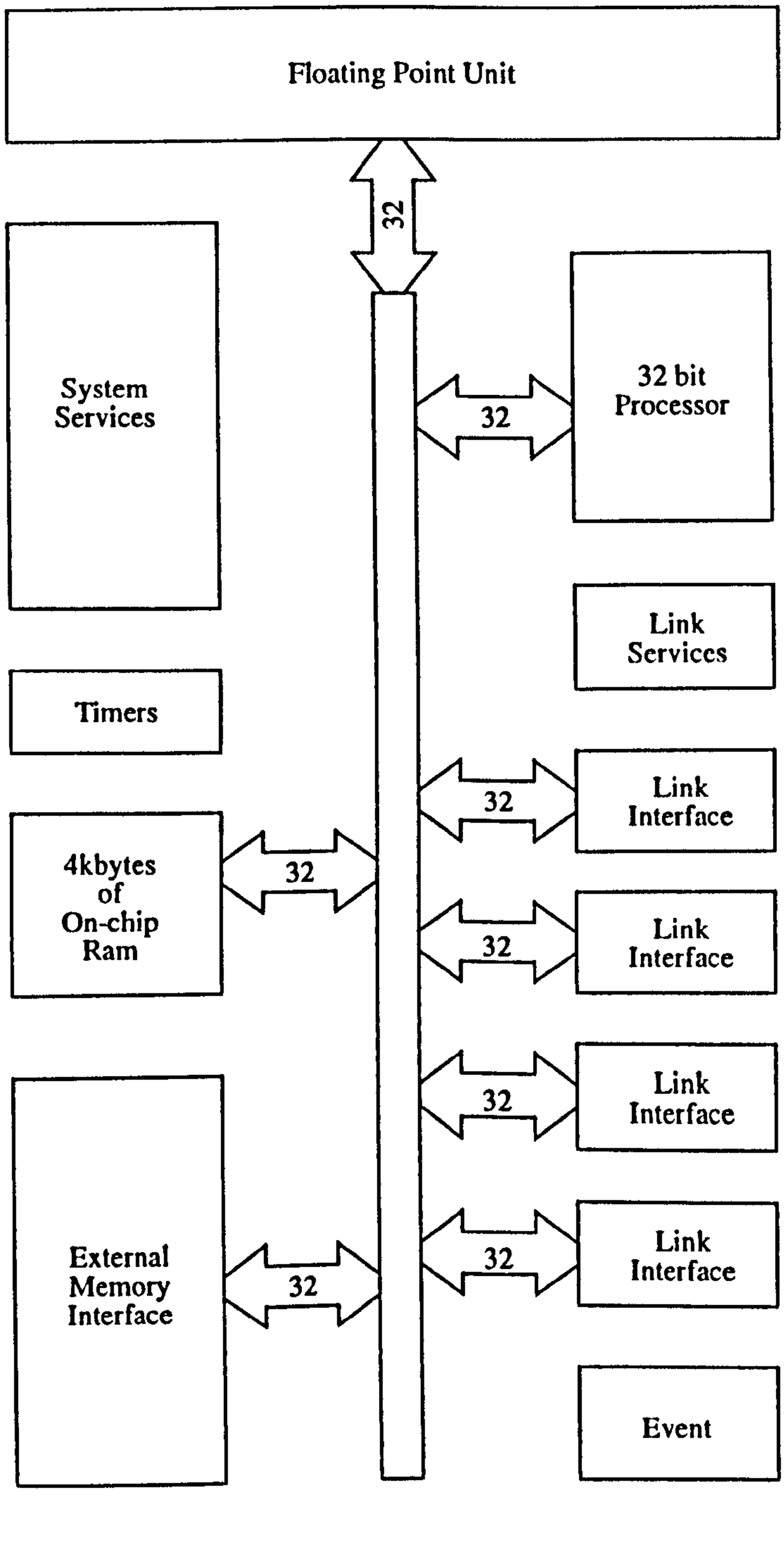


FIGURE 3.3. IMS T-800 block diagram

A transputer can be used in a single processor system or in a network using the INMOS OS Links (as described in Section 3.2.1.2 on page 67) to connect the transputers together. Within a single transputer the CPU operates a time sharing system whereby it can share its time between any number of concurrent processes. This allows users to

develop parallel programs on single transputer and then run them on a network of transputers with little alteration.

The IMS T-800 uses a DMA (Direct Memory Access) mechanism to transfer messages between memory and another transputer product via the INMOS OSLinks. This allows the link interfaces and the CPU to operate concurrently; i.e programs can continue execution whilst data is being transferred on the links.

Whilst transputers can be programmed in most high level languages such as FORTRAN and C a special purpose parallel language called OCCAM[5] was developed for the transputer. OCCAM can be used with other microprocessors although its principal use is with transputers. By using OCCAM the system designers task is eased because of the architectural relationship between OCCAM and the transputer.

OCCAM is based on the process model of computation. A process is an independent computation with its own program and data, which can communicate with other processes executing at the same time. A process can be thought of as a black box with inputs and outputs, that can communicate by message passing using explicitly defined channels.

Processes are connected together by channels which are built up to produce complex concurrent systems. Communication between processes is synchronised; if a process A tries to send a message to process B on channel C, it will block until B is ready to receive on channel C. A channel can be an INMOS OS Link between transputers or a software channel between processes on the same transputer.

OCCAM enables a system to be described as a collection of concurrent processes which communicate with each other through channels. An OCCAM program may execute on an array of transputers and the same program can also execute almost unchanged on a smaller array, or even on a single transputer. An OCCAM channel describes communication in the abstract and does not depend upon a particular hardware implementation. The processes that communicate via channels can be on the same transputer or different transputers.

Examples of statements which send and receive variables on channels are shown in Figure 3.4. The symbol **?** is for input in OCCAM and **!** is for output.

chan1 ? xvar	sets the variable **xvar** to the value input from the channel **chan1**
chan2 ! yvar	outputs the value of the variable **yvar** to the channel **chan2**

FIGURE 3.4. Examples of input and output statements

Transputers are connected together via INMOS OSlinks but there has to be some way of interfacing these links to the outside world (i.e. in order that a network of transputers can be loaded with code and data at least one transputer has to be connected to a host computer). This is achieved by a device called a C012.

3.2.3 C012 Link Adaptors

The INMOS C012[4] is a link adaptor which interfaces INMOS serial OS links to microprocessor buses (amongst other things) by converting the bi-directional serial link into parallel data streams. A block diagram of the IMS C012 is illustrated in Figure 3.5.

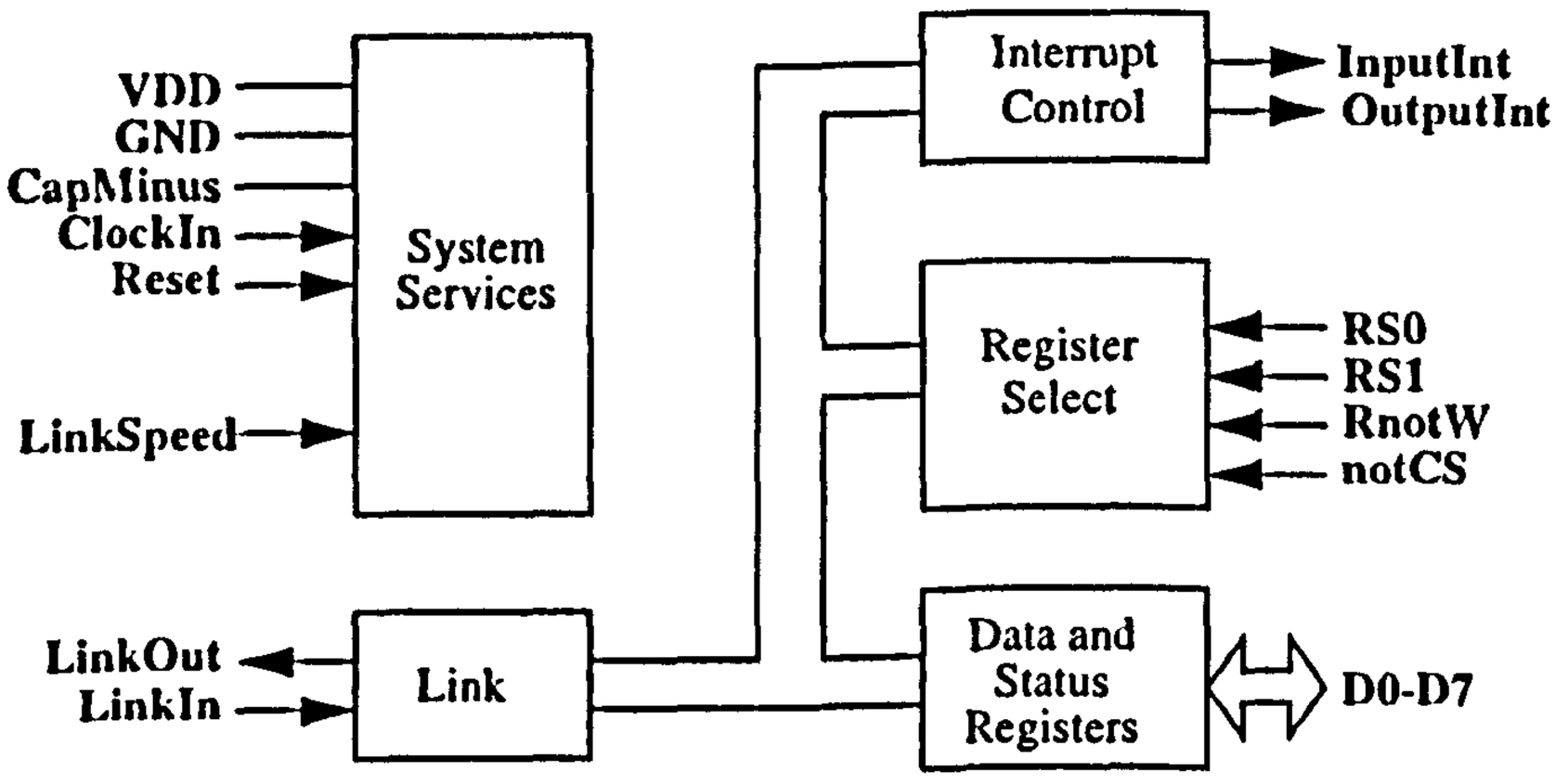


FIGURE 3.5. IMS C012 block diagram

The status and data registers for both input and output ports can be accessed via the byte wide bi-directional interface (**D0-D7**). Registers are selected by **RS0-1** and **RnotW**, and the chip is enabled by **notCS** (i.e. the chip is enabled when **notCS** is low).

RnotW selects the registers for read or write mode. When **RnotW** is high, the contents of the addressed register appear on the data bus **D0-D7** and when **RnotW** is low the data on **D0-D7** is written into the addressed register.

RSO-RS1 select one of the four registers; the read-only data input register, the write-only data output register or the read/write status registers. The addresses for the registers are shown in Table 3.3. .

TABLE 3.3. IMS C012 register selection

RS1	RS0	RnotW	Register
0	0	1	Read Data
0	0	0	Invalid
0	1	1	Invalid
0	1	0	Write Data
1	0	1	Read Input Status
1	0	0	Write Input Status
1	1	1	Read Output Status
1	1	0	Write Output Status

The input data register holds the last data packet received from the INMOS serial OS Link. It never contains an acknowledge packet. The output data register contains data that is to be transmitted out of the serial link as a data packet.

The input status register contains the 'data present' flag and the 'interrupt enable' control bit for **InputInt** (See Figure 3.6). The 'data present' flag is set to indicate that data in the data input buffer is valid. It is reset low only by reading the data input buffer, or by **Reset**. When writing to this register, the 'data present' bit must be written as zero. The **InputInt** output is set high when a data packet has been received on the INMOS OS serial link. It is inhibited from going high if the 'interrupt enable' bit is set to low.

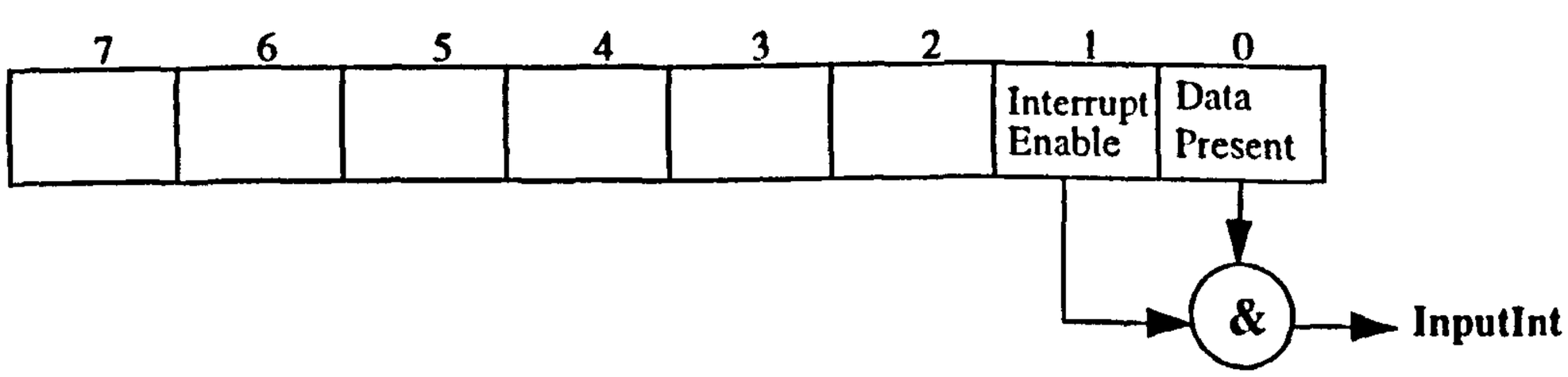


FIGURE 3.6. IMS C012 input status register

The output status register contains the 'output ready' flag and the 'interrupt enable' control bit for **OutputInt** (See Figure 3.7). The 'output ready' flag is pulled high to indicate that the data output buffer is empty and it is only reset low when data is written to the data output buffer; it is set high by **Reset**. When writing to this register, the 'output ready' bit must be written as zero. The **OutputInt** output is set to indicate that the INMOS OS Link is ready to receive data from **D0-D7.** It is inhibited from going high when the 'interrupt enable' bit is set low.

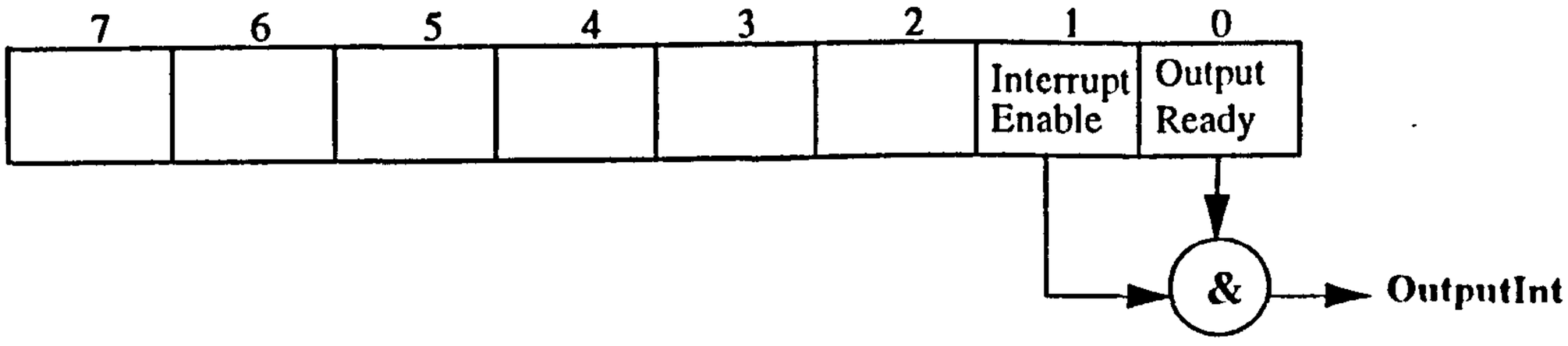


FIGURE 3.7. IMS C012 output status register

The system services for the C012 are the same as for the C004.

3.3 Hardware for Static Circuit Switched Network

The basic layout of the static circuit switched network is illustrated in Figure 3.8. Two of the links on a node are used to connect the nodes in a pipeline and the remaining two links are connected to two crossbar switches. This allows Link 1 on a node to be connected to Link 2 on any other node. The crossbar switches used in the system are INMOS C004s and the nodes are INMOS T-800 transputers although the same principles could be applied with other nodes and crossbar switches.

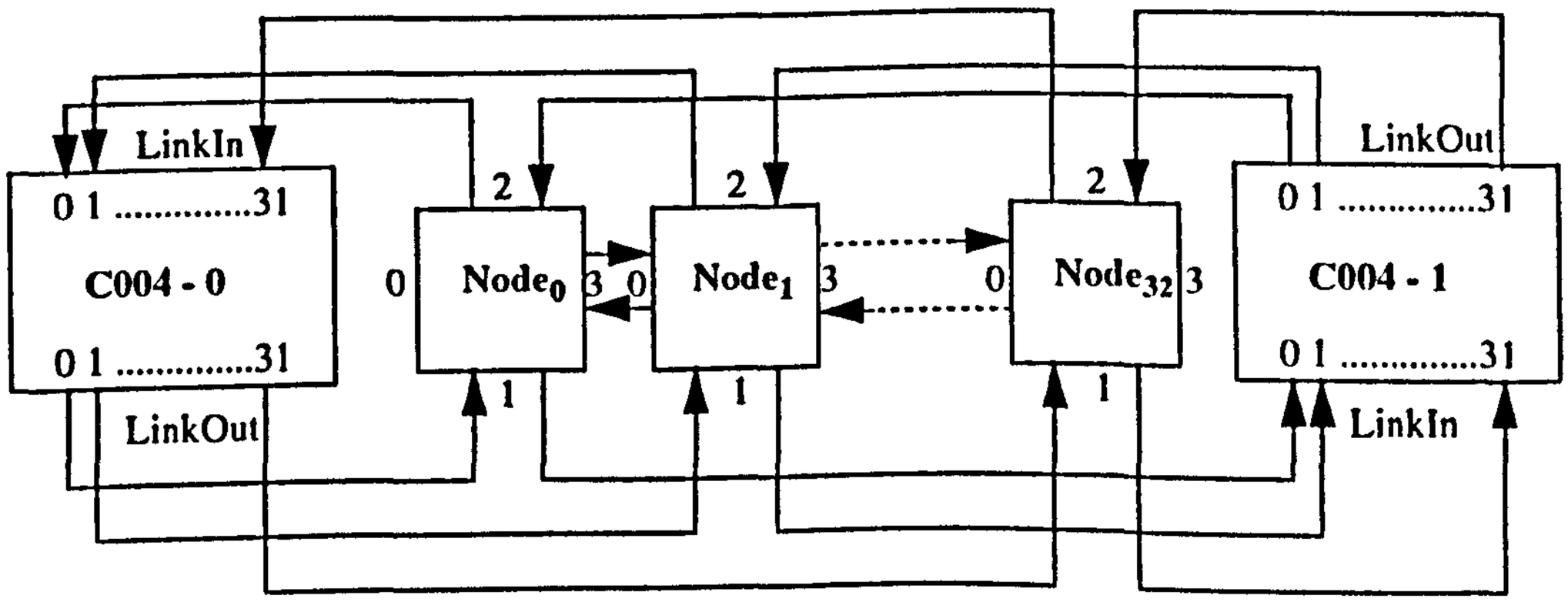


FIGURE 3.8. Layout of circuit switched network

3.3.1 Hardware setup

The parallel machine used with the switch system contains thirty two transputers arranged on four printed circuit boards with eight transputers on each. Each circuit board contains a DIN41612 plug which all the links on the transputers are connected to (See Figure 3.9).

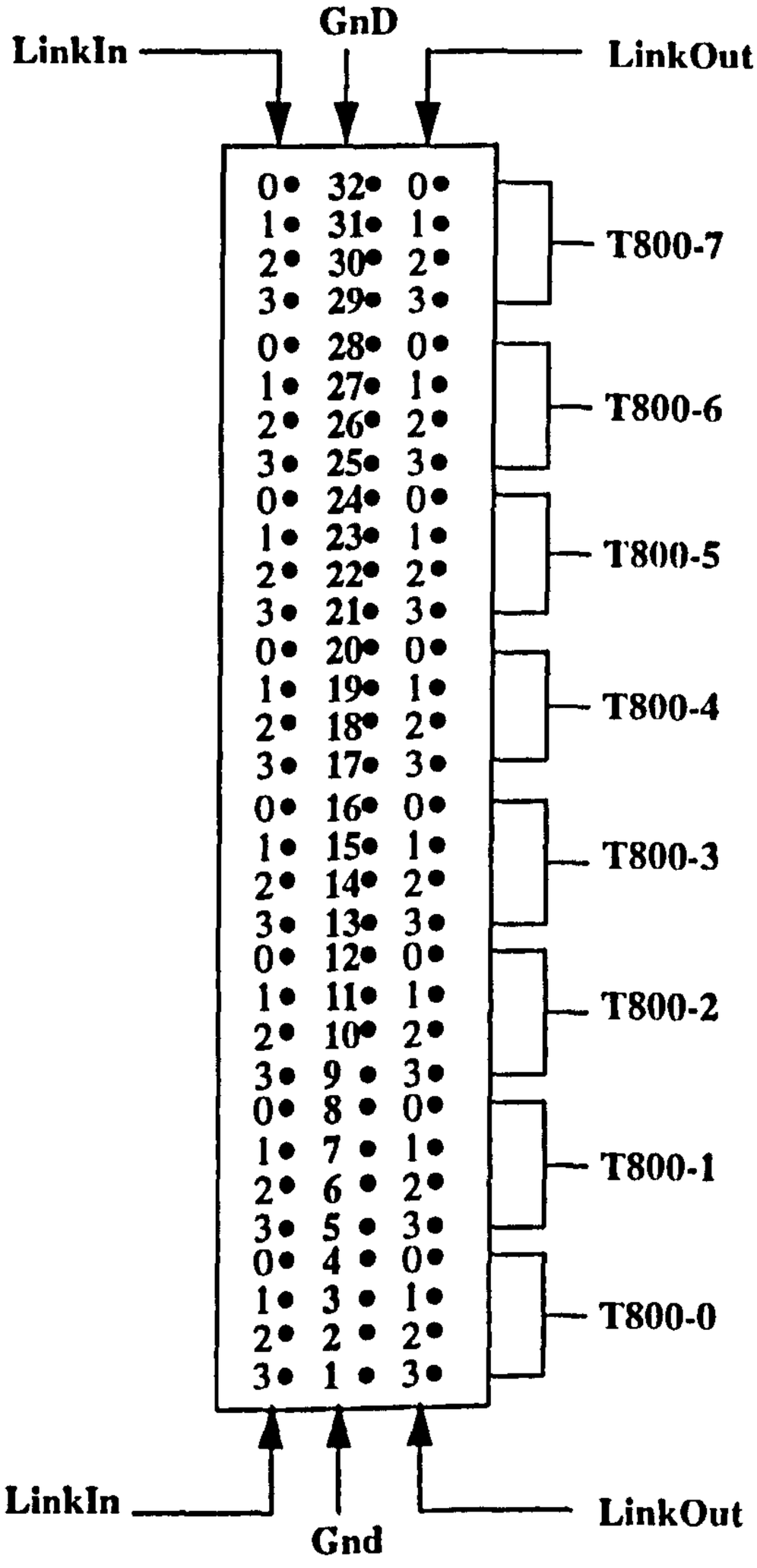


FIGURE 3.9. Connections from transputer board to DIN41612 plug

In order to connect the transputers and C004s in the configuration shown in Figure 3.8 on page 73 a circuit board (switch board) was constructed which plugged into the transputer boards (See Figure 3.10 and Figure 3.11 on page 75). The 32-way DIN41612 sockets on the switch board are plugged into the equivalent plugs on the

transputer boards. The sockets on the switch board are wired such that they connect links 0 and 3 in a pipeline and links 1 and 2 to the crossbar switches. A photograph of the board is shown in Appendix D, Figure 1 on page 304.

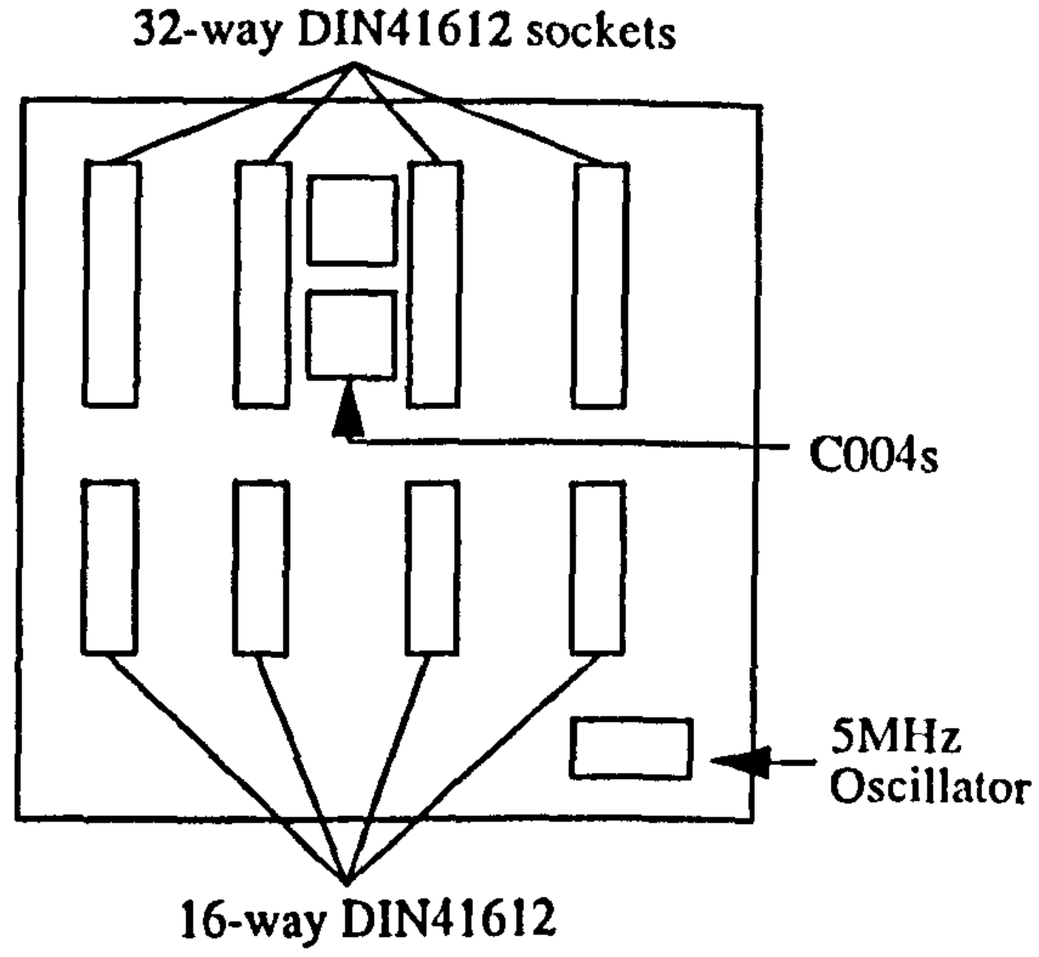


FIGURE 3.10. Block Diagram of Switch board

The overall arrangement of the boards is illustrated in Figure 3.11 (the transputer boards are housed in a purpose built box). Two of the 16-way DIN41612 connectors are attached to the host computer (a PC) which sends the messages to program the required connections on the C004s. The connections to the 16-way DIN41612 sockets on the switch board are shown in Figure 3.12 on page 76. These two 16-way DIN41612 sockets are connected to a dual C012 link adaptor board plugged into the host computer.

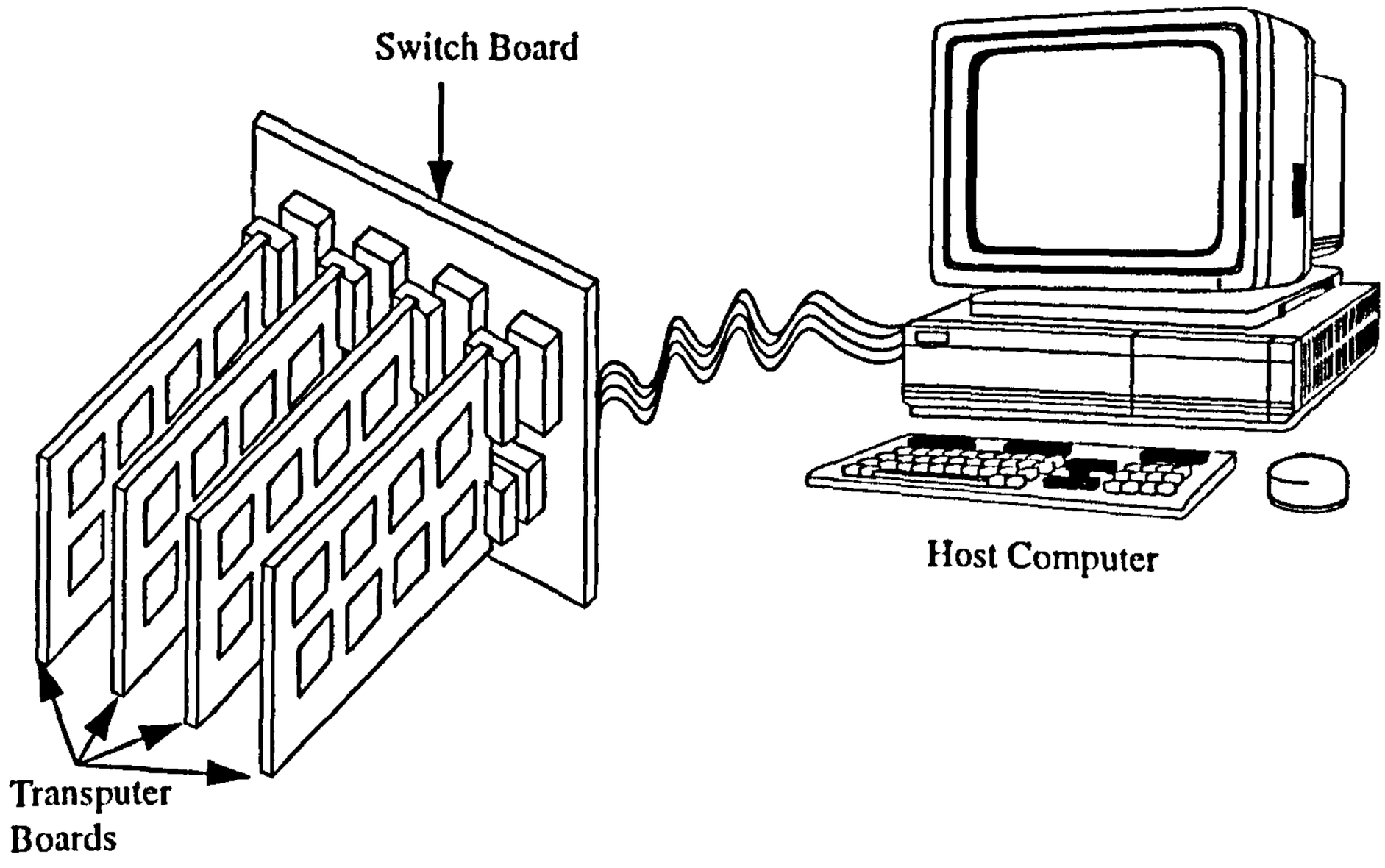


FIGURE 3.11. Overall arrangement of transputer boards

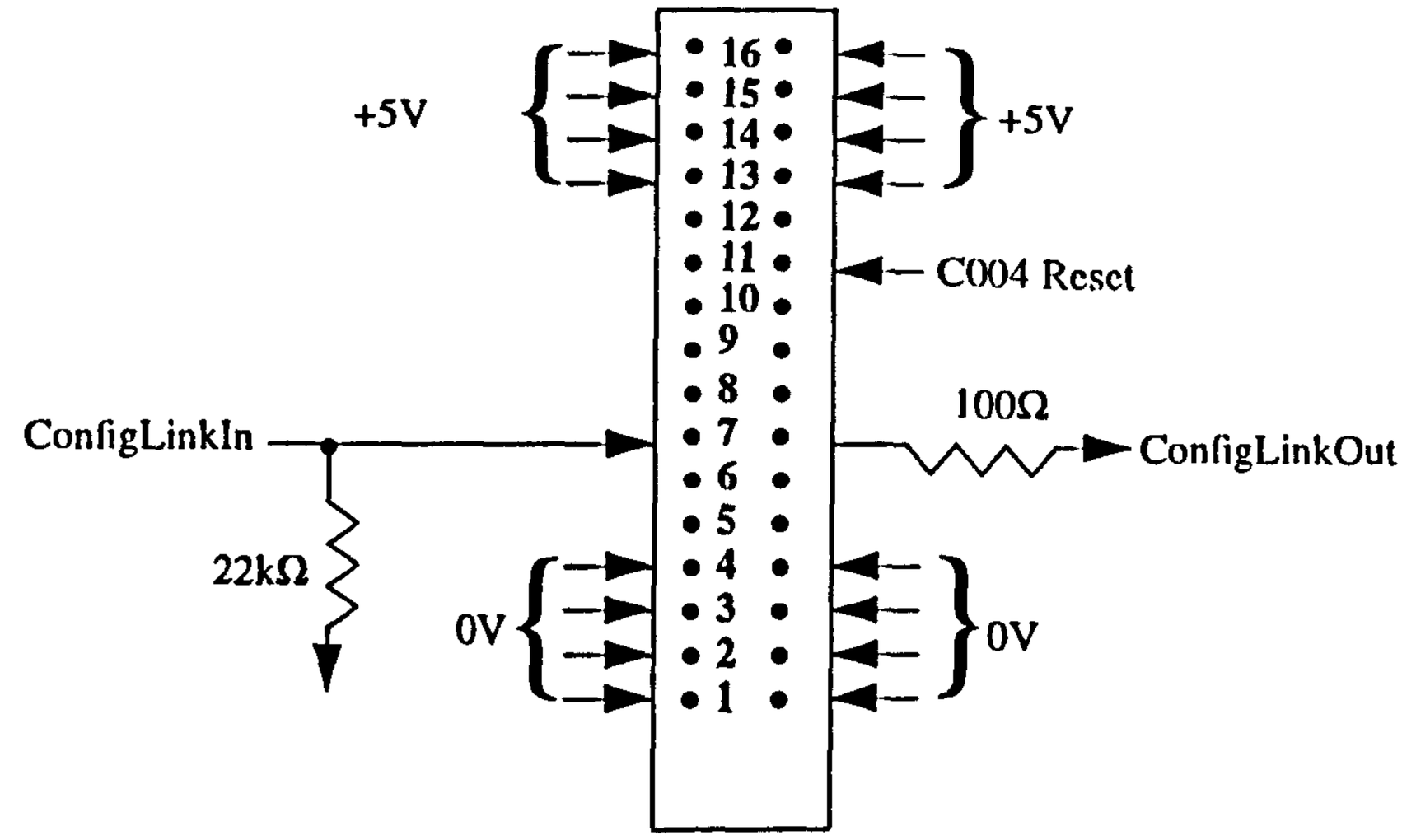


FIGURE 3.12. Connections to 16-way DIN41612 socket

3.3.2 Dual Link Adaptor Board

The dual link adaptor circuit board, which connects to the two C004s on the switch board, contains two C012s interfaced to the PC bus. A PC contains several (~8) system-bus expansion card slots[6]. The connectors on the card are capable of supporting 62 signal connectors to a card, 31 on each side of a card (See Figure 3.13 on page 77).

Cards are retained by attaching an L bracket to the back end of the card; the bracket, in turn, attaches to the top of the system unit's bulkhead. Cables are attached to the card by attaching a connector to the card, and extending the connector through the L bracket out through the slots cut in the rear of the bulkhead of the system unit.

The dual link adaptor circuit was built on a wire-wrap card of this type. The two C012s each have an address and the PC communicates with them by reading/writing data at this address. A circuit diagram of the card is shown in Figure 3.14 on page 78 and a photograph of the card is shown in Appendix D, Figure 2 on page 304.

The ICs used on the board are the two C012s, an octal bus transceiver with tri-state outputs (SN74LS245) and two programmable logic devices (P22V10L). The DB9 connector is used to connect to the switch board.

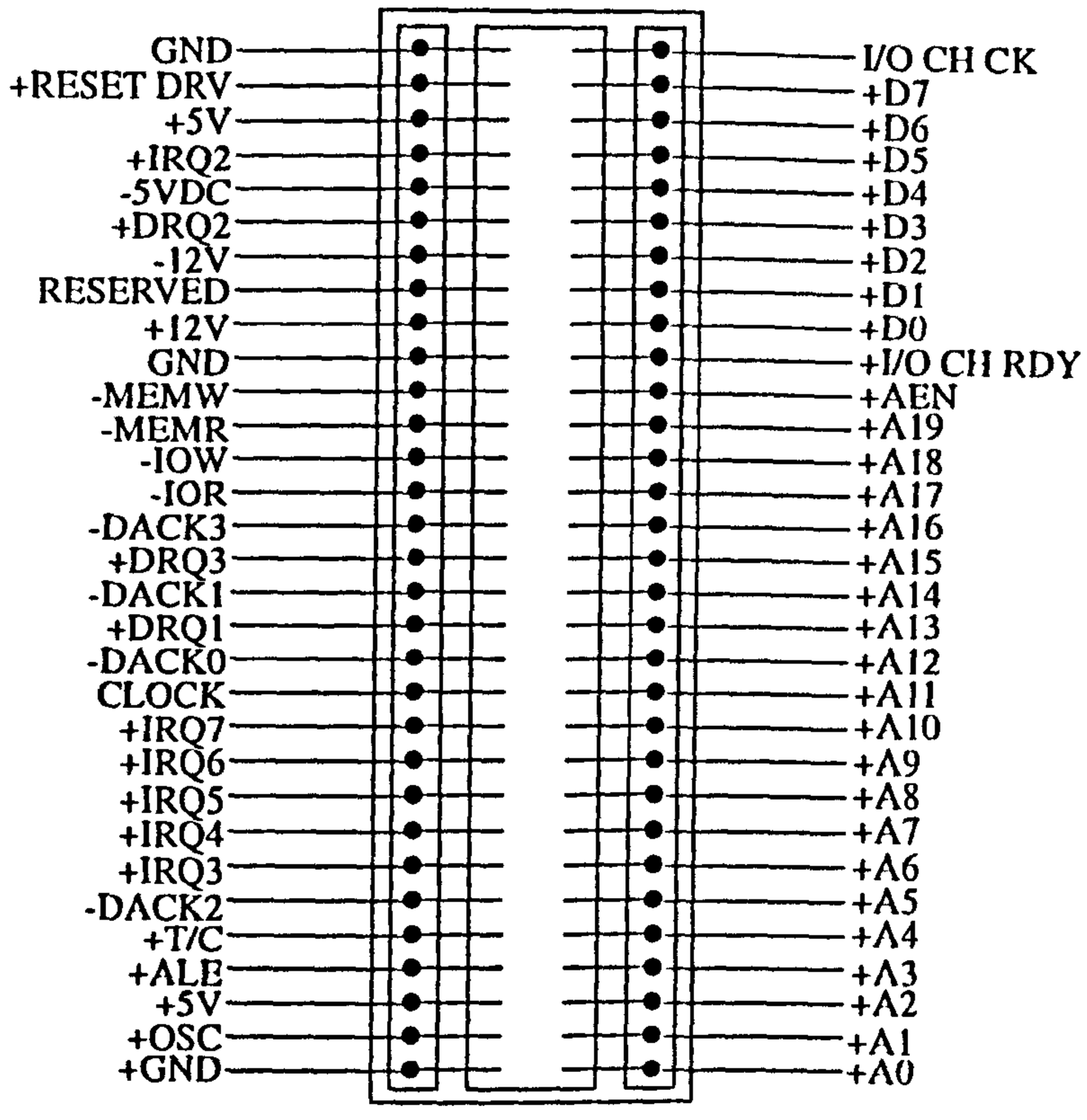


FIGURE 3.13. Pin and signal definitions for the PC card slots

3.3.2.1 245' Octal Bus Transceiver

The SN74LS245[7] acts as a buffer between the C012s and the PC bus. It allows transmission from the A bus to the B bus and vice versa depending on the logic level at the direction control (**DIR**) input (See Table 3.4.). When the enable input (**G***) is pulled high the device is effectively isolated from the A and B buses. The **DIR** and **G*** inputs are generated from a P22V10.

ENABLE (G*)	DIRECTION CONTROL (DIR)	OPERATION
L	L	B data to A bus
L	H	A data to B bus
H	X	Isolation

TABLE 3.4. Function Table for 245'

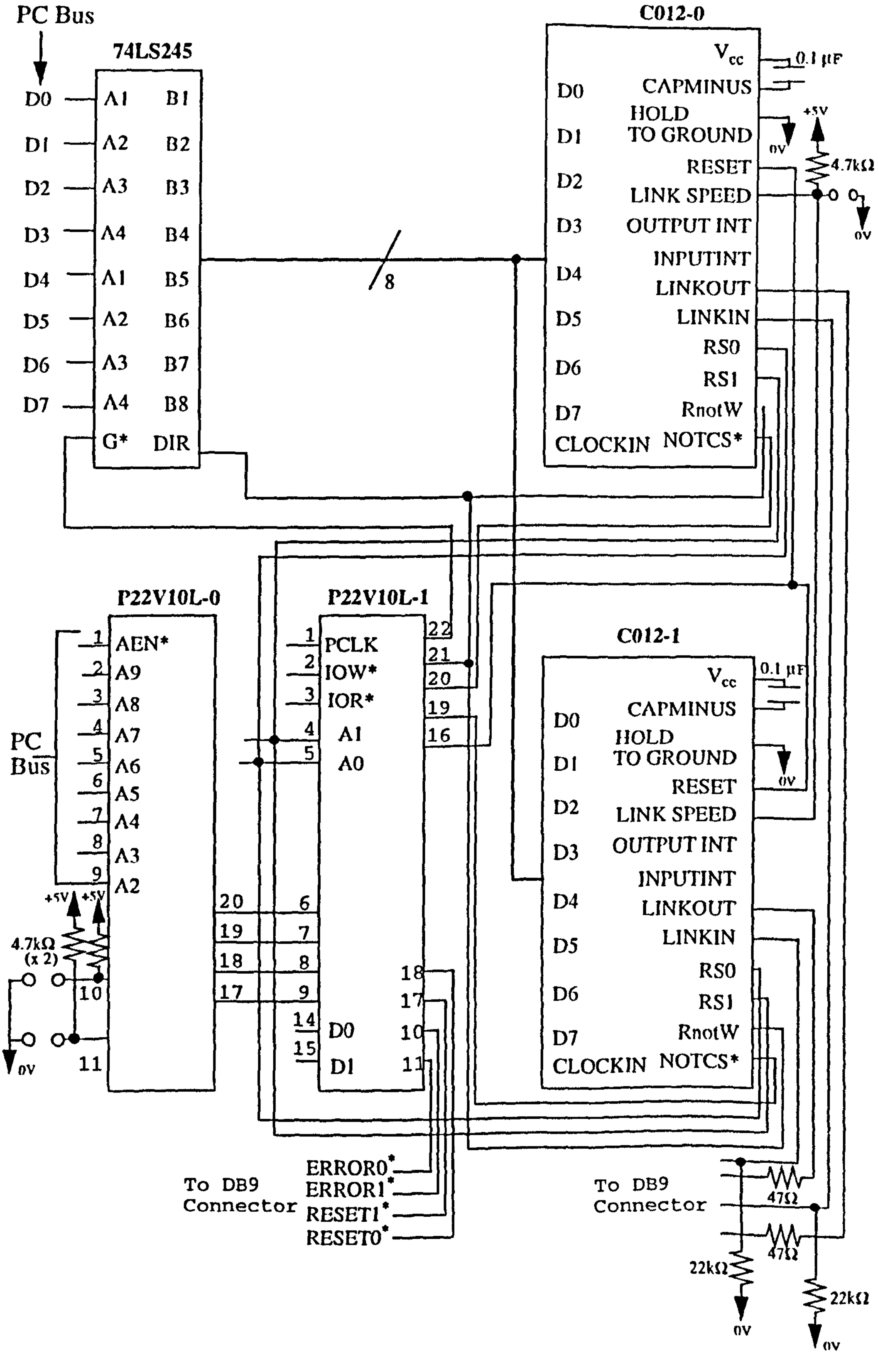


FIGURE 3.14. Dual Link Adaptor Board

3.3.2.2 P22V10L-0 Programmable Logic Device

The P22V10L is a CMOS high performance electrically erasable 24 pin PAL[8]. A diagram of the pin configuration of the PAL is shown in Figure 3.15. It contains 10 D-type registers which are provided with synchronous preset and asynchronous reset terms.

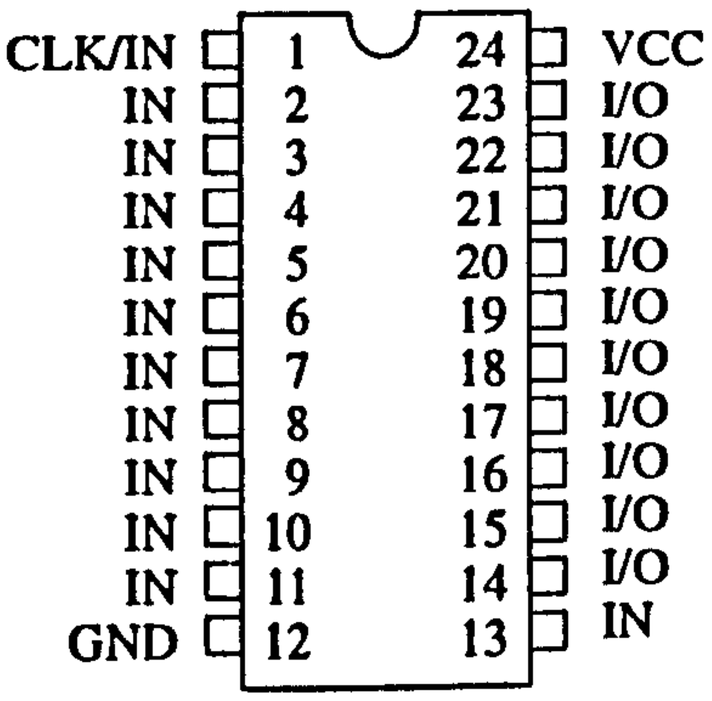


FIGURE 3.15. Pin Configuration of P22V10L

The CUPL source code for both the PALs is shown in Figures 3.16 and 3.17 on pages 80, 82 and 83. Between them, the PALs generate the **notCS, RnotW** and **Reset** signals for the C012s, and the **DIR** and **G*** signals for the buffer.

P22V10L-0 decodes the addresses for the link adaptors and control information. The inputs to P22V10L-0 are the PC bus address lines (**A2-A9**), the address enable signal (**AEN**) and the select signals (**Se10 & Se 11**). The active low outputs are **NotSys0, NotSys1, NotLadp0** and **Not Ladp1.**

A FIELD declaration is used to define the address bus (i.e. **FIELD IBMaddr = [A9 .. A2]**. When the FIELD variable is used with an equality operator (i.e. **IBMAddr : [120]**), CUPL assumes that the address bus includes **A1** and **A0** although they are not in fact connected to the PAL. The equality operator therefore compares **A2-A9** with the top eight bits of the hexadecimal number **120** (in this case).

The **AEN** signal is an active high signal from the PC which indicates when a DMA (Direct Memory Access) cycle is in operation (i.e. a DMA device has access to the PC bus). The C012s must not be accessed during a DMA cycle and therefore the **AEN** signal must be logic low during a read/write cycle to the C012s. This signal is therefore made active low in the CUPL pin assignments.

```
PIN 1 =         !NotAEN;
PIN 2 =         A9;
PIN 3 =         A8;
PIN 4 =         A7;
PIN 5 =         A6;
PIN 6 =         A5;
PIN 7 =         A4;
PIN 8 =         A3;
PIN 9 =         A2;
PIN 10=         Sel0;
PIN 11=         Sel1;

PIN 17 =        !NotSys0;
PIN 18 =        !NotSys1;
PIN 19 =        !NotLadp0;
PIN 20 =        !NotLadp1;

FIELD           IBMAddr = [A9..A2];
FIELD           Select  = [Sel1..Sel0];

/** Enable link adaptor and system control signals **/
NotLadp1 =      IBMAddr:[120] & Select:[1] & NotAEN #
                IBMAddr:[220] & Select:[2] & NotAEN #
                IBMAddr:[320] & Select:[3] & NotAEN ;

NotLadp0 =      IBMAddr:[124] & Select:[1] & NotAEN #
                IBMAddr:[224] & Select:[2] & NotAEN #
                IBMAddr:[324] & Select:[3] & NotAEN ;

NotSys1 =       IBMAddr:[130] & Select:[1] & NotAEN #
                IBMAddr:[230] & Select:[2] & NotAEN #
                IBMAddr:[330] & Select:[3] & NotAEN ;

NotSys1 =       IBMAddr:[134] & Select:[1] & NotAEN #
                IBMAddr:[234] & Select:[2] & NotAEN #
                IBMAddr:[334] & Select:[3] & NotAEN ;
```

FIGURE 3.16. CUPL source code for P22V10L-0

Signals **Se10 & Se 11** are controlled by jumpers (See Figure 3.14 on page 78). These signals are assigned to the FIELD variable **Select** and are used to select which address of the three alternatives is used. This is to give greater flexibility when selecting addresses for the C012s.

Table 3.5. shows the output signals and their meaning. These signals are logic true when one of three alternative addresses is set to the appropriate value, the select signal is set to the correct value for that address and the **NotAEN** signal is logic true. These outputs are then fed into P22V10L-1.

Pin Outs	Definition
NotLadp1	Selects link adaptor 1
NotLadp0	Selects link adaptor 0
NotSys1	Selects control information for link adaptor 1
NotSys0	Selects control information for link adaptor 0

TABLE 3.5. Pin Outs of P22V10L-1

3.3.2.3 P22V10L-1 Programmable Logic Device

P22V10L - 1 generates the signals **notCS0, notCS1, RnotW(notWrite)** and **Reset (C012Reset)** for the C012s and the **DIR** and **G*(BufEn)** for the **'245** (See Figure 3.17 on pages 82 and 83 for CUPL code).

The inputs to P22V10L - 1 are **PCLK** (PC Clock), **IOR*** (PC Read Cycle), **IOW*** (PC Write Cycle), **A1, A0, NotIBMError1, NotIBMError0** and the signals (**NotSys0, NotSys1, NotLadp0** and **Not Ladp1**) from P22V10L-0. The signal **D0** from the PC data bus is used as both an input and output. The signals **NotIBMReset1, NotIBMReset0** and **NotStatWr** are inputs to the D-type internal registers of the P22V10L.

Inputs **A0** and **A1** from the PC address bus are assigned to the FIELD variable **Register**. These inputs are also connected to the **R0** and **R1** pins on the C012s. They are therefore used to address the various registers on the C012 (See Table 3.3. on page 72). When addressing a C012 it is therefore the first two bits of the address that specify which register is being read/written. **A0** and **A1** are also connected to the P22V10 as when writing a "reset" to a C012 both these bits require to be zero.

```
PIN 1     =                    PClk;/* Register Clock */
PIN 2     =                    !NotIOR;
PIN 3     =                    !NotIOW;
PIN 4     =                    A1;
PIN 5     =                    A0;
PIN 6     =                    !NotLadp1;
PIN 7     =                    !NotLadp0;
PIN 8     =                    !NotSys1;
PIN 9     =                    !NotSys0;
PIN 10    =                    !NotIBMError1;
PIN 11    =                    !NotIBMError0;

PIN 14    =                    D0;
PIN 15    =                    D1;
PIN 16    =                    C012Reset;
PIN 17    =                    NotIBMReset1;
PIN 18    =                    NotIBMReset0;
PIN 19    =                    !NotCs1;
PIN 20    =                    !NotCs0;
PIN 21    =                    !NotWrite;
PIN 22    =                    !BufEn;
PIN 23    =                    !NotStatWr;

FIELD     Register = [A1..A0];
FIELD     Outputs= [NotIBMReset1,NotIBMReset0,NotStatWr]

/** Resets & Presets **/
Outputs.ar     =          'b'0; /** Switch off all async resets **/
Outputs.sp     =          'b'0; /** Switch off all sync presets **/

/** Definitions **/
ReadSys1       =          NotIOR & NotSys1;
WriteSys1      =          NotIOW & NotSys1;
ReadSys0       =          NotIOR & NotSys0;
WriteSys0      =          NotIOW & NotSys0;
```

```
ReadLink1       =           NotIOR & NotLadp1;
WriteLink1      =           NotIOW & NotLadp1;
ReadLink0       =           NotIOR & NotLadp0;
WriteLink0      =           NotIOW & NotLadp0;

WriteReset1     =           WriteSys1 & Register:[0];
WriteReset0     =           WriteSys0 & Register:[0];

/** C012 & 245 control signals **/
NotStatWr.d     =           NotIOW;
NotCs1          =           WriteLink1 & NotStatWr # ReadLink1;
NotCs0          =           WriteLink0 & NotStatWr # ReadLink0;
NotWrite        =           NotIOW # NotStatWr;
BufEn           =           NotCs1 # NotCs0;
C012Reset       =           NotIBMReset1 # NotIBMReset0;

/** Reset, analyze, & error **/
NotIBMReset1.d =            D0 & WriteReset1 #
                            NotIBMReset1   & !WriteReset1;
NotIBMReset0.d =            D0 & WriteReset0 #
                            NotIBMReset0
D1              =           !NotIBMError1;
D1.oe           =           ReadSys1;
D0              =           !NotIBMError0;
D0.oe           =           ReadSys0;
```

FIGURE 3.17. CUPL source code for P22V10L-1

The inputs to the D-type registers (**NotIBMReset1, NotIBMReset0, NotStatWr**) are assigned to the FIELD variable **Outputs.** The asynchronous resets and synchronous presets of these registers are turned off by the statements, **Outputs.ar = 'b'0** and **Outputs.sp = 'b'0** (i.e. the **ar** and **sp** are set to binary ('**b**') zero (OV)).

Table 3.6. on page 84 shows the intermediate variables and their definitions. These are declared in order to simplify later expressions.

The **NotStatWr** signal is required to effectively create a delayed **IOW*** signal which is used to generate the **notCS*** signals for the C012s. The **notCS*** signals must be logic

true when reading or writing to a C012. A timing diagram for writing data to a C012 is shown in Figure 3.18. This shows that there must be a gap between the **RnotW** signal being pulled low and the **notCS*** going low. It is therefore not correct to use the expression **notCS = WriteLink # ReadLink** to generate the **notCS** signals as the **WriteLink** signals are generated from the **IOW*** signal as is the **RnotW** signal (i.e. the **RnotW** signal will become logic true at the same time as **notCS**.

Pin Out	Definition
ReadSys1	Read system info. from Link Adaptor 1
WriteSys1	Write system info. to Link Adaptor 0
ReadSys0	Read system info. from Link Adaptor 1
WriteSys0	Write system info. to Link Adaptor 0
ReadLink1	Read data from Link Adaptor 1
WriteLink0	Write data to Link Adaptor 0
ReadLink1	Read data from Link Adaptor 1
ReadLink0	Write data to Link Adaptor 0

TABLE 3.6. Intermediate variables for P22V10L

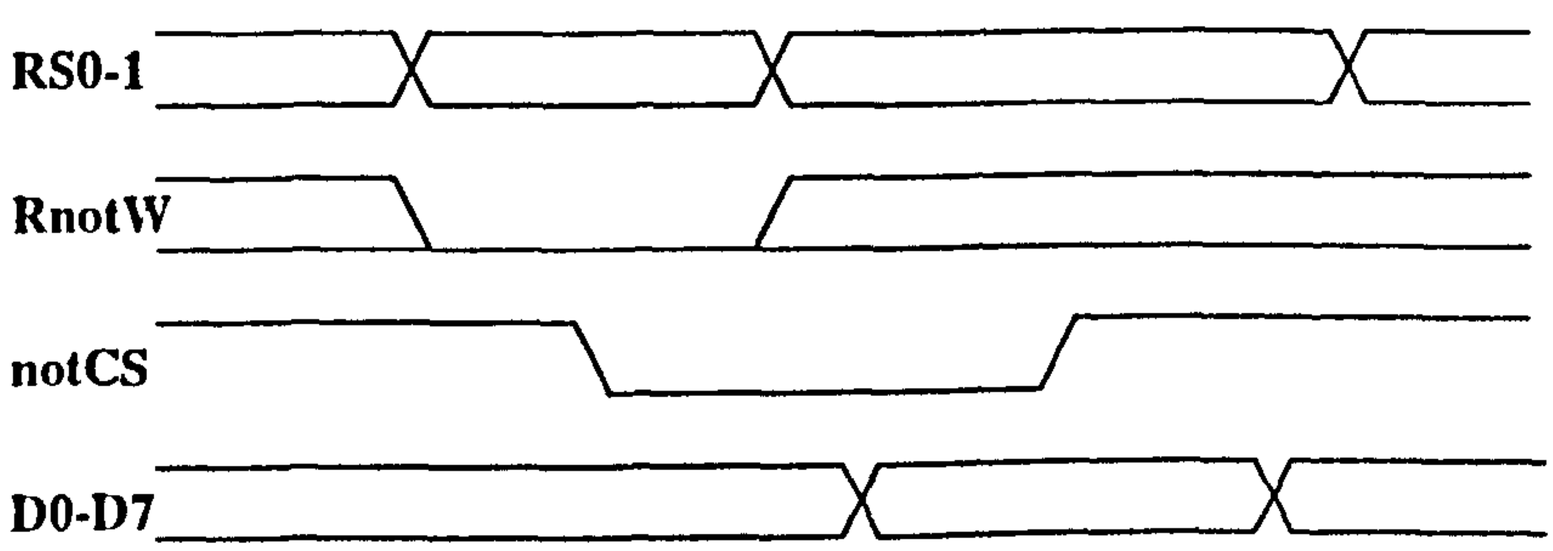


FIGURE 3.18. Timing diagram for write to C012

Signal **NotStatWr** is the D-input of a D-type flip-flop (hence the *.d* extension in the expression for **NotStatWr**) and hence the logic level on the input (which is **NotIOW**) is only transferred to the Q-output on a leading clock edge. A timing diagram for the **NotStatWr** signal in relation to the **IOW***, **RnotW(NotWrite)** and **notCS*** signals is shown in Figure 3.19 on page 85. By ensuring that the chip select signals are only true when both **NotStatWr** and **WriteLink** are true or **ReadLink** is true, the timing of the signals is therefore correct.

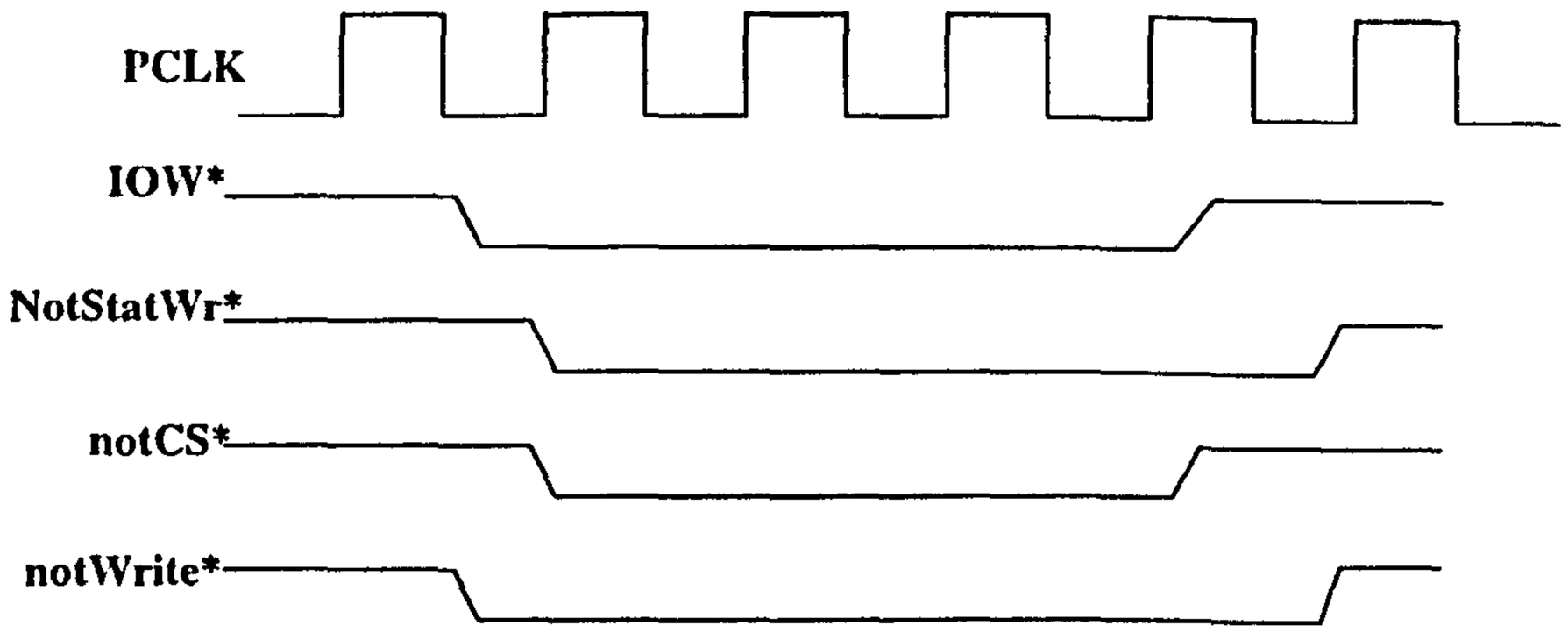


FIGURE 3.19. Timing diagram for NotStatWr signal

A similar delayed signal is not required when reading from the C012s as the **NotWrite*** signal is by default in the logic high state (i.e. when reading from the C012 the **RnotW** signal must be logic high), therefore the signal is always true before the **notCS*** signal becomes logic true.

The chip select signals for the C012s (**notCS0*** and **notCS1***) are therefore logic true when reading from or writing to the link adaptors. The **RnotW** signal is logic true when either the **NotIOW*** or **NotStatWr*** signals are logic true (i.e. when the C012s are being written to).

To enable the '245 buffer which isolates the C012s from the PC bus the **BufEn*** signal is used. This signal is logic true when either of the C012s are selected. The **DIR** signal on the '245 is controlled by the **RnotW** signal, as the direction of the buffer is dependent on whether information is being transmitted or received.

The C012s are put into reset by writing a logic high to the system control address. The **NotIBMReset.d** signals are logic true when the **D0** and **WriteReset** signals are logic true. The second part of the expression for **NotIBMReset** (**NotIBMReset & !WriteReset**) is required in order that the logic level of the signal is maintained after the write cycle from the PC is finished (i.e. on the previous clock edge **NotIBMReset** was true and **WriteReset** is no longer logic true). The C012s are reset when either **NotIBMReset1** or **NotIBMReset0** is true.

The **NotIBMError1** and **NotIBMError0** signals are inputs from the transputer boards which indicate when errors have occurred. These signals are read on **D0** and **D1** and therefore these signals are only enabled as outputs from the P22V10L when reading system information.

3.4 Software requirements

To program the required connections on the crossbar switches the configuration messages must be sent from the host computer via the dual link adapter board to the **ConfigLinkIn** and **ConfigLinkOut** pins on the C004. This is achieved by assembler routines on the host which input and output bytes to the link adapter board. These routines are called from a FORTRAN program which provides a user interface for entering the required connections between the processors.

3.4.1 User interface with a command line

The first version of the user interface written in Microsoft FORTRAN[9], entered the values of the processor and links to be connected via a string of text entered by the user.

i.e. CONNECT PROCESSOR A LINK B TO PROCESSOR C LINK D

To extract the values of the processors and links from the string the positions of the key words (i.e processor and link) were found by using the FORTRAN INDEX function. The values were then extracted from the spaces between the words as substrings (See Figure 3.20).

```
Iproc1 = INDEX(STATEMENT,'processor')
Ilink1 = INDEX (STATEMENT, link')
Ito = INDEX (STATEMENT,'to')
Proc1 = statement (iproc1 + 9: ilink1 -1)
Link1 = statement (ilink1 + 4: ito -1)
```

FIGURE 3.20. FORTRAN code to extract values from string

The first three statements in this section of code locate the position of the first occurrence of the words 'processor', 'link' and 'to' respectively. The next two

statements extract the substrings in between the words which contain the processor and link numbers.

The initial statement is stored as a character variable and therefore the values extracted from the statement are stored as character variables. In order that the values can be used in further calculations they have to be converted to integer variables. This is achieved by the subroutine INTEG (See Figure 3.21).

```
      SUBROUTINE INTEG(D,DUMR,X)
      CHARACTER*10 DUMR
      INTEGER D
      INTEGER*2 X(64)
C     CONVERT CHARACTERS INTO INTEGERS
      READ(DUMR,'(I10)') X(D)
      END
```

FIGURE 3.21. Subroutine INTEG

This routine uses internal files to transfer the character variable **DUMR** into the integer variable **X**.

Since the program depends on finding the key words in the statement, if there are any spelling mistakes in the statement, it has to be re-entered. Once the processor and link numbers are established their values are stored in four separate arrays (i.e. two for the processor numbers and two for the link numbers).

3.4.2 Graphical user interface

Entering the processor and link numbers via a string of text is quite cumbersome and it was therefore decided to create a graphical interface using the Microsoft FORTRAN graphics library[10]. The graphical interface takes the form of 32 boxes on screen to the represent the 32 processors in the switch network. Smaller boxes within the processors represent the links. Figure 7 and Figure 8 on page 307 in Appendix D show the graphical interface.

To make a connection between two processors a line (or series of lines) is drawn between the processors using the mouse. If the user tries to make a connection not on a processor, or on a link that is not allowed then the line will not be drawn. Once a

connection has been established the processor and link numbers are stored in arrays (NP1, NP2, NL1, NL2).

The boxes which represent the processors and links were drawn using the Microsoft FORTRAN "rectangle" and "line" routines. To control the mouse the public domain package "Mouse Driver Interface Package for Turbo C and Microsoft Fortran" was utilised. The routine GETMOUSECURSOR position in this package was used to find the physical coordinates of the mouse and the state of the mouse buttons. The lines to represent the connections were drawn using the "line" routine

The main routine in the graphical interface is the routine PRESSMOUSE which establishes where the mouse button has been pressed, draws the appropriate lines, and then stores the processor and link numbers in arrays once a connection has been completed. It can be divided into two parts: the first part considers the situation when the mouse button has been pressed and second part considers the situation when the mouse button has been released (See Figure 3.22 on page 89).

When a mouse button is pressed at the start of making a connection between two processors, the start position is set to the position of the mouse (it is actually altered so that the start of the line is positioned in the centre of the link box). If the mouse is pressed in the middle of a connection then the start position for the line is set to the end of the previous line. This is also the case when the previous line has been disallowed.

Before actually drawing a line the graphicsmode is set to XOR. This allows a line to be deleted by drawing over the line with the same colour and also leaves any background to the line intact. The purpose of this is so that while the mouse is depressed a line is drawn constantly between the start position and the current mouse position (i.e. the mouse cursor can be moved and the line will follow it). This is achieved by deleting the previous line before drawing the next line if the mouse has moved.

```
Call routine GETMOUSECURSORPOSITION
Check whether mouse pressed on allowed processor and link
 If mouse pressed on allowed position and not on finish
     If not at start of a connection then
          set start point for line to end of previous line
     else if last line was a disallowed connection then
          set start point for line to end of previous line
     else
          set start point for line to mouse position
    set writestyle to XOR

    While mouse button is pressed down
          store previous position of mouse
          find new position of mouse
          if mouse has moved
             draw over previous line
             draw new line
```

Part 1

```
Call routine GETMOUSECURSORPOSITION
Draw over previous line

Check whether mouse released on allowed processor and link
 If mouse released on allowed position
     draw new line
      increment counter which counts no. of lines used to make a connection
     store start and end points of line and no.of line in an array
If connection complete
     store processor and link numbers in arrays
```

Part 2

FIGURE 3.22. Pseudocode for routine PRESSMOUSE

While the mouse button is depressed the current position of the mouse is determined by the GETMOUSECURSORPOSITION routine. If the mouse has moved position since the previous call to the routine the last line is deleted. A line is then drawn between the start point for the line and the current position of the mouse.

When the mouse is released a call is again made to the subroutine GETMOUSECURSOR position to obtain the new position of the mouse. The last line drawn is then erased. If the mouse is released on an allowed position then the new line is drawn. The position and number of the line is stored in an array in order that the connection can be deleted. To delete a connection the user double clicks on either end of the connection.

Once a completed connection is made the processor and link numbers of the connection are stored in arrays. When the user has completed all the connections he/

she wants he/she clicks on the "Finish" box on the screen. The program then makes the required connections on the C004s.

The FORTRAN code for both the command line interface and the graphical user interface is shown in Appendix A.

3.4.3 Programming the C004s

To establish which connections are required on the C004s it is necessary to construct tables which contain the connections from the processors to the switches. Two tables (one for each C004) are therefore created which are arranged in the following way:-

Link No.	Processor No.	LinkIn on C004	Link No.	Processor No.	Link Out on C004

The pseudocode for the subroutine CONNECTIONS is illustrated in Figure 3.23. This routine establishes what connections are required on the C004s. This achieved by considering in turn each processor and link number used in a connection, and then scanning the tables containing the connections to the C004s looking for match. Once a match has been found the values of the links used on the C004s are stored in arrays.

```
For each processor number and link number in a connection
    For each row in each table containing connections to C004s
        If the processor number and link numbers match the values in columns 1 & 2 of table
            store value of LinkIn on C004 in array
        If the processor and link numbers match the values in columns 4 & 5 of table
            store value of LinkOut on C004 in array
```

FIGURE 3.23. Pseudocode for subroutine CONNECTIONS

Before any connections are actually made on the C004s the dual link adapter board which sends the messages to the C004s must be reset. The is achieved by using the assembler routine 'RUN' which is called from the main FORTRAN program (See Figure 3.24 overleaf). The assembler was written in Microsoft Macro Assembler 5.0[11].

This routine just sends a specified value to a specified address and does not by itself reset the link adapters. To achieve this the routine is called twice: the first time a logic 1 is sent to the reset address and the second time a logic 0 is sent to this address.

The .MODEL directive at the start of the assembler source code defines the memory model used by the program. In this case it is HUGE which means that code and data can be greater than 64K (the size of a segment). This matches the memory model used by the FORTRAN compiler.

The .CODE directive marks the start of a code segment. The PUBLIC directive declares a symbol (in this case RUN) public in order that it can be accessed by other routines.

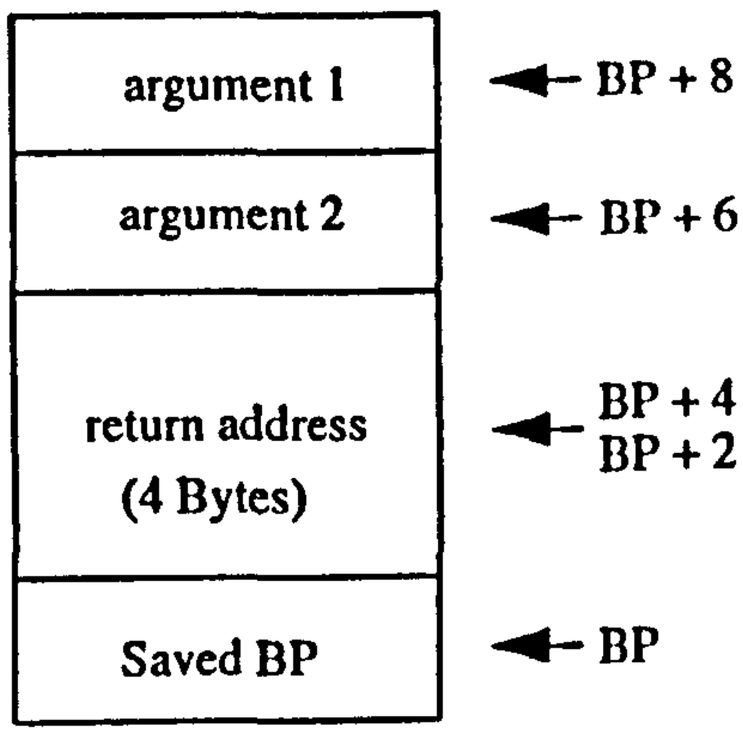
```
.MODEL          HUGE
.CODE
                PUBLIC _run
_run            PROC FAR
                push bp         ;save old bp
                mov bp,sp       ;set stack frame pointer

                mov dx,[bp+8]   ;load address of board
                mov al,[bp+6]   ;load byte to be output
                out dx,al       ;output byte

                pop bp
                ret
_run            ENDP
                END
```

FIGURE 3.24. Assembler routine RUN

When sending values to an assembler program from FORTRAN, the stack is loaded in the following manner:-



FORTRAN pushes values onto the stack in the order they appear in the call to the assembler routine so therefore argument 1 is higher in memory than argument 2.

Immediately after the routine is called from FORTRAN the stack pointer(**sp**) points to the return address. The statement **push bp** decrements **sp** and pushes the value of the base pointer (**bp**) onto the stack. At this point **sp** points to the saved value for **bp**. The statement **mov bp,sp** therefore moves the value of **sp** into **bp** and therefore **bp** points to the saved value of **bp**. This is in order that **bp** can be used to point to the base of a frame of reference within the stack.

When calling the RUN routine from FORTRAN, argument 2 is the byte to be output and argument 1 is the address at which the byte is output. Initially a binary 1 (reset) is sent to both link adapters and then a binary 0 (run). Obviously separate calls to the routine are required for each link adapter as they have different addresses.

The statement **mov dx, [bp + 8]** moves the address of the link adapter into the register **dx.** The command **mov al, [bp + 6]** moves the output byte (1 or 0) into register **al**. The output byte is then written to the link adapter address by the command **out dx,al.**

Before returning control to the FORTRAN program the value of **bp** has to be restored by the statement **pop bp**. The **RET** command pops the return address off the stack and returns control to the FORTRAN program. The **ENDP** directive marks the end of the procedure.

Once the link adapter board has been reset in this way it is ready to receive messages for the C004s. Table 3.1. on page 67 shows that before making a connection on a C004 a command byte [4] must be sent to reset the switch. To program a connection a command byte [0] is sent followed by the numbers of the input and output on the C004. The FORTRAN code which achieves this is in Figure 3.25 on page 93.

The addresses of the link adapters are stored in the array LKAD and the values of the C004 inputs and outputs are stored in the arrays C40IN, C40OUT, C41IN and C41OUT.

```
CALL LINKOUT (4, LKAD(1))
CALL LINKOUT (4, LKAD (2))

DO 30 N = 1,LINKNO
     CALL LINKOUT (0,LKAD(1))
     CALL LINKOUT (C40IN(N),LKAD(1))
     CALL LINKOUT (C40OUT(N),LKAD(1))
     CALL LINKOUT (3,LKAD(1))
     CALL LINKOUT (0,LKAD(2))
     CALL LINKOUT (C41IN(N),LKAD(2))
     CALL LINKOUT (C41OUT(N),LKAD(2))
     CALL LINKOUT (3,LKAD(2))
30 CONTINUE
```

FIGURE 3.25. FORTRAN *code to make connections on C004s*

The subroutine LINKOUT is an 8086 assembler routine similar to RUN which outputs bytes to the link adapters (See Figure 3.26 on page 94). Before transmitting a byte the output status register is checked to ensure the link adapter is ready to receive a byte. The address of the output status register is loaded into register **dx** by the statements **mov dx, [bp + 8]** and **add dx, 03H**. Table 3.3. on page 72 shows that the output status register is 03H above the base address of the link adapter board.

When a link adapter is ready to receive a byte, the first bit of the output status register is set to logic 1. This is checked by first loading the contents of the status register into **dx (in al,dx)** and then masking off the top 7 bits of the byte (statement **and al,01H**). If the first bit is not logic 1 **(cmp al,01H)** then the program loops round **(jne loop_1)**.

Once the link adapter is ready to receive a byte the address of the output data register is loaded into **dx (sub dx,02H)**. The data to be transmitted is then sent in a similar way as in routine RUN.

Using the routine LINKOUT the switches are programmed to make the required connections between the processors using the C004s. It is possible to check that the connections have been made successfully by interrogating the outputs on the C004. This is achieved by sending a command byte [2] to the C004 followed by the number

of the output to be interrogated. The C004 returns the value of the input that the output is connected to.

```
.MODEL          HUGE
.CODE
                PUBLIC _linkout
_linkout        PROC FAR
                push bp         ;save old bp
                mov bp,sp       ;set stack frame pointer

                mov dx,[bp+8]   ;load link adaptor base address
                add dx,03H      ;get address ofinput_status register

loop_1:         in al,dx        ;read value of status register
                and al,1        ;look at first bit
                cmp al,01H      ;see if equal to 1
                jne loop_1      ;loop round if not equal

                sub dx,02H      ;find input address
                mov al,[bp+6]   ;load byte to be output
                out dx,al       ;output the byte in al

                pop bp
                ret
linkout         ENDP
                END
```

FIGURE 3.26. Assembler routine LinkOut

The assembler routine LINKOUT is used to send the byte [2] and the output number. To receive bytes back from the C004 an assembler function called LINKIN was written (See Figure 3.27 on page 95).

This function is similar to LINKOUT. The major difference is that in the case of LINKIN it is the input status register that is polled instead of the output status register as with LINKOUT. The incoming data is read from the input data register which is at the base address of the link adapter. The byte is returned to the FORTRAN routine in register **ax**.

```
.MODEL          HUGE
.CODE
                PUBLIC _linkin
_linkin         PROC FAR
                push bp         ;save old bp
                mov bp,sp       ;set stack frame pointer

                mov dx,[bp+6]   ;load link adaptor base address
                add dx,02H      ;get address of input_status register

loop_1:         in al,dx        ;read value of status register
                and al,1        ;look at first bit
                cmp al,01H      ;see if equal to 1
                jne loop_1      ;loop round if not equal

                sub dx,02H      ;find input address

                in al,dx        ;read input register
                xor ah,ah       ;load 0 into ah

                pop bp
                ret
                _linkin
                ENDP
                END
```

FIGURE 3.27. Assembler routine LinkIn

The main FORTRAN program offers two ways to test connections on the C004s. In the first method a particular output can be interrogated by entering the number of the output (see Figure 3.28 on page 96). The second method interrogates all the outputs on the C004s and then prints out the results in form illustrated in Figure 3.29 on page 96.

```
WRITE (*,*) 'ENTER NUMBER OF OUTPUT',OUTPUT
CALL LINKOUT (2,LKAD(2))
IN = LINKIN(LKAD(2))
IN = IN - 128
IF ((IN.GT.0).AND.(IN.LT.32))
 WRITE(*,*) 'THIS OUTPUT IS CONNECTED TO INPUT',IN
ELSE
 WRITE(*,*) 'THIS OUTPUT IS NOT CONNECTED'
END IF
```

FIGURE 3.28. FORTRAN code to interrogate an output

```
Processor A LinkIn B is connected to Processor C LinkOut D
```

FIGURE 3.29. Format of statement showing connections

3.5 Conclusions

A basic system has been presented which allows the topology of a multiprocessor network to be altered before computation. This provides a more flexible and efficient set-up than a fixed topology system as some parallel algorithms are more suitable to one particular configuration than another.

The final version of the software to program the switches allowed for eight links on the nodes, although of course in the case of transputers this was not implemented. Some other processors such as the Texas Instruments C40 have more than four links and therefore this allows the graphical interface to be ported easily to other systems.

Since the switch board was built on a wire-wrap board and there are 64 connections to each C004 the board is not very reliable. In retrospect it would probably have been more efficient to get a PCB (printed circuit board) made of the design.

This prototype system does however provide a simple method to vary the topology of a transputer network. It would allow a particular algorithm to be tested on different topologies. As stated previously the same methodology could be applied to other processors such as the C40 and also with alternative crossbar switches.

References

[1] Gaughan, Patrick T. and Yalamanchili, Sudhakar. A daptive Routing Protocols for Hypercube Interconnection Networks. *Computer*, May 1993, pp 12-22

[2] W.J. Dally and C.L. Sietz, Deadlock Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Computers*, Vol. C-36, No. 5, May 1987. pp 547-553

[3] Sriskanthan *et al.* An adaptive switching architecture for multiprocessor networks. *Microprocessors and Microsystems*, Vol. 18, No. 6, July/August 1994

[4] INMOS® Limited. *The Transputer Databook.* Third Edition 1992

[5] Bowler *et al.* An Introduction to Occam 2 programming. Chartwell-Bratt (Publishing and Training) Ltd, 1987, ISBN 0 86238 137 1

[6] Eggebrecht, Lewis C. Interfacing to the IBM® Personal Computer. Howard W. Sams and Company, 1990, ISBN 0 672 22722 3

[7] Texas Instruments. TTL Data Book Volume 1, 1989, ISBN 3 88078 078 1

[8] Atmel Corporation. CMOS Integrated Circuit Data Book, 1991-92. pp 8-19, 8-33

[9] Microsoft® FORTRAN Reference Manual, 1991

[10] Microsoft® FORTRAN Advanced Topics, 1991

[11] Microsoft® Macro Assembler 5.0 Programmer's Guide

Chapter 4

A Circuit Switched Network for Inmos OS Links

An efficient circuit switching mechanism allowing dynamic-on-demand allocation of physical links between processing nodes is described in this chapter. This cost-effective, memory-mapped system sends connection requests via an INMOS OSLink to a control processor which programs a crossbar switch. By setting up point-to-point direct physical links between nodes this allows every node to communicate directly with every other node of a parallel computer.

A brief description of the basic paradigm used for providing dynamic-on-demand allocation of physical links is presented first. The various designs considered are then detailed before describing the final design.

4.1 Overview of Dynamic Circuit Switching Systems

Dynamic circuit switching systems[1-4] allow inter-node connections to be established on-demand during program run-time. A dedicated point-point communication link is set-up between the two communicating nodes and is maintained until a message has been transferred completely.

These schemes have the advantage over packet switching methods that no additional software is required on each node to manage the passing of the packets and since the messages do not pass via intermediate nodes no additional buffering facilities are required on the nodes. Under normal circumstances dynamic on-demand circuit switching out performs packet switching[3].

4.1.1 Hardware Configurations for Dynamic Link Switching

To perform inter-node communications on a dynamically reconfigurable architecture an application program must be aware of the changing connective state of the network. This can be achieved by allowing the application itself to control the switching of links between nodes. To accomplish this the nodes send connection requests to a control

processor which then programs the required connection on a crossbar switch (See Figure 4.1).

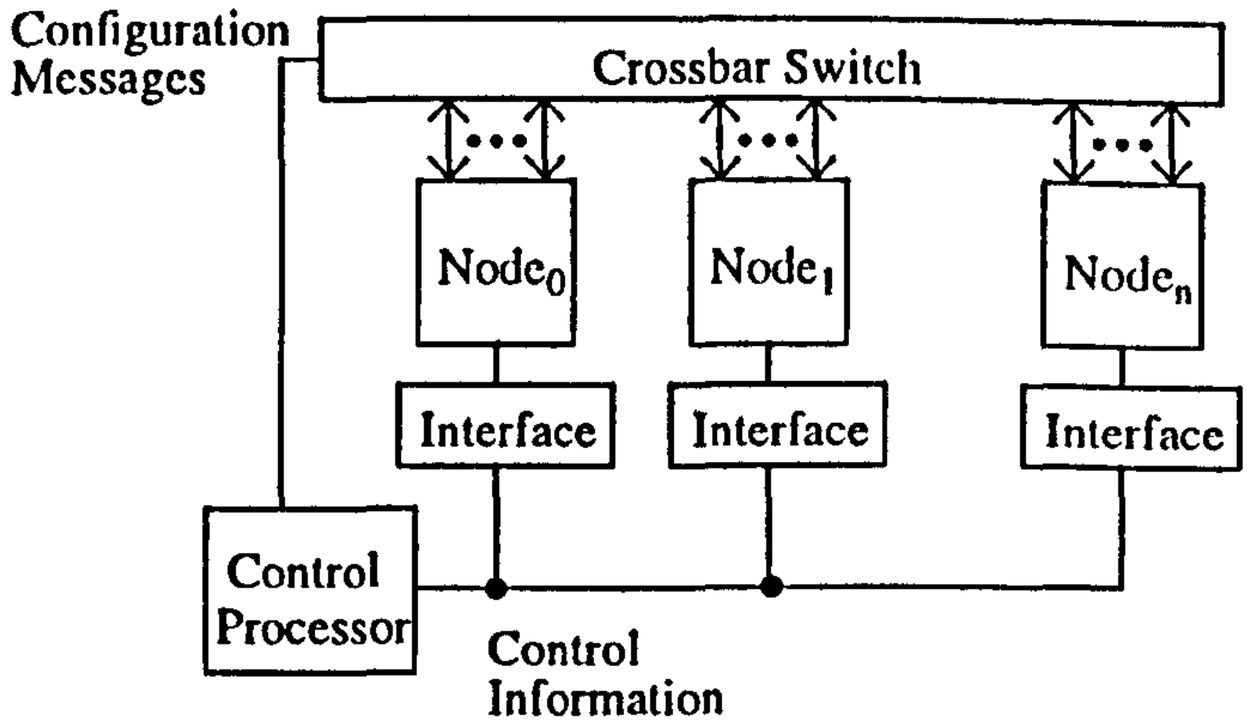


FIGURE 4.1. General structure of a dynamic switching scheme

The interface from the nodes to the control processor must be efficient and fast in order to reduce the message latency (i.e the time interval from when a message is initiated until it actually leaves the node). If the message latency is too long then this will reduce the benefits of using circuit switching over packet switching.

Several different mechanisms have been identified for interfacing the nodes to the control processor and these are described in the following sections.

4.1.1.1 Link-pipeline driven reconfiguration

In this method connection requests are sent via a link pipeline between the working nodes and the control processor (See Figure 4.2 on page 100). This requires that the worker nodes need to manage the passing of the connection requests which will slow down the application processes in the worker nodes. This is not very efficient so therefore this system is rarely used. It does have the advantage however that many commercial systems are already hardwired in a pipeline.

4.1.1.2 Memory-driven reconfiguration

This solution uses a byte-wide parallel bus to connect the local memories of worker nodes with that of the control processor (See Figure 4.3 on page 100). This bus is used for sending connection requests, acknowledges and link releases. The attention of the control processor can be attracted by a worker node by sending a request signal to the

Event input or by setting a flag inspected by the control processor. The control processor is master of the bus.

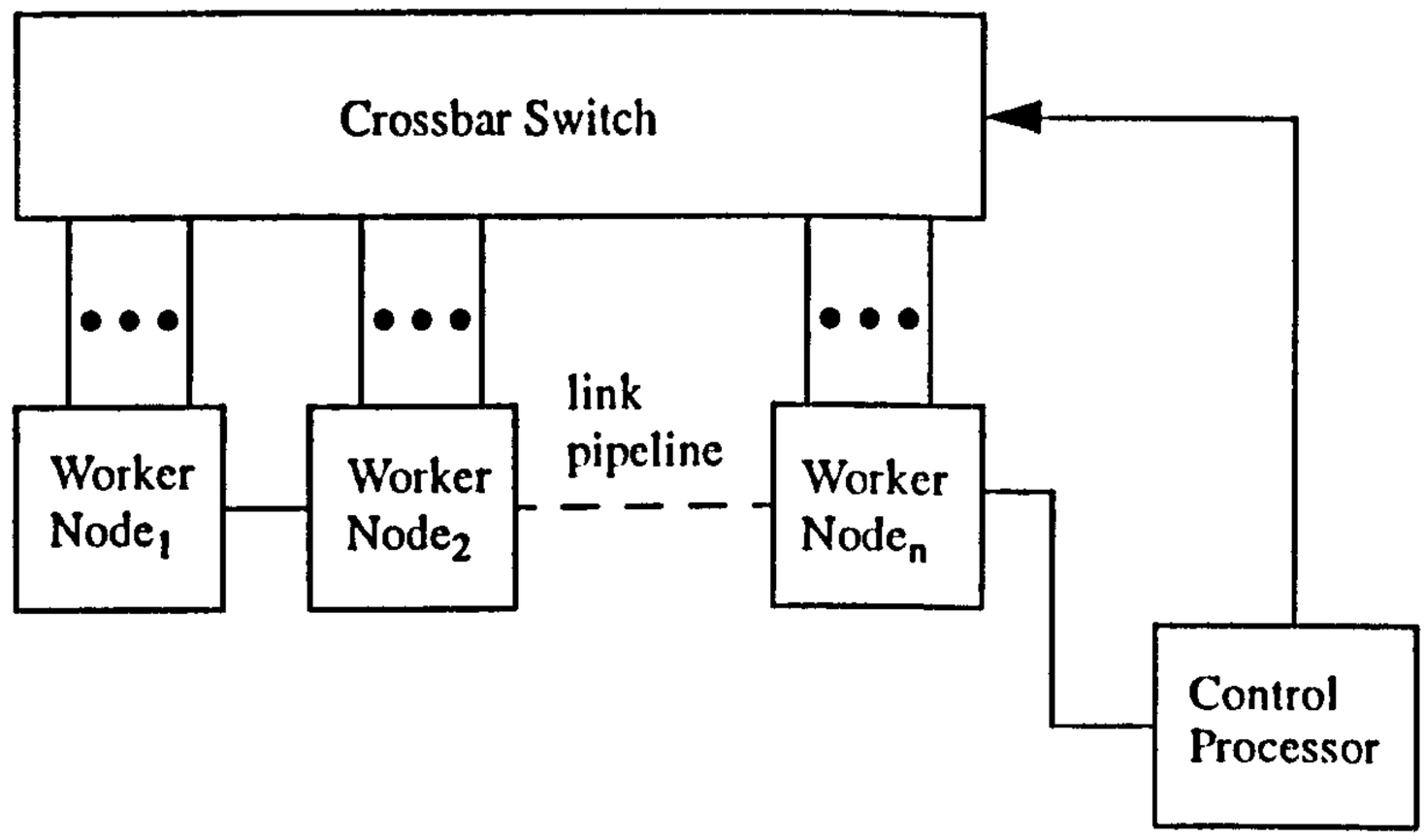


FIGURE 4.2. Link Pipeline Driven Reconfiguration Control

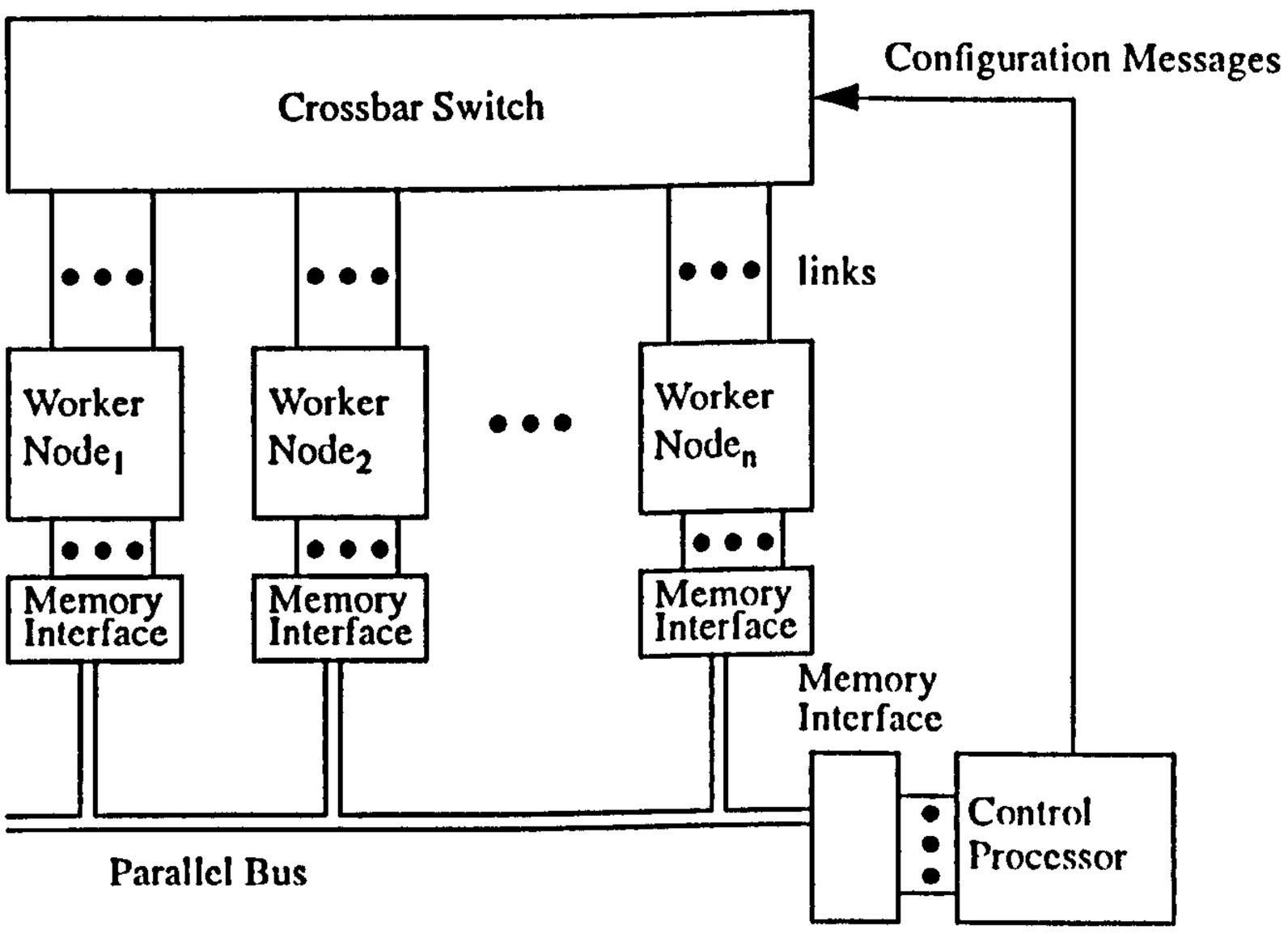


FIGURE 4.3. Memory driven reconfiguration

The control parallel bus is structurally a good solution as it leaves all the links on the nodes free to connect to the crossbar switch allowing the maximum communication bandwidth available from the nodes. However the speed of the bus should be at least as fast as the links on the nodes to achieve greater efficiency than packet switching. This is not the case in existing control bus implementations mainly because they were designed for supervision and not for reconfiguration purposes.

If a system was designed with a fast parallel bus, however, it would be a good solution. The main drawback would be the number of wires.

4.1.1.3 Serial bus driven reconfiguration

This solution uses a serial bus to exchange control information between the worker nodes and control processor (See Figure 4.4). The write access to the bus is controlled by a token which circulates among controllers which interface the node links to the bus. There is no master of the bus and therefore each message includes a destination address.

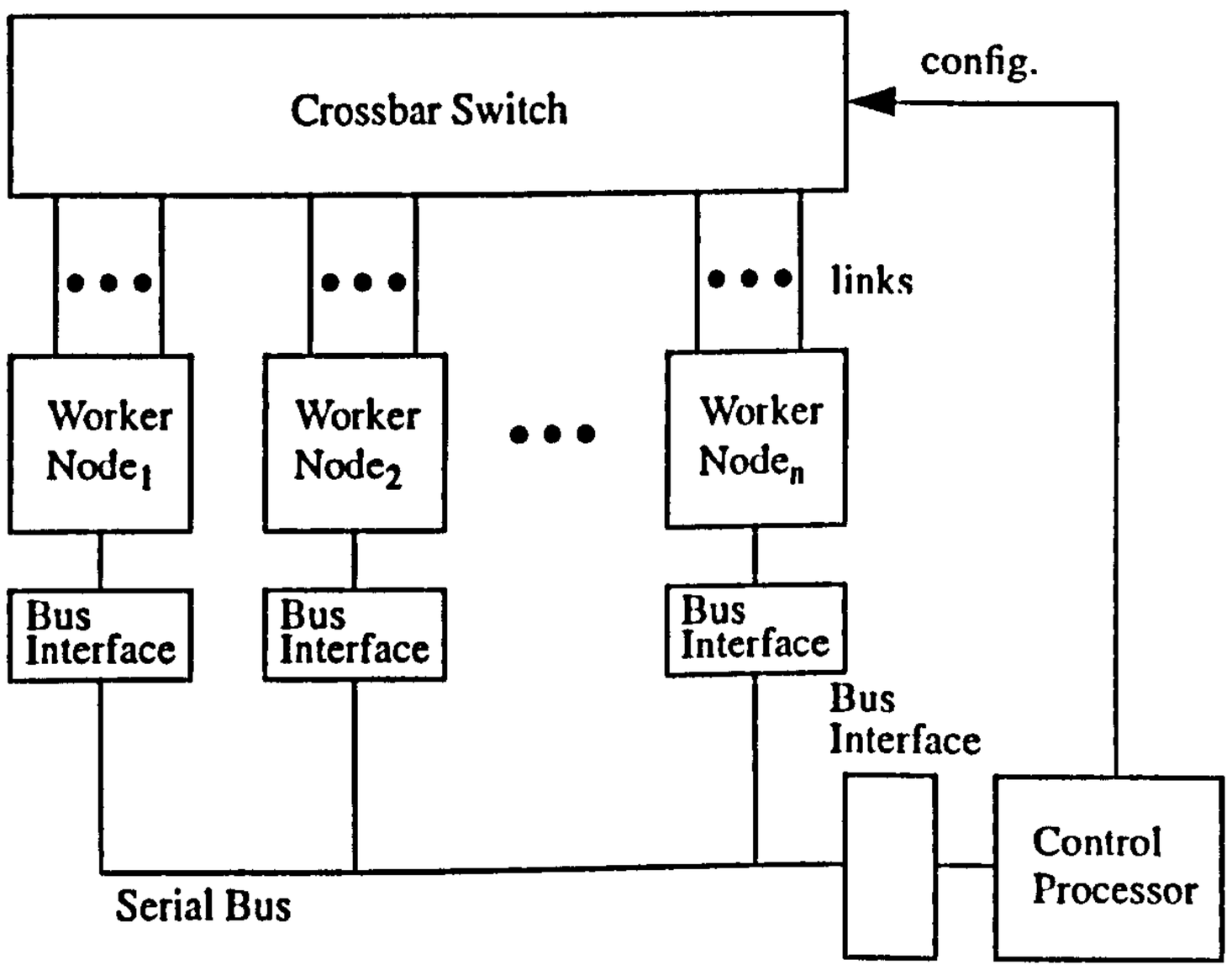


FIGURE 4.4. Serial Bus Driven Reconfiguration Control

Tudruj and Kalinowski[1] have developed a system of this type using a TRANSBUS[6] controller to act an interface between the worker nodes and the control processor. A TRANSBUS is an application specific integrated circuit (ASIC) designed to connect transputers together into a LAN (Local Area Network) (See Figure 4.5 on page 102).

Each transputer is connected to a TRANSBUS controller on one side using one of its standard links leaving only three for interprocessor communication. On the other side, the TRANSBUS is connected to four lines:

- a DATA line to transmit the data bytes
- an ACK line to transmit and receive acknowledges for each transmitted byte
- 2 lines wired as a ring to manage the token-passing

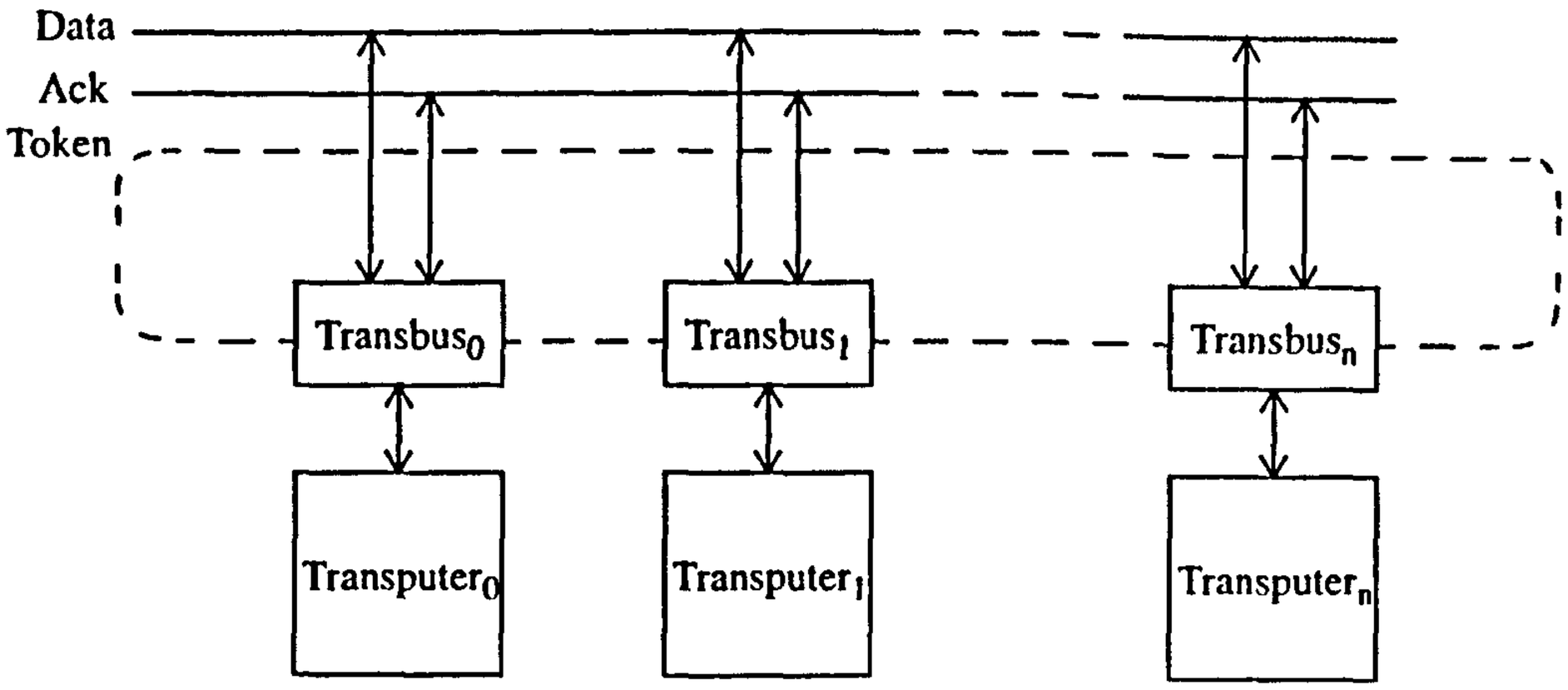


FIGURE 4.5. Interconnecting transputers by the TRANSBUS controller

The format of a message sent via the serial bus is shown below:-

destination address	message length	data

The destination address identifies the receiving transputer and the message length is the total number of bytes of data sent. A destination address of zero signals a broadcast to all the other transputers.

If a transputer wishes to communicate through the bus, it sends the destination address of the message to its TRANSBUS controller. At this point the address is buffered waiting for the token. When the token arrives, the address byte is sent to the DATA line of the bus and the controller waits for acknowledges from all other controllers. The next byte of a message can only be sent after all TRANSBUS controllers on the bus have acknowledged the current byte.

Once an ACK is received from all the other TRANSBUS components an acknowledgement is sent to the sending transputer via its link connection to the TRANSBUS controller. The sending transputer then sends the remainder of the message via the TRANSBUS controller to the receiving transputer.

The destination address is read and decoded by all TRANSBUS controllers on the bus. Only for the addressed transputer (or for all in the case of a broadcast) however is the rest of the message received by the controller and transmitted via the transputer link to the receiving transputer.

Figure 4.6 shows how these TRANSBUS controllers can be used to implement a dynamic switching scheme. The serial bus is used for reconfiguration control, however, to increase the control communication bandwidth three serial control buses working in parallel are employed.

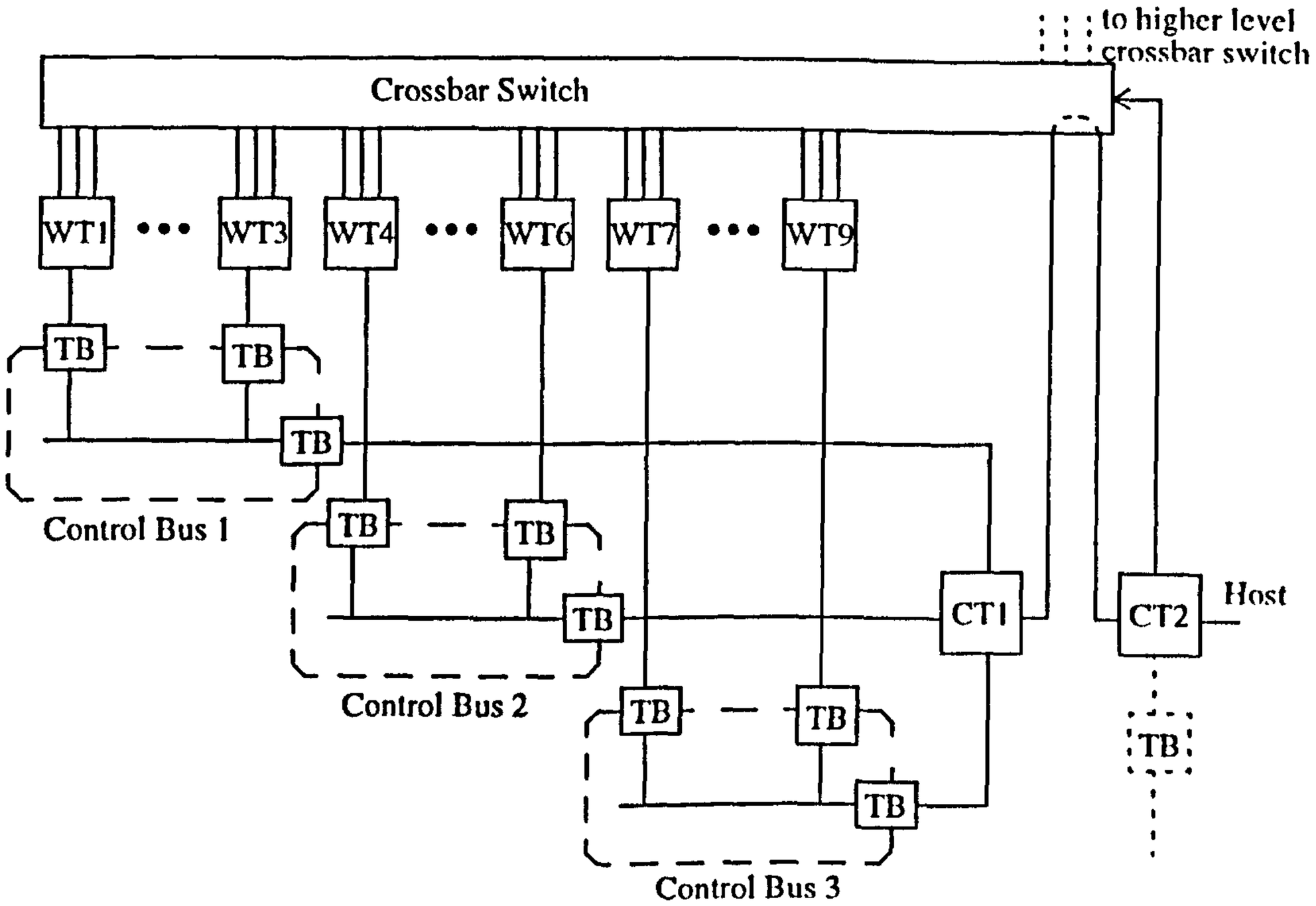


FIGURE 4.6. Structure of a single cluster TRANSBUS system

Three links of each worker transputer are connected to the crossbar switch which is programmed by the control transputer CT2. One link of each worker transputer is connected to the TRANSBUS controller, which provides the interface with the control bus.

The control transputer CT1 sends and receives information from the three control buses. It collects the connection requests, synchronises them and sends acknowledges to worker transputers for the connections established by the CT2 transputer. Besides configuring the crossbar switch CT2 provides working transputers with program loading, collecting computation results and the communication with the host.

This is an efficient and low message latency system. It does however use one of the valuable communication links on the nodes for connection requests. The system would

provide greater communication bandwidth if this could be avoided. Also the TRANSBUS controllers although efficient are not commercially available.

The novel dynamic-on-demand circuit switched network described in this chapter employs some of the same principles as detailed in the previous sections. Connection requests are sent via a memory mapped system to the control processor and a token passing mechanism is used to select nodes for servicing. Before the final design was implemented however two other designs were considered. These are described in the following sections.

4.2 Preliminary Designs

4.2.1 Interrupt Driven Architecture

The hardware design of this system is shown in Figure 4.7 on page 105. Here several 16-1 multiplexers are used to interface the worker nodes to the control processor. The address pins on the multiplexers (i.e. the pins which select the input that is selected as the output of the multiplexer) are connected to a counter which effectively counts through the nodes and if a node requires service then the counter stops and the control processor is interrupted.

The design shown in Figure 4.7 is for a 32 node system. The number of nodes can be increased by using more multiplexers.

There are four sets of two multiplexers:-

- the Flag_Out multiplexers
- the LinkIn multiplexers
- the LinkOut multiplexers
- the Interrupt Acknowledge multiplexers

Each set can provide a direct connection from the control processor to a node. The nodes are numbered from 0-31 and the counter effectively addresses each node one at a time (i.e. direct connections are established simultaneously to the Flag_Out, Int Ack, LinkIn and LinkOut pins on a node) and if a node is requesting service the counter is stopped.

The outputs of the counter are connected to the address inputs on the multiplexers. These address inputs select which input of the sixteen is connected to the output on the multiplexer. By connecting D0-D3 on the counter to D0-D3 on the multiplexer this effectively cycles through all the inputs on the multiplexer. To select which multiplexer of the two in a set is enabled a 2 line to 4 line encoder (See Figure 4.8 on page 106) enables one of the multiplexers at a time.

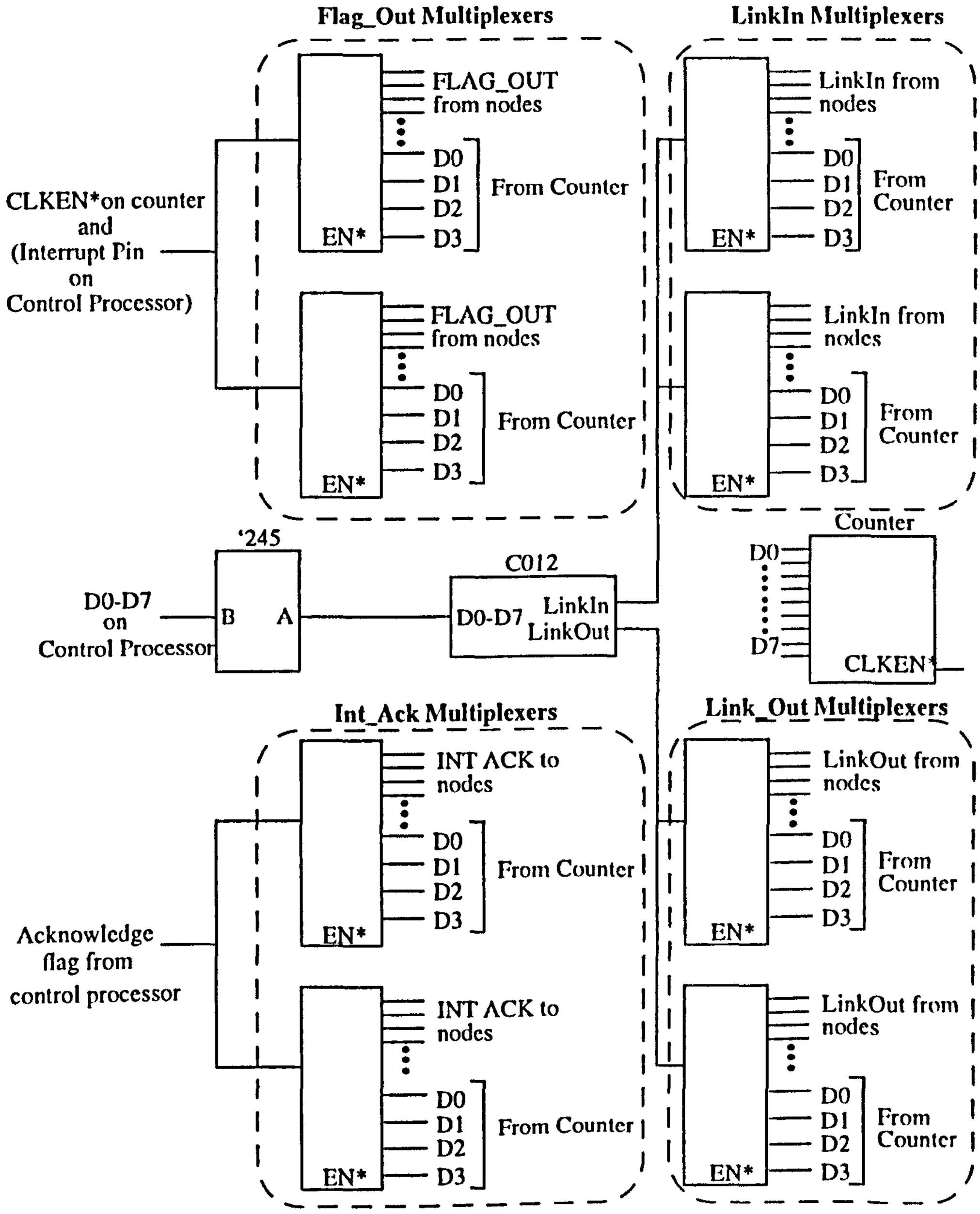
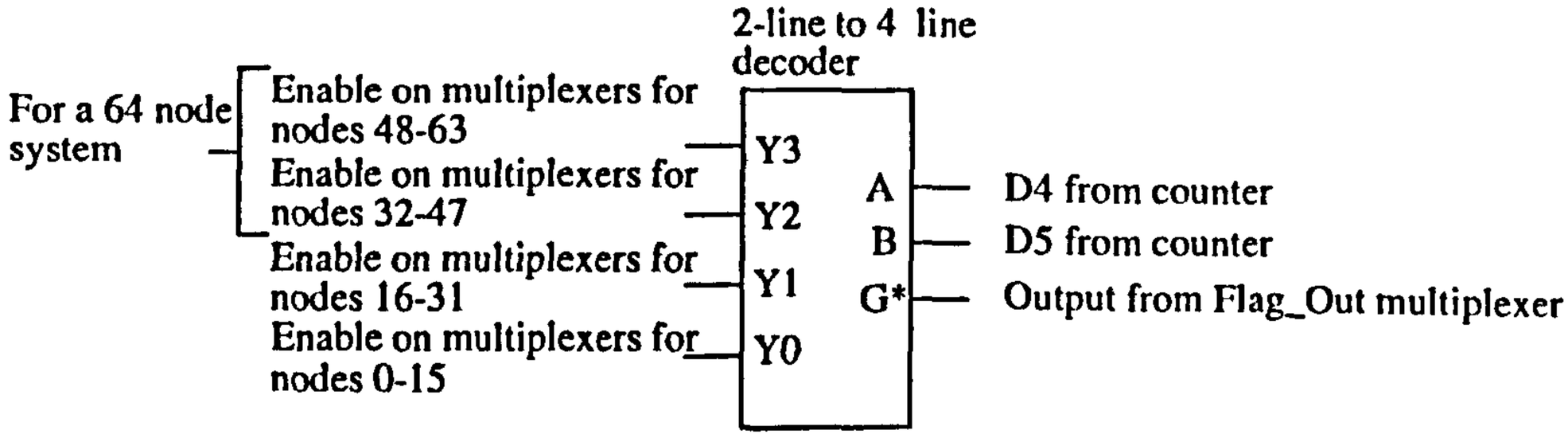


FIGURE 4.7. Interrupt Driven Design

When a node (the source node) wants to send a connection request to the control processor it will set it's Flag_Out pin logic true. The outputs of the Flag_Out multiplexers are connected to CLKEN* of the counter (enables the counter) and the interrupt pin of the control processor. Therefore when a node is being addressed by the counter and the Flag_Out pin is logic true (+5V) this will stop the counter (i.e the CLKEN* pin will be logic false) and interrupt the control processor.



Function Table

G*	B	A	Y0	Y1	Y2	Y3
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
H	H	H	H	H	H	L

FIGURE 4.8. Connections on 2-line to 4-line decoder

Once the control processor has been interrupted it will send an acknowledge to the source node to prompt it to send its connection request. This acknowledge is sent via the Interrupt Acknowledge multiplexers. The interrupt acknowledge line could be connected to the FLAG_IN input on the requesting node.

The node sends its connection request via the LinkIn and LinkOut multiplexers. The output from these multiplexers is connected to a C012 which is interfaced to the control processor.

When the control processor has received the connection request it decides whether the connection required is available by looking up a table stored in memory. If available the connection is made and a message sent to the source node indicating this. The source node then transfers data to the destination node via the crossbar switch. The counter is then restarted by the source node by pulling its FLAG_OUT pin logic false.

This design was however rejected mainly because of lack of scalability. As soon as you add more nodes to the system the number of multiplexers requires to be increased and the counter extended. For a large number of processors the system would become impractical. Also the set-up uses one of the links on the nodes to send connection requests making poor use of the total communication bandwidth available from the nodes.

4.2.2 Memory Mapped Architecture using the COM20020 Network Controller

This design uses COM20020 Universal Local Area Network Controllers[7] to interface the nodes to the control processor (See Figure 4.9). It is similar to the design using the TRANSBUS controller described previously in that a token passing mechanism is used to restrict access to the serial bus connected to the control processor. However in this case the network controller is memory mapped to the nodes instead of using one of the links on the nodes to send the connection requests.

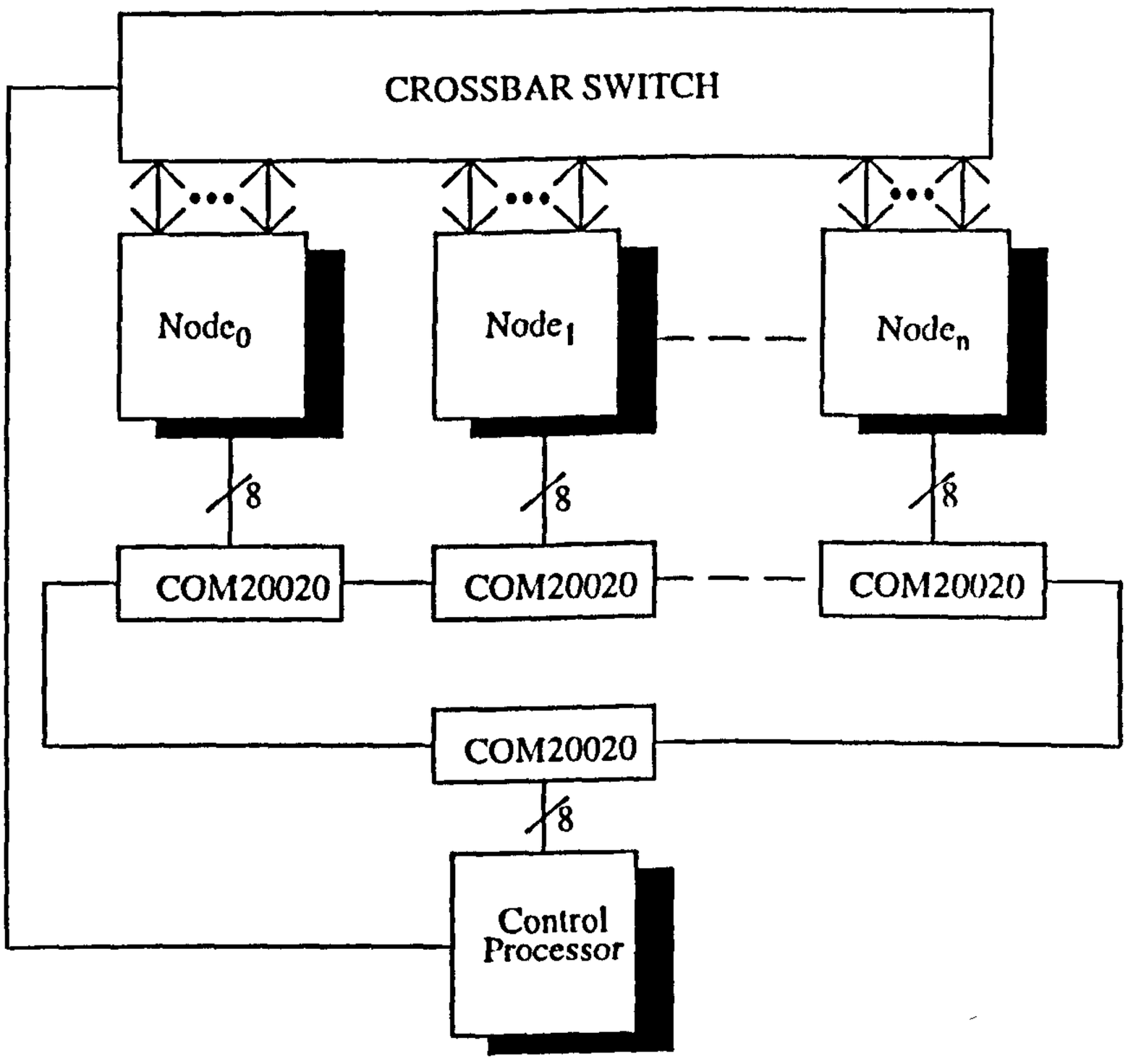


FIGURE 4.9. COM20020 Interface to Control Processor

The COM20020 is a special purpose communications controller for networking microcontrollers and intelligent peripherals using an ARCnet protocol. It is interfaced

to nodes via an 8-bit data bus, an address bus and control bus (See Figure 4.10). Data is transmitted via a serial bus that supports data rates from 156.25 Kbps to 5.0 Mbps.

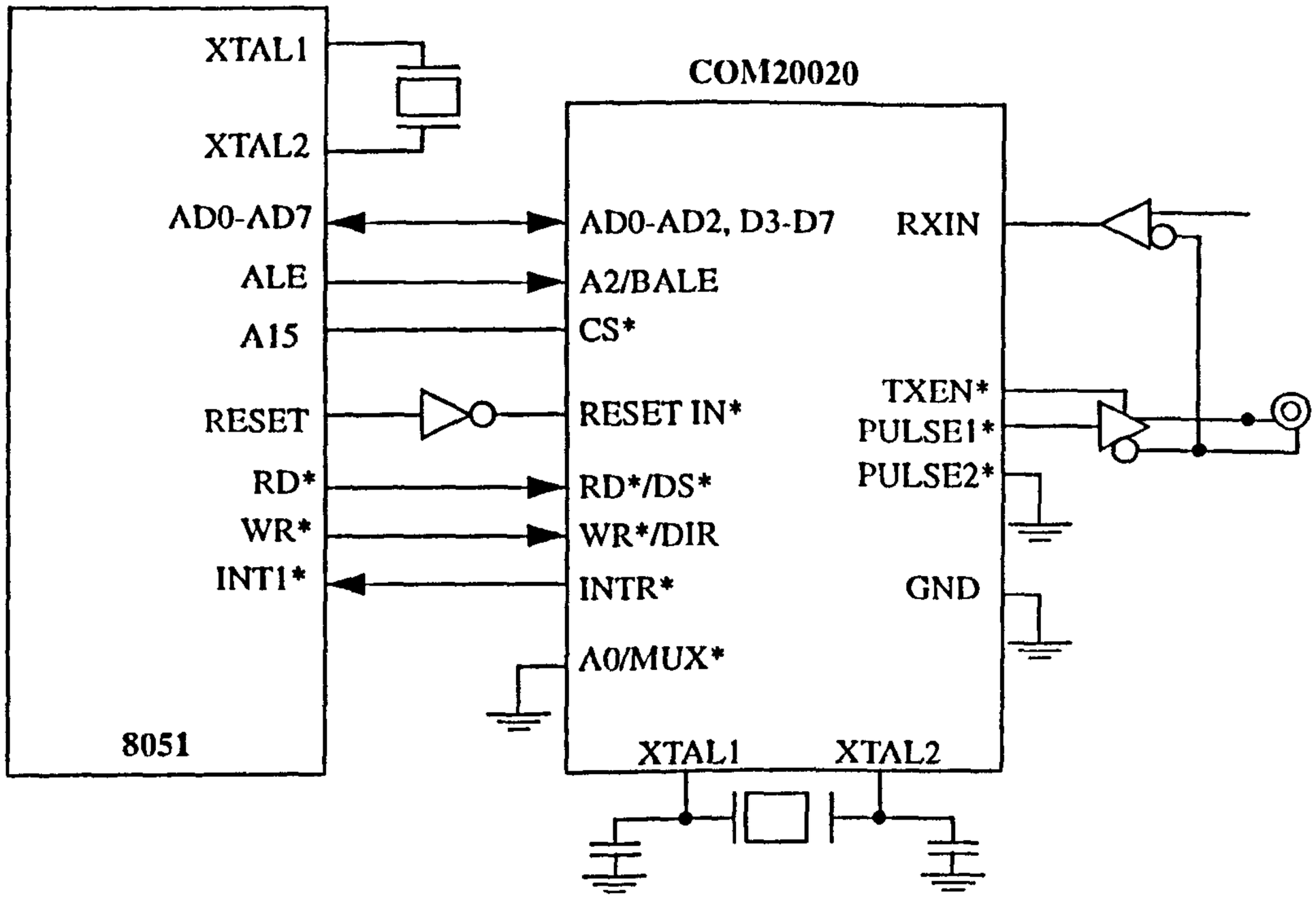


FIGURE 4.10. Multiplexed, 8051 - like bus interface with COM20020

If a node wants to transmit data it simply loads a data packet and its destination ID into the COM20020 RAM buffer, and issues a command to enable the transmitter. When the COM20020 next receives the token it first verifies that the receiving node is ready, and if so it transmits the data packet followed by a 16-bit CRC (cyclical redundancy checksum). If the receiving node is not ready then the token is simply passed on.

The token is passed between the controllers by transmitting an Invitation to Transmit signal from controller to controller. This is given by the following sequence of bits:-

- An ALERT BURST (6 unit intervals of logic 1)
- An EOT (End of Transmission: ASCII code 04H)
- Two (repeated) DID (Destination ID) characters

By interfacing a COM20020 to all the nodes in a network and to a control processor it is possible to send connection requests to the control processor via a memory mapped

system. When a node wants to send a connection a request it simply loads the message into the COM20020 along with the destination ID of the control processor.

Although this architecture leaves all of the communication links on the nodes free for inter-node communication its main drawback is the message latency of the C0M20020 (~100 μs). This does not provide very efficient communication with the control processor. In order to achieve greater performance benefits over packet switching schemes using INMOS OSLinks, a faster link to the control processor is required.

The best features of the previous two designs such as the token passing protocol and the fast Inmos OSLink to the control processor were used in the final design.

4.3 Novel dynamic 'on-demand' circuit switched network

This combines an INMOS OSLink, memory mapping and a token passing mechanism to communicate with the control processor[8].

The same principles as the TRANSBUS system are employed except that several commonly available integrated circuits (ICs) are utilised to interface the nodes with the control processor. Connection requests from the nodes are sent via a memory mapped system which leaves all the links on the nodes free for interprocessor communication. The nodes are not restricted to transputers, the system can be used with any processor which provides an external memory interface.

4.3.1 Basic Procedure

When a node (the source node) wants to communicate with another node (the destination node) via a crossbar switch, it writes its connection request (a three byte packet) into a FIFO (First In First Out Memory), which stores the request until it is honoured. To select nodes for servicing a token passing protocol is used.

The token circulates between the nodes and when a node receives the token and there is a request pending, the request is passed out of the FIFO to the control processor, via an INMOS OSLink. The control processor then decides whether the required connection is available and if so makes the connection. A message indicating success or failure to

make the connection is sent to the source node. In this way the control processor processes connection requests from the nodes in a sequential manner.

4.3.2 Hardware subsystem

A diagram of the hardware for one node is illustrated in Figure 4.11. A node requiring service writes its connection request into the First In First Out Memory (FIFO), which is mapped into the nodes memory address space. This allows the node to continue with other tasks while its connection request remains stored in the FIFO until honoured. When a node receives the token, its request is clocked out of the FIFO into a C011 Link Adapter (explained later). The data is then transferred via the INMOS OSLink to the control processor which programs the crossbar switch. Access to the link is gated by a buffer ('125) which is only enabled when the node has the token and there is a request pending.

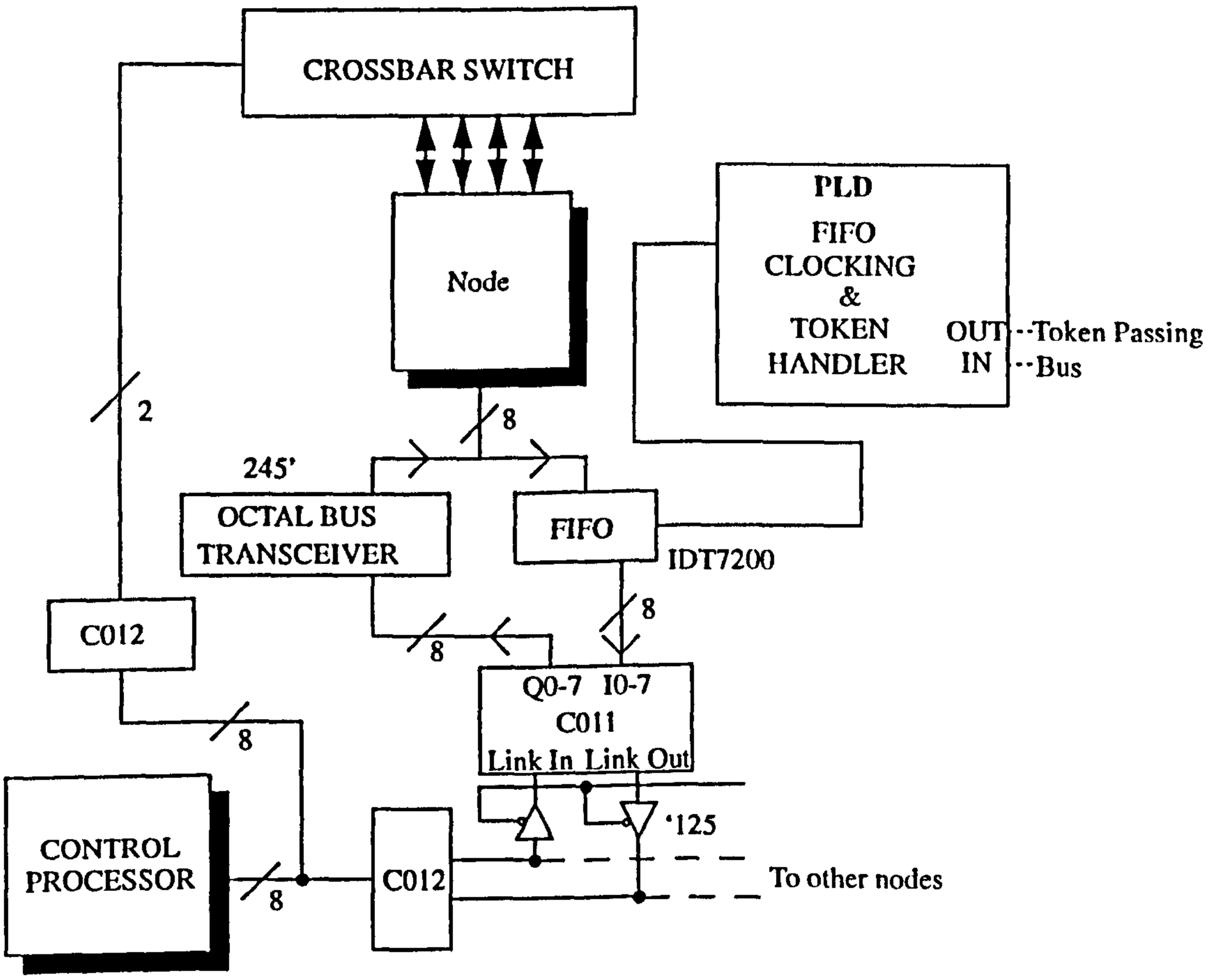


FIGURE 4.11. Dynamic Interconnection Network (1 node)

There are therefore four basic elements to the design:-

- token passing
- fifo clocking
- hardware interface to control processor
- software on the control processor

These will be described individually along with the test hardware/software used to verify the principles

4.3.3 Token Passing

4.3.3.1 State Machines

The token passing is achieved by a finite state machine (SM) implemented in PLDs. A state machine has a set of states and a set of transition rules for moving between the states at each clock edge (the clock is derived externally). The transition rules depend on the both the present state and on the particular combination of input levels present at the next clock edge.

A diagram of a state machine is shown in Figure 4.12 on page 112. The information stored in the memory section, as well as the inputs to the combinatorial logic (I_0, I_1, ..., I_m) is required for proper operation of the circuit. At any given time, the memory is in a state called the *present state* and will advance to a *next state* on a clock pulse as determined by conditions on the excitation lines (Y_0, Y_1, ..., Y_p). The present state of the memory is represented by the state variables (Q_0, Q_1, ..., Q_m). These state variables, along with the inputs (I_0, I_1, ..., I_m), determine the system outputs (O_0, O_1, ..., 0_m)

Not all state machines have input and output variables as described. Sometimes the state variables are also the outputs (i.e. the state variables bypass the combinatorial logic).

Since PLDs contain combinatorial logic and memory they are ideal for implementing state machines. The programming language CUPL (as explained in Chapter 2) contains special instructions for state machine design. These will be explained in the course of describing the token passing using a finite state machine.

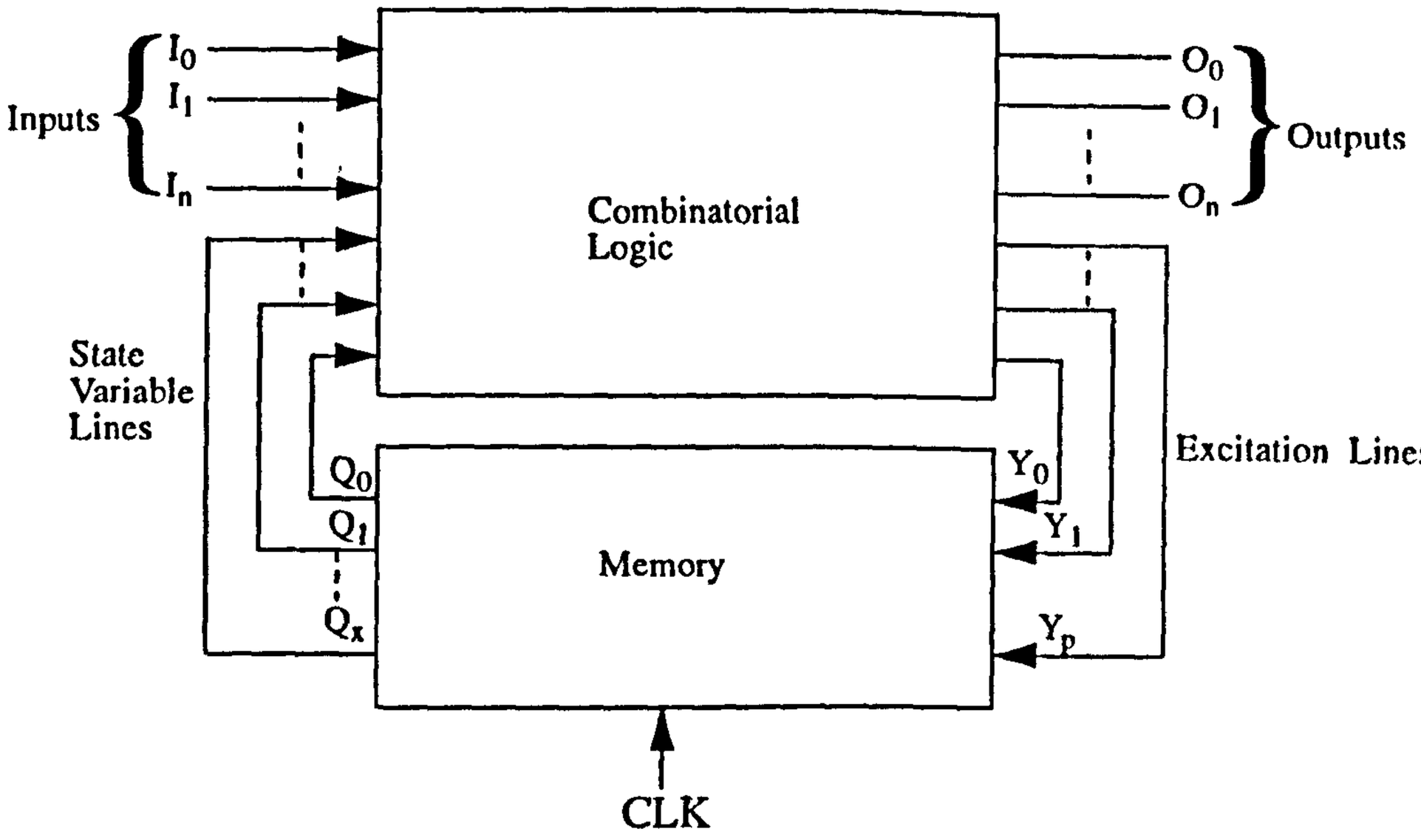


FIGURE 4.12. State Machine

4.3.3.2 Token passing using a finite state machine implemented in PLDs

The token is effectively a bit (binary '1') which passes between the PLDs and each node has a PLD associated with it (See Figure 4.13). The token passing bus consists of two lines: one which passes the token and one which acknowledges the passing of the token. The state variables are **TokenOut** and **AckOut**, and the inputs are **TokenIn**, **AckIn**, and **HoldToken**. The clock used for the token passing is 8MHz.

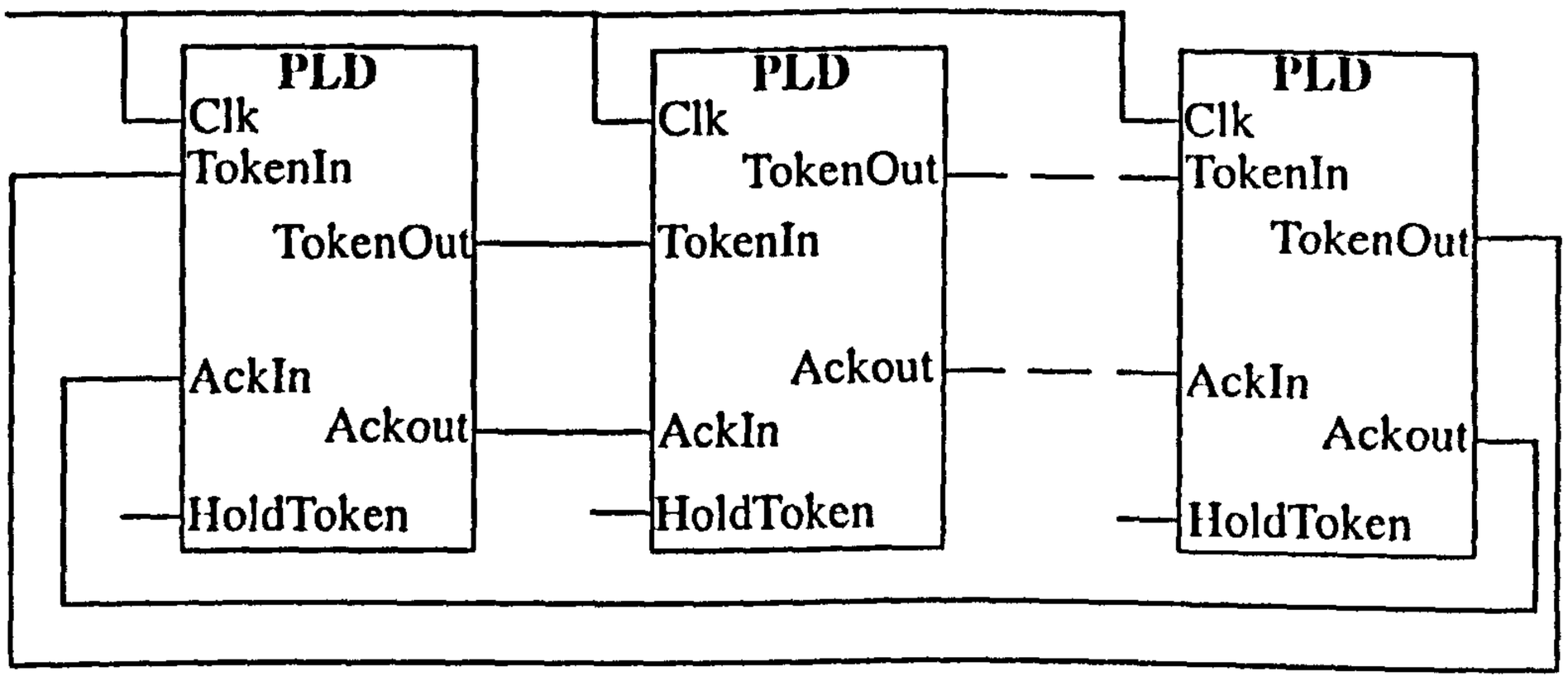


FIGURE 4.13. Token Passing

If a node receives the token and there is a request packet in the FIFO, then the token must be retained until the FIFO has been emptied and the node no longer requires the

token. This is achieved by the **HoldToken** signal, which is generated by using a combination of the Empty_Flag* (**EF***) signal from the FIFO (logic false (+5V) when the FIFO contains bytes), and a D-type flip-flop (See Figure 4.14).

The Q-output of the flip-flop is used as the **HoldToken** signal. The **EF*** signal clocks the flip-flop: therefore when **EF*** becomes logic false (+5V) indicating data is in the FIFO, the level at the D-input (logic high (+5V)) is transferred to the Q-output of the flip-flop. To release the token, the node pulls the **CLR*** signal on the flip-flop logic low (0V) for a short period, which clears the Q-output back to logic low releasing the token.

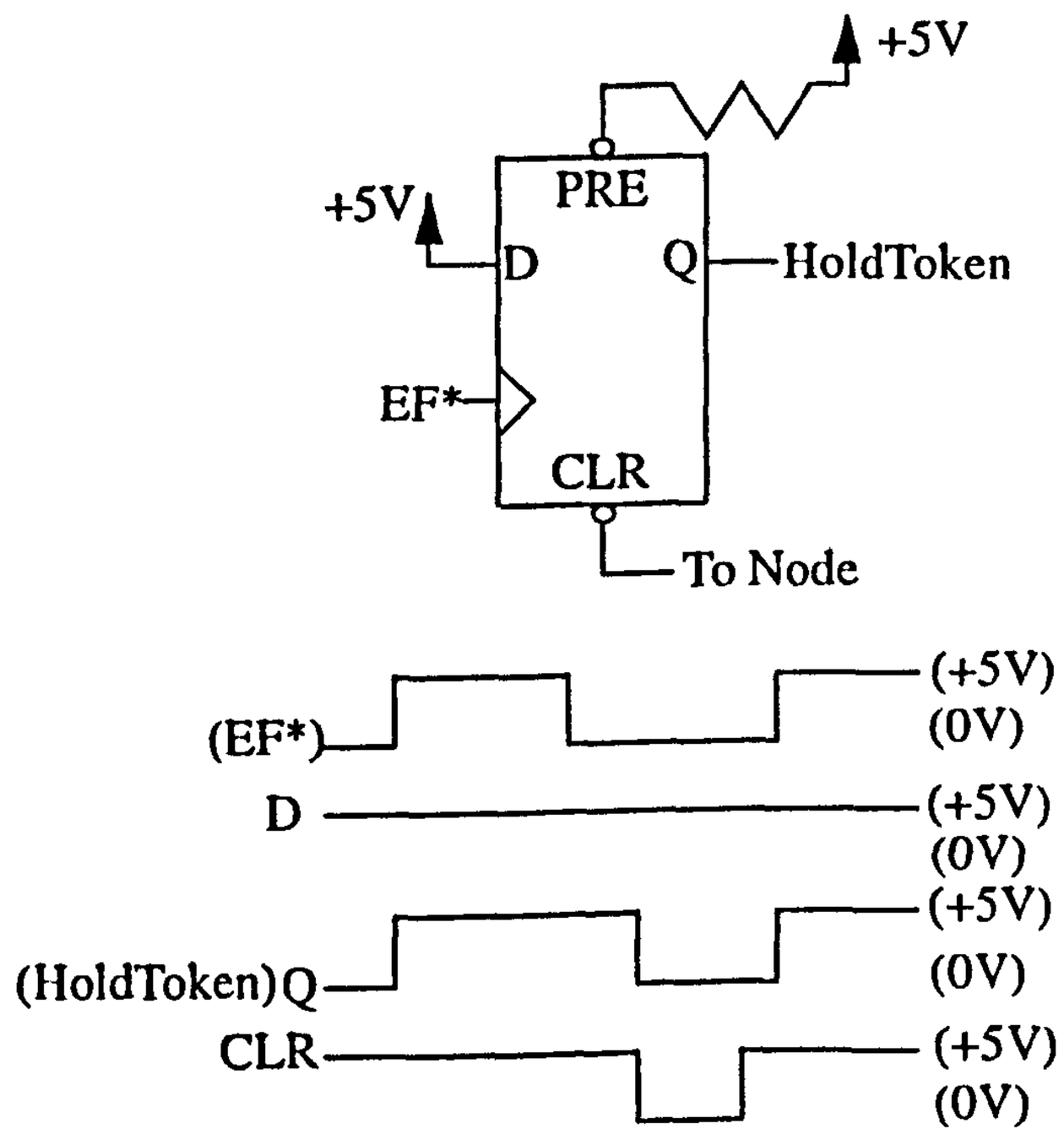


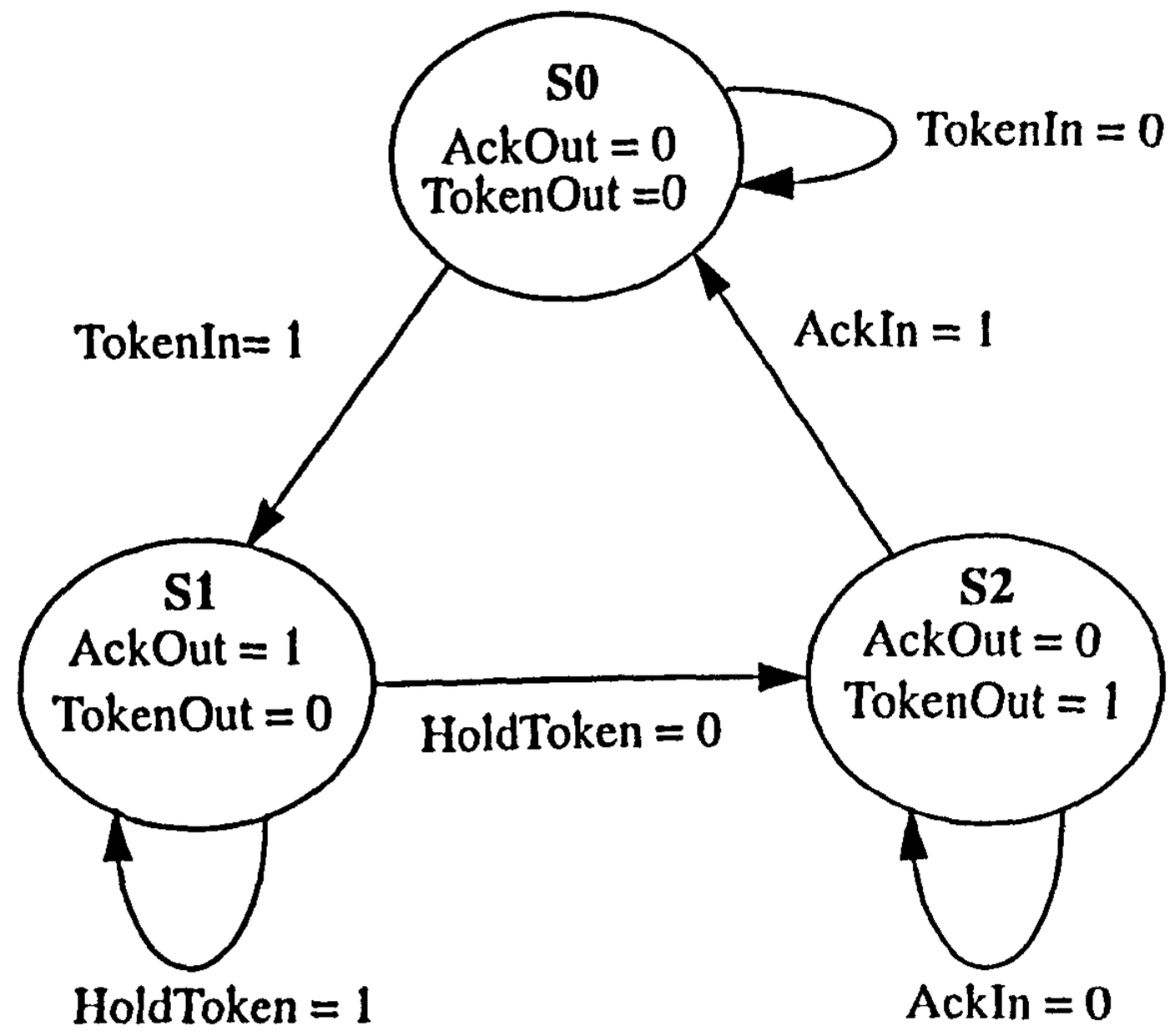
FIGURE 4.14. Generation of HoldToken signal

A state diagram (diagram which represents the mechanism of a state machine) for the token passing state machine is shown in Figure 4.15 on page 114. The SM remains in state zero (**S0**) until the token arrives (i.e **TokenIn = 1**) and then on the next clock edge proceeds to state one (**S1**) which acknowledges the arrival of the token by setting **AckOut** true. If **HoldToken** is true then the state machine remains in state one (**S1**), otherwise on the next clock edge it proceeds to state two (**S2**) which passes the token on by setting **TokenOut** logic true.

The SM does not go back to state zero until the passing of the token has been acknowledged (**AckIn = 1**). To inject the token into the system one state machine is programmed with the initial state holding **TokenOut** true (i.e the initial state is **S2**).

4.3.3.3 Token passing test circuit

To verify the algorithm for the token passing a test circuit was built (See Figure 4.16 on page 115). This contained three PLDs (GAL16V8s) each programmed with the token passing state machine. Light Emitting Diodes (LEDs) were attached to the **AckIn** pins in order to see the token as it passes round the circuit. To enable the flashing of the LED to be seen the state machine was clocked manually by toggling a switch. The switch is debounced by an S-R flip-flop. A photograph of the test circuit is shown in Appendix D, Figure 3 on page 305.



0 = logic 'false'

1 = logic 'true'

FIGURE 4.15. State Diagram for token passing

The state machines were programmed using the CUPL language. The source code for the token passing state machine is shown in Figure 4.17 on page 116 and the source code for the state machine that injects the token is shown in Figure 4.18 on page 117.

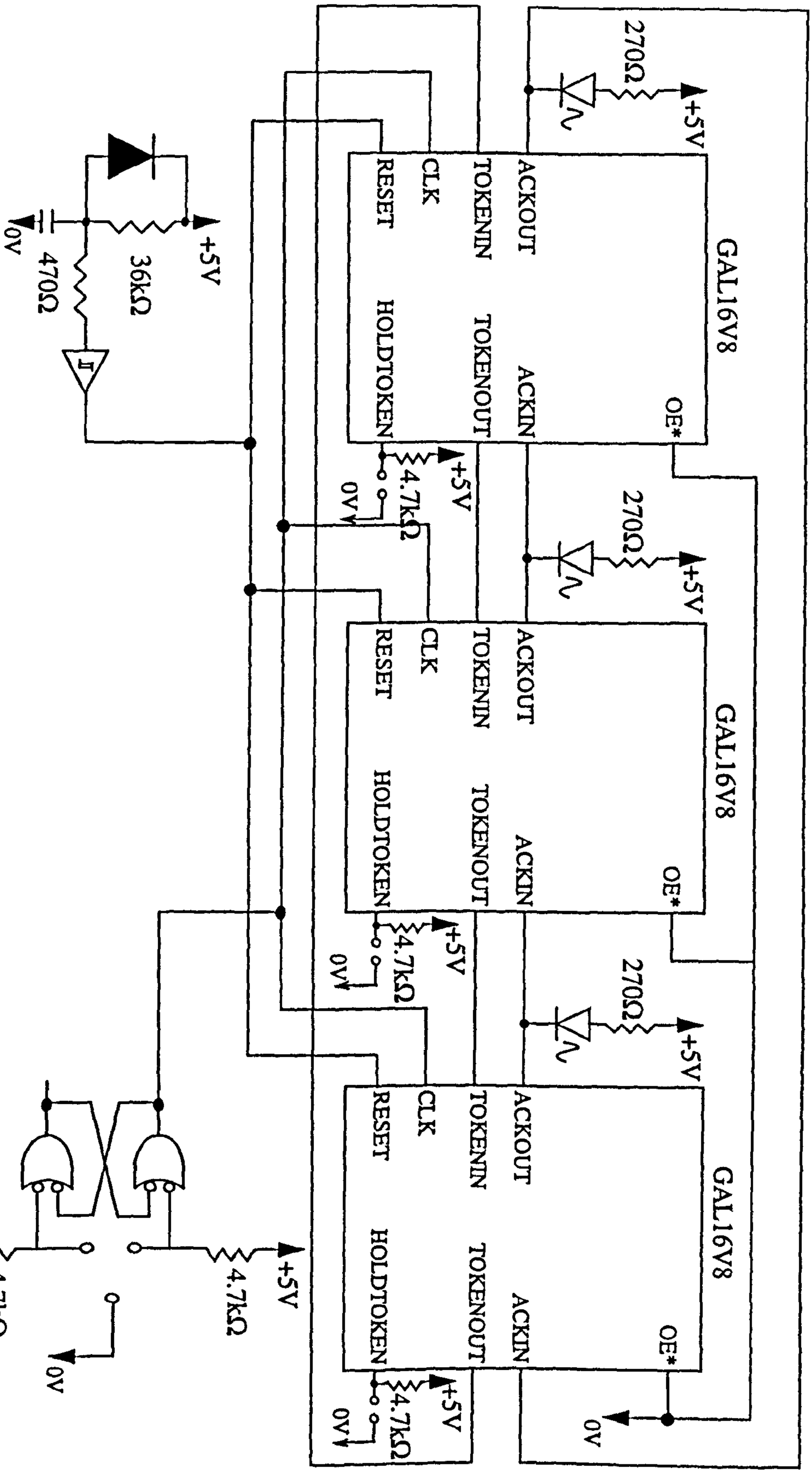


FIGURE 4.16. Token passing test circuit

```
/**INPUTS**/

PIN 1                   =CLK;
PIN 2                   =TOKENIN;
PIN 3                   =!ACKIN;
PIN 4                   =RESET;
PIN 5                   =HOLDTOKEN;
PIN 11                  =!OE;

/**OUTPUTS**/

PIN 14                  =TOKENOUT;
PIN 15                  =!ACKOUT;

FIELD STATEBIT          =[TOKENOUT,ACKOUT];

$DEFINE                 S0 'b'00
$DEFINE                 S1 'b'01
$DEFINE                 S2 'b'10

/**DEFINITIONS**/

NOTOKEN           =   !TOKENIN & !RESET;
TOKEN             =   TOKENIN & !RESET;
TOKENPASSED       =   ACKIN & !RESET;
TOKENNOTPASSED    =   !ACKIN & !RESET;
NOTHOLD           =   !HOLDTOKEN & !RESET;
HOLD              =   HOLDTOKEN & !RESET;
CLEAR             =   RESET;

SEQUENCE STATEBIT{

PRESENT S0      IF NOTOKEN          NEXT S0;   /*TOKEN NOT ARRIVED*/
                IF TOKEN            NEXT S1;   /*TOKEN ARRIVED*/
                IF CLEAR            NEXT S0;   /*RESET*/

PRESENT S1      IF NOTHOLD          NEXT S2;   /*PASS TOKEN*/
                IF HOLD             NEXT S1;   /*HOLD TOKEN*/
                IF CLEAR            NEXT S0;   /*RESET*/

PRESENT S2      IF TOKENPASSED      NEXT S0;   /*TOKEN PASSED*/
                IF TOKENNOTPASSED   NEXT S2;   /*TOKEN NOT PASSED*/
                IF TOKEN            NEXT S0;   /*POWER-UP STATE */
                IF CLEAR            NEXT S0;   /*RESET*/

}
```

FIGURE 4.17. CUPL source code for token passing

```
/**INPUTS**/

PIN 1      =              CLK;
PIN 2      =              TOKENIN;
PIN 3      =              !ACKIN;
PIN 4      =              RESET;
PIN 5      =              KEEPTOKEN;
PIN 11     =              !OE;
/**OUTPUTS**/
PIN 14     =              TOKENOUT;
PIN 15     =              !ACKOUT;
FIELD STATEBIT=           [TOKENOUT,ACKOUT];
$DEFINE S0 'b'10
$DEFINE S1 'b'00
$DEFINE S2 'b'01


/**DEFINITIONS**/

SEQUENCE STATEBIT{
NOTOKEN    =              !TOKENIN & !RESET;
TOKEN      =              TOKENIN & !RESET;
TOKENPASSED=              ACKIN & !RESET;
TOKENNOTPASSED=           !ACKIN & !RESET;
NOTHOLD    =              !HOLDTOKEN & !RESET!
HOLD       =              HOLDTOKEN & !RESET;
CLEAR      =              RESET;

PRESENT S0                IF TOKENPASSED      NEXT S1;  /*TOKEN PASSED*/
                          IF TOKENNOTPASSED   NEXT S0;/*TOKEN NOT PASSED*/
                          IF TOKEN            NEXT S0;
                          IF CLEAR            NEXT S0;  /*RESET*/

PRESENT S1                IF NOTOKEN          NEXT S1; /*TOKEN NOT ARRIVED*/
                          IF TOKEN            NEXT S2;  /*TOKEN ARRIVED*/
                          IF CLEAR            NEXT S0;  /*RESET*/

PRESENT S2                IF NOTHOLD          NEXT S2; /*PASS TOKEN*/
                          IF HOLD             NEXT S2; /*HOLD TOKEN*/
                          IF CLEAR            NEXT S0;  /*RESET*/
}
```

FIGURE 4.18. CUPL source code for node which injects token in to system

The syntax for the state machine is fairly self explanatory. The state variables (also the outputs) are defined using the FIELD statement and the three states (**SO**,**S1** and **S2**) of the state variables are defined using the $DEFINE statement. A binary ('b') 0 in the $DEFINE statement indicates the state variable is logic false and a binary 1 indicates the state variable is logic true (independent of whether it was defined active high or low in the pin assignments).

The "DEFINITIONS" define the different combinations on the input pins and the SEQUENCE statements actually run the state machine. For each state (**S0**,**S1** and **S2**) the next state is determined by the levels on the input pins at the next clock edge. By following through the SEQUENCE statements and comparing them with the state diagram it is reasonably simple to see how the state machine functions.

The **HoldToken** signal was generated placing a jumper between the signal and ground (by default the signal is pulled logic high). This was obviously just for test purposes.

4.3.4 FIFO Access

4.3.4.1 C011

The connection request in the FIFO must be clocked out a byte at a time to the C011 and then sent to the control processor. A C011 is similar to a C012 in that it converts a bi-directional serial link into parallel data streams. The link adapter can operate in one of two modes.

In Mode 1 the IMS C011 converts between a link and two independent fully handshaken byte-wide interfaces, one input and one output. It can be used by a peripheral device to communicate with a transputer, an INMOS peripheral processor, or another link adapter, or it can provide programmable input and output pins for a transputer.

When in Mode 2 the CO11 provides an interface between an INMOS serial link and a microprocessor system bus. In fact a C011 behaves in exactly the same way as a C012 when in Mode 2. However, the C011 used in the dynamic on-demand circuit switched network is in Mode 1 (See Figure 4.19).

The eight bit parallel input port **IO-7** can be read by a transputer family device via the serial link. **IValid** and **IAck** provide a simple two-wire handshake for this port. When data is valid on **IO-7, IValid** is taken high by the peripheral device to commence the handshake. The link adapter transmits data presented on **IO-7** out through the serial link.

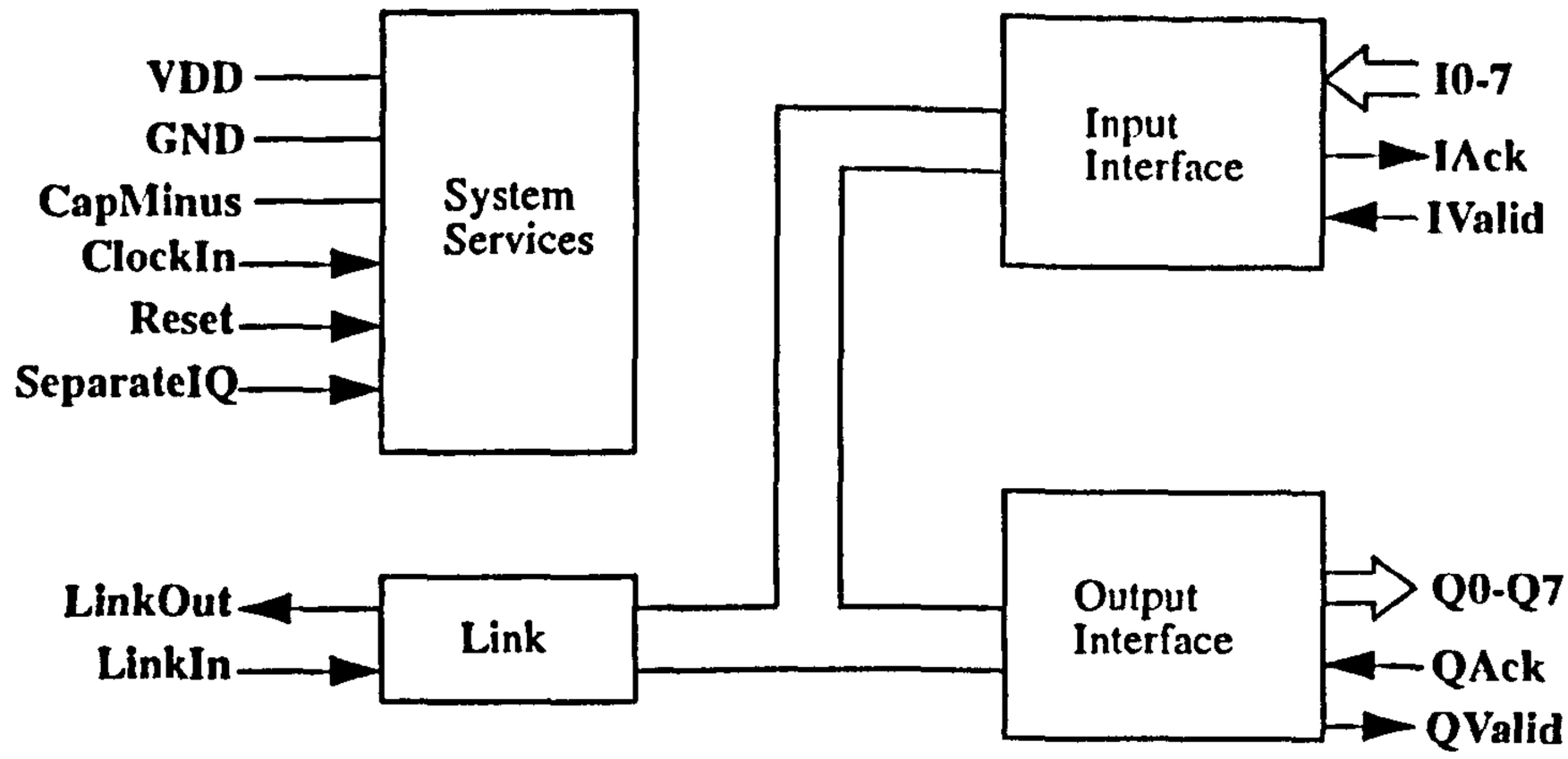


FIGURE 4.19. IMS C011 Mode 1 block diagram

After the data byte transmission has been completed and an acknowledge packet is received on the input link, the IMS C011 sets **IAck** high. To complete the handshake, the peripheral device must return **IValid** low. The link adaptor will then set **IAck** low.

The eight bit parallel output port **Q0-7** can be written to by a transputer family device via the serial link. **Qvalid** and **QAck** provide a simple two-wire handshake for this port.

A data packet received on the input link is transferred onto **Q0-Q7**; the link adapter then takes **QValid** high to initiate the handshake. After reading the data from **Q0-Q7**, the peripheral device sets **QAck** high. The IMS C011 will then send an acknowledgement packet out of the serial link to indicate a completed transaction and set **QValid** low to complete the handshake.

The rest of the signals on the C011 are the same as a C012 apart from the **SeparateIQ** signal. This is used to set the C011 to the different modes (Mode 1 and Mode 2). Mode 1 is selected by connecting **SeparateIQ** to **VDD** (sets the LinkSpeed to 10Mbits/sec) or to **ClockIn** (20Mbits/sec).

In the circuit switching network it is the input port **(IO-7)** that is connected to the FIFO and the output port **(Q0-7)** is connected to a buffer.

4.3.4.2 FIFO clocking state machine

The clocking of the data from the FIFO to the C011 is achieved by a finite state machine (See Figure 4.20).

The state machine controls the **RD*** (read) signal on the FIFO and the **IAck** and **IValid** signals on the C011. Pulling the **RD*** signal low transfers a byte out of the FIFO to the C011 parallel port. In order to transmit the byte from the parallel port to the INMOS OSLink, **IValid** is pulled high. To indicate the byte has been transferred successfully **IAck** is pulled high by the C011 and then **IValid** returned low by the SM.

The **Empty_Flag*** on the FIFO signals to the state machine when data is present in the FIFO, and an output called **TokenArrived** from the token passing state machine indicates when the token is present. The state machine waits at **S0** while the FIFO is empty or the token has not arrived. When the token arrives and there is data in the FIFO the SM then proceeds to **S1** on the next clock edge and this initiates a read cycle on the FIFO. On the next clock edge the SM then unconditionally jumps to **S2** which takes **IValid** true and enables the buffer ('125) that restricts access to the serial bus. The SM then waits for **IAck** to become true, indicating the transfer of a byte to the INMOS OSLink, before proceeding back to **S0**.

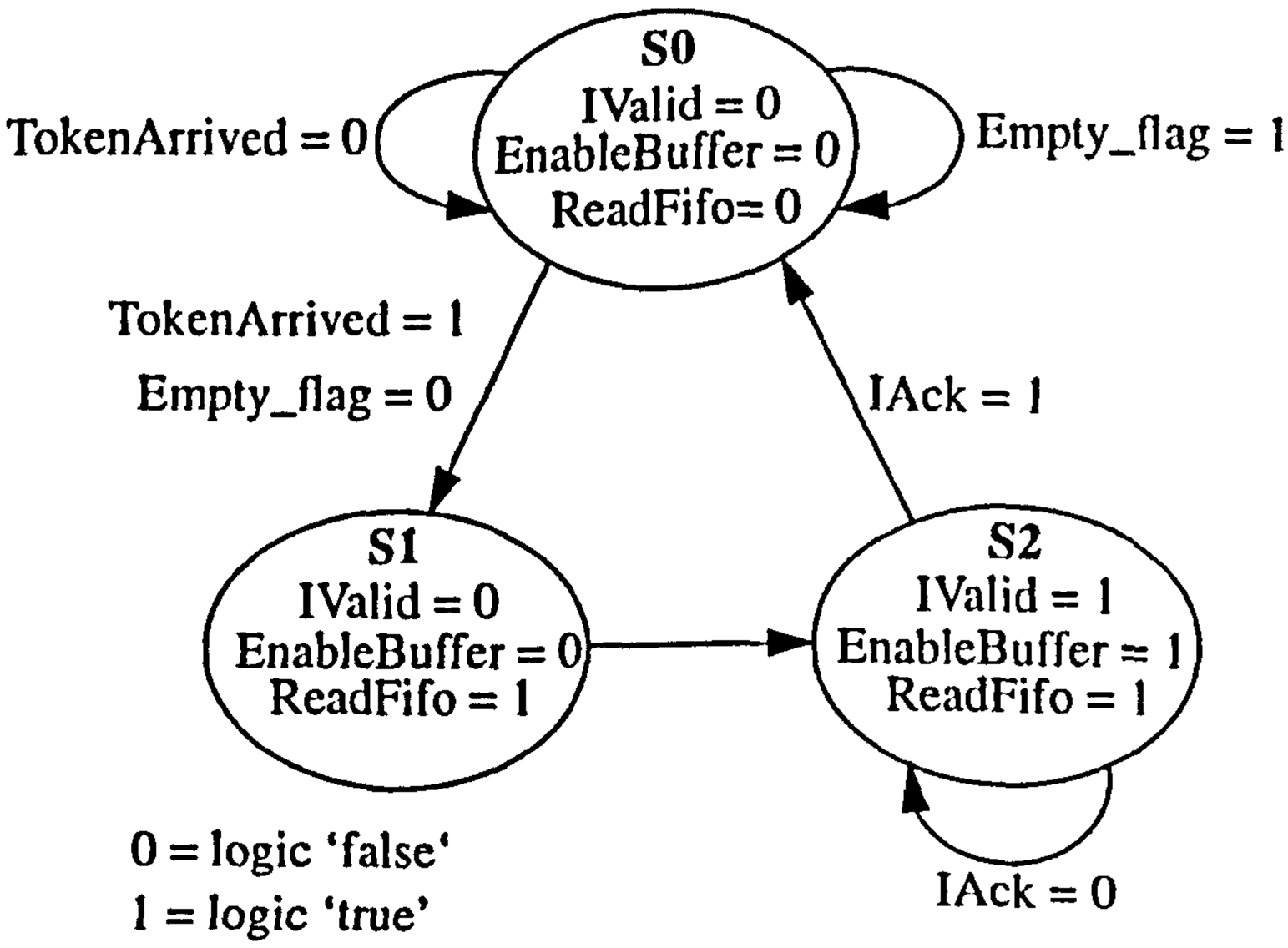


FIGURE 4.20. State Diagram for FIFO clocking

4.3.4.3 Fifo clocking test Circuit

To test the state machine for the FIFO clocking a test circuit was built on a PC card (See Figure 4.21). A byte (or bytes) is written from the PC into the FIFO and then the state machine clocks the byte (or bytes) out of the FIFO to the C011. The message is then sent from the C011 via the '125 buffer to the DB9 connector which is connected to a dual link adapter board (as described in Chapter 3) plugged into the same PC. A photograph of the circuit board is shown in Appendix D, Figure 4 on page 305.

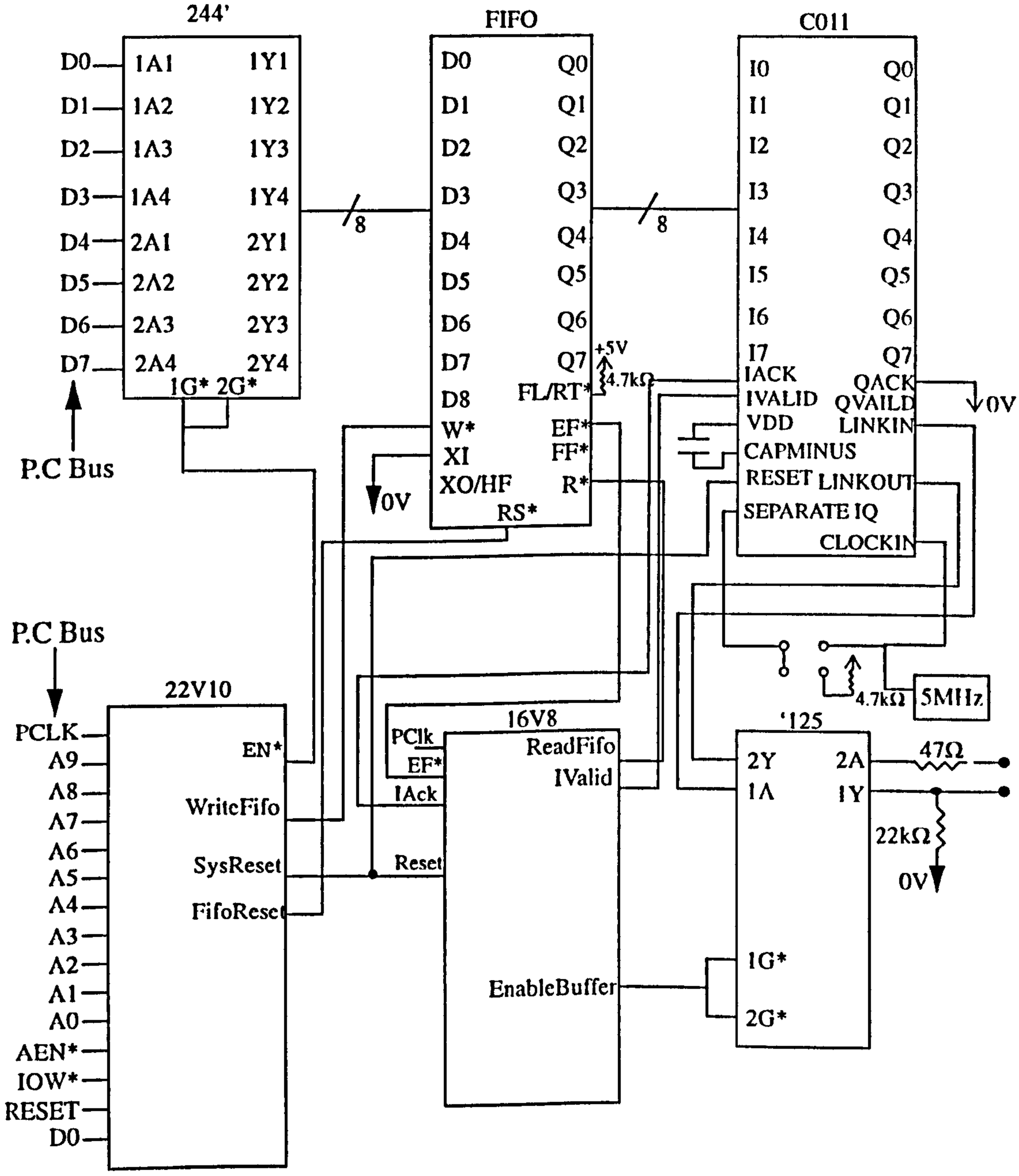


FIGURE 4.21. Fifo clocking test circuit

The message is received by the dual link adapter board and it is then compared with the message transmitted to the FIFO to verify that the message is being transferred correctly.

A PAL (P22V10L) is used for the FIFO address decoding (CUPL source code for the PAL is shown in Figure 4.22).

```
/**INPUTS**/
PIN 1     =     PCLK;
PIN 2     =     A9;
PIN 3     =     A8;
PIN 4     =     A7;
PIN 5     =     A6;
PIN 6     =     A5;
PIN 7     =     A4;
PIN 8     =     A3;
PIN 9     =     A2;
PIN 10    =     A1;
PIN 11    =     A0;
PIN 13    =     !NOTAEN;
PIN 14    =     !NOTIOW;
PIN 15    =     RESETDRV;
PIN 16    =     D0;

/**OUTPUTS**/

PIN 17    =     !EN;
PIN 18    =     !WRITEFIFO;
PIN 19    =     SYSRESET;
PIN 20    =     LATCHRESET;
PIN 21    =     FIFORESET;

/**RESETS AND PRESETS**/

LATCHRESET.AR       =     'b'0;
LATCHRESET.SP       =     'b'0;

/**DEFINITIONS**/

FIELD ADDRESS = [A9..A0];

/**INTERMEDIATE VARIABLES**/

FIFO              =     ADDRESS:'h'[100] & NOTAEN;
WRITERESET        =     ADDRESS:[101] & NOTAEN & NOTIOW;

WRITEFIFO         =     NOTIOW & FIFO & !RESETDRV;
EN                =     WRITEFIFO;
LATCHRESET.d      =     D0 & WRITERESET # LATCHRESET & !WRITERESET;
SYSRESET          =     RESETDRV # LATCHRESET;
FIFORESET         =     SYSRESET;
```

FIGURE 4.22. CUPL source code for P22V10L in FIFO clocking circuit

In order to write data into the FIFO the PC writes to address H#100. This is similar to the address decoding for the dual link adapter board in Chapter 3. Also similar is the system reset which again allows the user to reset the board by writing a binary '1' to address H#101. The board is also reset during power on as **SYSRESET** is logic true when **RESETDRV** (the PCs system reset) is logic true.

The CUPL source code for the FIFO clocking is shown in Figure 4.23 on page 124. Again by comparing this code with the state diagram for the FIFO clocking it should be clear how the code functions. The main difference between the code and state diagram is that in the code there is also a third state (**S3**). This is required as the PAL will power up in this state and needs to be reset to state zero (S0). Also the **TokenArrived** signal from the token passing state machine is not required as it is only the FIFO clocking that is being tested.

4.3.5 Hardware Interface to control processor

This circuit contains the control processor (an Analog devices ADSP-2105), two C012s, an EPROM and a PAL (See Figure 4.24 on page 125). The serial link on one of the C012s is connected to the Inmos OSLink connected to the nodes and the other C012 is connected to the crossbar switch. The control processor receives messages from the nodes via one C012 and then programs the crossbar switch via the other C012. A photograph of the circuit board is shown in Appendix D, Figure 5 on page 306.

4.3.5.1 ADSP-2105

The ADSP-2105 is a 12MHz microcomputer suitable for high-speed numeric processing applications[9,10]. It contains 1K words of on-chip program memory RAM and 512 words of on-chip data memory RAM (i.e Harvard Architecture). The internal program memory can be loaded from an EPROM (i.e. the contents of the EPROM are loaded into program memory).

```
/**INPUTS**/

PIN 1           =CLK;
PIN 2           =IACK;
PIN 3           =!EF;
PIN 5           =CLEAR;
PIN 11          =!OE;

/**OUTPUTS**/
PIN 12          =IVALID;
PIN 13          =!ENABLBUFFER;
PIN 14          =!READFIFO;

FIELD STATEBIT=[IVALID,ENABLEBUFFER,READFIFO];

$DEFINE S0 'b' 000
$DEFINE S1 'b' 001
$DEFINE S2 'b' 111
$DEFINE S3 'b' 100

/**DEFINITIONS**/

FIFOEMPTY       =EF & !CLEAR;
FIFONOTEMPTY    =!EF & !CLEAR;
DATASENT        =IACK & !CLEAR;
DATANOTSENT     =!IACK & !CLEAR;
RESET           =CLEAR;

SEQUENCE STATEBIT{

PRESENT S0      IF FIFOEMPTY            NEXT S0;
                IF FIFONOTEMPTY         NEXT S1;
                IF RESET                NEXT S0;

PRESENT S1      IF RESET                NEXT S0;
                IF FIFOEMPTY            NEXT S2;
                IF FIFONOTEMPTY         NEXT S2;

PRESENT S2      IF DATANOTSENT          NEXT S2;
                IF DATASENT             NEXT S0;
                IF RESET                NEXT S0;

PRESENT S3      NEXT S0;

}
```

FIGURE 4.23. CUPL code for FIFO clocking

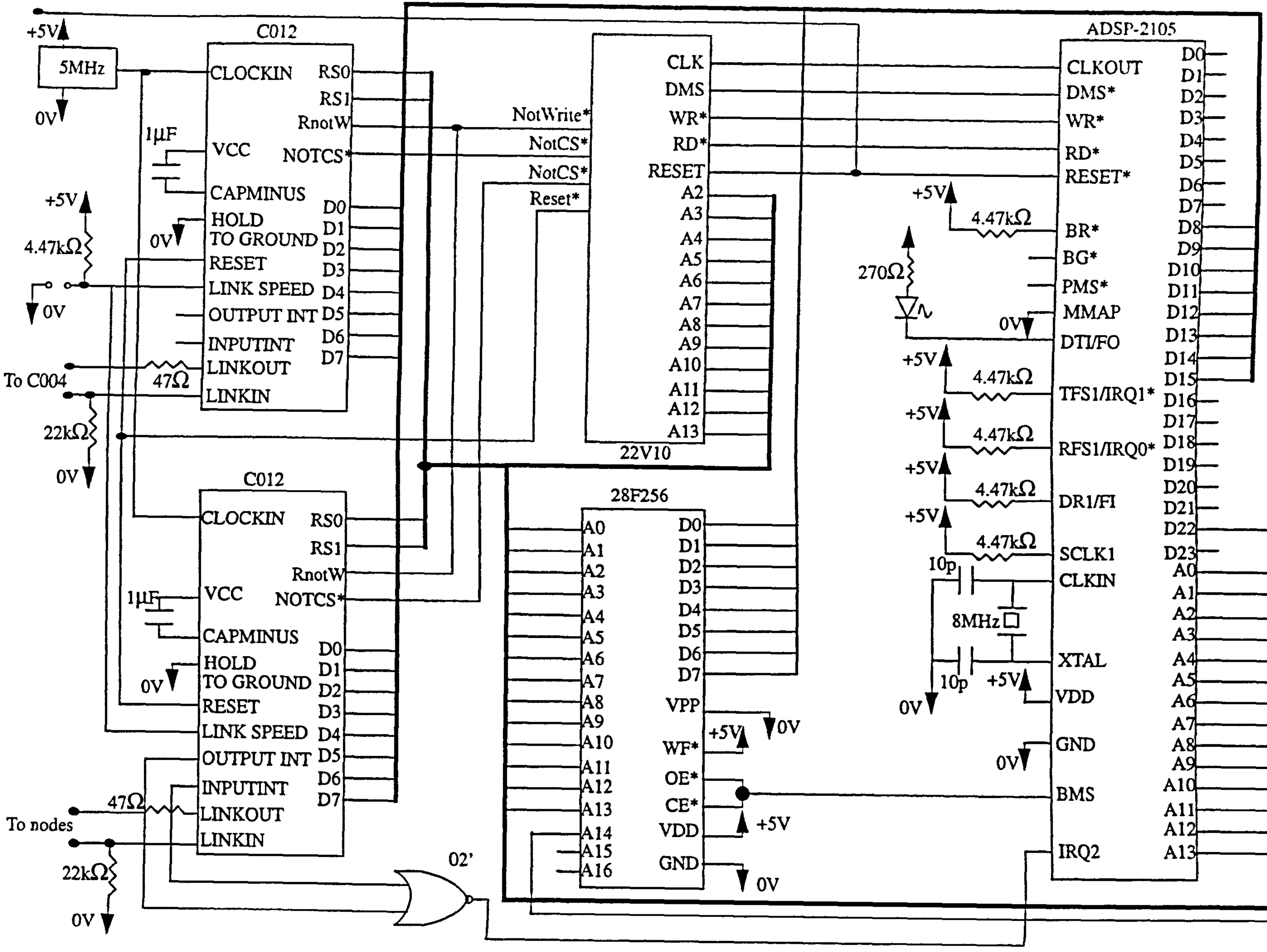


FIGURE 4.24. Hardware Interface to Control Processor

The core architecture of the ADSP-2105 consists of the following elements:

- Arithmetic-Logic Unit (ALU)
- Multiplier-Accumulator (MAC)
- Barrel Shifter
- Two Data Address Generators (DAG)
- Program Sequencer
- Program Memory Address (PMA) Bus
- Program Memory Data (PMD) Bus
- Data Memory Address (DMA) Bus
- Data Memory Data (DMD) Bus
- Result (R) Bus

Figure 4.25 on page 127 shows a block diagram of this core internal architecture

The computational units process 16-bit data directly and have provision to support multiprecision computations. Table 4.1. shows the operations performed by each of the computational units.

Computational Unit	Operations Supported
ALU	Arithmetic and Logic (Division Primitives also)
MAC	Single-cycle multiply, multiply/add and multiply/subtract
Shifter	Logical and arithmetic shifts, normalisation, denormalisation and derive exponent

TABLE 4.1. Operations supported by Computational Units

Instruction addresses are supplied to the program memory from the program sequencer. The sequencer is driven by the Instruction Register which holds the currently executing instruction. Instructions are fetched, loaded into the instruction register, and decoded during one processor cycle; and executed during the following cycle while the next instruction is prefetched.

The data address generators (DAGs) handle address pointer up-dates. Each DAG maintains four address pointers. Whenever the pointer is used to access data (indirect addressing), it is post-modified by the value of a specified modify register. A length value may be associated with each pointer to implement automatic modulo addressing for circular buffers. The two DAGs differ: DAG1 only generates data memory addresses, but provides an optional bit-reversal capability; DAG2 can generate both data memory and program memory addresses, but has no bit-reversal capability.

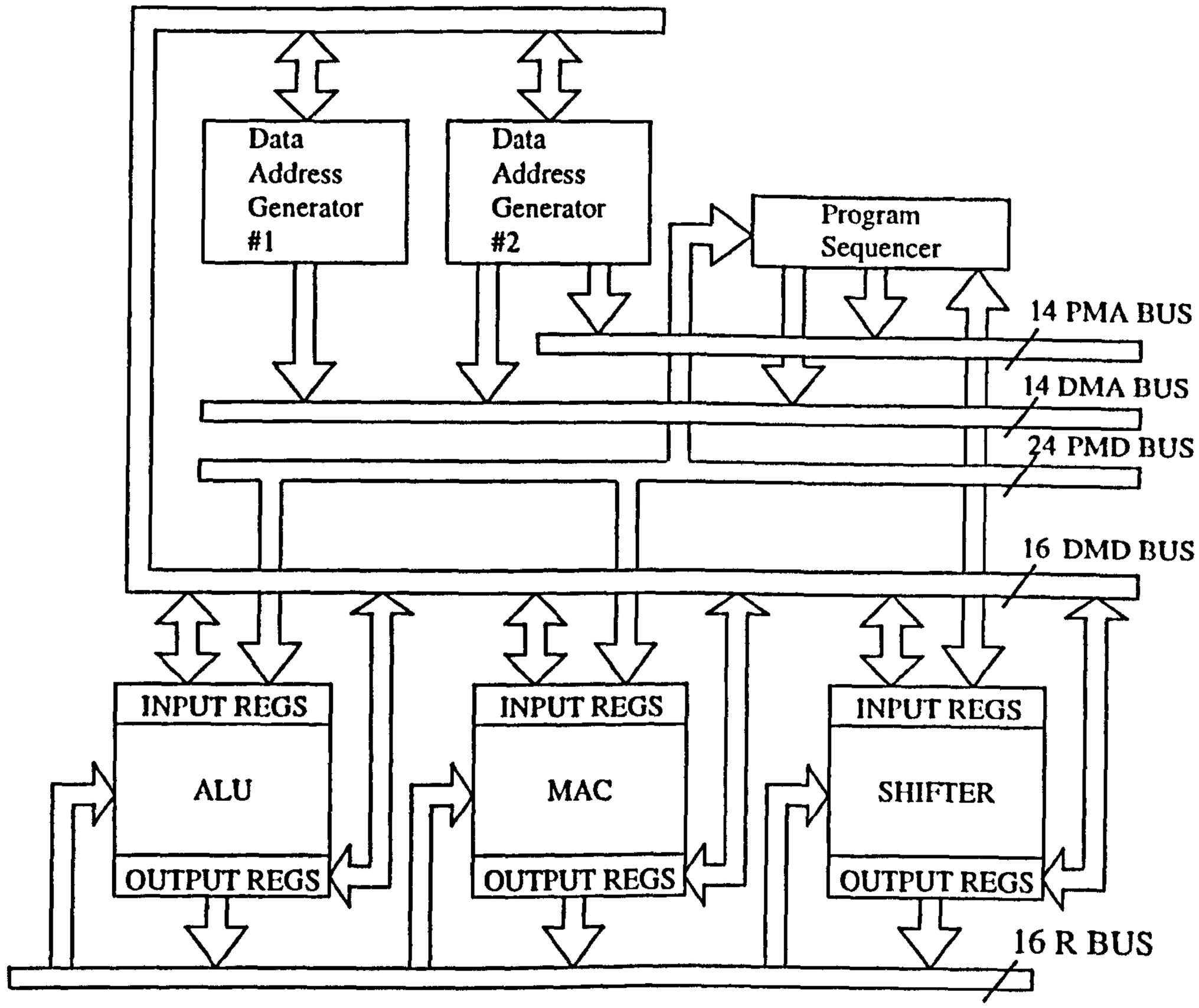


FIGURE 4.25. Core Architecture of ADSP-2105

Five internal buses support the internal components: The PMA and DMA buses are used internally for the addresses associated with Program and Data Memory. The Program Memory Data (PMD) and Data Memory Data (DMD) buses are used for the data associated with the memory spaces. These two pairs of buses are multiplexed off chip to the external address and data buses. The BMS* (Boot Memory Select), DMS*(Data Memory Select) and PMS* (Program Memory Select) signals (pins) select the different address spaces. The R bus is an internal bus which transfers intermediate results directly between the various computational sections.

The contents of any register in the processor can be transferred to any other register or to any external data memory location in a single cycle via the DMD bus. The data memory address comes from two sources: an absolute value specified in the instruction

code (direct addressing) or the output of a data address generator (indirect addressing). Only indirect addressing is supported for data fetches from program memory.

Program memory can store both instructions and data, permitting the ADSP-2105 to fetch two operands in a single cycle, one from program memory and one from data memory. The ADSP-2105 can fetch an operand from on-board program memory and the next instruction in the same cycle.

The ADSP-2105 contains many registers. Some of these store values; i.e AX0 stores an ALU operand, I4 stores a DAG2 pointer. Other registers consist of control bits and fields, or status flags. For example, ASTAT contains status flags from arithmetic operations, and fields in DWAIT control the numbers of wait states for different zones of data memory. The purpose of each of the registers will be explained as required whilst describing the software developed for the ADSP-2105.

The instruction set provides flexible data moves and multifunction (one or two data moves with a computation) instructions. Every instruction can be executed in a single processor cycle. The ADSP-2105 assembly language uses an algebraic syntax for ease of coding and readability. The details of the assembler will be described when the software for the switching network is explained.

The ADSP-2105 is supported by a complete set of tools for software and hardware development. The System Builder provides a high-level method for defining the architecture of systems under development. The Assembler produces object code and the Linker combines object code modules and library calls into an executable file. To aid in hardware and software debugging of ADSP-2105 systems an interactive instruction level simulator is provided. To create a PROM burner compatible file a PROM splitter is used.

4.3.5.2 Interface between ADSP-2105 and EPROM

The ADSP-2105 in the switching network is booted from an EEPROM. The boot memory space consists of an external 32K by 8 space, divided into eight separate 4K by 8 pages. Boot loading from page 0 after RESET* is initiated automatically if the MMAP pin is grounded.

Figure 4.24 on page 125 shows the interface between the ADSP-2105 and the EEPROM (28F256). The 28F256 is an 32K by 8 electrically erasable PROM. When the CE* and OE* pins are pulled logic low and WF* is logic high, the data stored at the memory location determined by the address pins is asserted on the outputs. The outputs are put in the high impedance state whenever CE* or OE* is high.

To initiate a programming cycle a low pulse is applied to the WF* or CE* input with CE* or WF* low (respectively) and OE* high. The address is latched on the falling edge of CE* or WF*, whichever occurs last. The data is latched by the first rising edge of CE* or WF*. Once a byte write has been started it will automatically time itself to completion. During a write cycle a supervoltage of 13V is applied to the V_{pp} pin.

The 28F256 is interfaced to the ADSP-2105 via the BMS*, D0-D7 and A0-A14 lines. The BMS* signal (active low) is used to select the boot memory interface and therefore this pin is attached the OE* and CE* pins on the 28F256; both these pins require to be low when reading from the EEPROM. The WF* pin is pulled logic high permanently as the 28F256 is only read from the ADSP-2105 and not written to.

Pins D0-D7 on the 28F256 are connected to pins D8-D15 on the ADSP-2105 as these are the pins used for 8-bit data on the microcontroller. The address lines between the two chips are connected as normal except for A14 which is connected to D22 on the ADSP-2105. This is as in order to accommodate up to eight pages of boot memory, the two MSBs of the data bus are used in the boot memory interface as the two MSBs of the boot address space.

The V_{pp} pin used to apply the supervoltage to program the device is grounded when in circuit.

4.3.5.3 Interface between ADSP-2105 and C012s

The ADSP-2105 is interfaced to two C012s; one connected to the nodes and the other to the C004 crossbar switch. A PAL (P22V10L) is used to address the two C012s. The CUPL source for the P22V10 is shown in Figure 4.26 on page 130.

```
Pin 1          = PCLK;          /*PROCESSOR CLOCK   */
Pin 2          = !NOTDMS;       /*DATA MEMORY SELECT SIGNAL   */
Pin 3          = !NOTWR;        /*WRITE SIGNAL   */
Pin 4          = !NOTRD;        /*READ SIGNAL   */
Pin 5          = !NOTRESET;     /*RESET SIGNAL FROM LKADAP BOARD
Pin [6..11]     = [A2..A7];     /*ADDRESS LINES*/
Pin [13..18]  = [A8..A13];      /* ADRESS LINES*/

/**Outputs**/

Pin 19         = !NOTWRITE;     /*NOTWRITE ON C012S*/
Pin 20         = !NOTCSLKADP0;/*SELECT LINK ADAPTER 0*/
Pin 21         = !NOTCSLKADP1;/*SELECT LINK ADAPTER 1*/
Pin 22         = RESET;         /*RESET*/
Pin 23         = !NOTSTATWR;  /*DELAY WRITE SIGNAL*/

/** Declarations and Intermediate Variable Definitions **/

FIELD ADDRESS = [A13..A2];
LKADP_1        = ADDRESS:[4];
LKADP_0        = ADDRESS:[8];
NOTSTATWR       = NOTWR;
READLKADP1      = LKADP_1 & NOTRD & NOTDMS;
READLKADP0      = LKADP_0 & NOTRD & NOTDMS;
WRITELKADP1     = LKADP_1 & NOTWR & NOTDMS;
WRITELKADP0     = LKADP_0 & NOTWR & NOTDMS;

/**  Logic Equations  **/

NOTWRITE        = NOTWR # NOTSTATWR;
NOTCSLKADP0     = READLKADP0 # (WRITELKADP0 & NOTSTATWR);
NOTCSLKADP1     = READLKADP1 # (WRITELKADP1 & NOTSTATWR);
RESET        = NOTRESET;
```

FIGURE 4.26. CUPL source code for P22V10

This interface is very similar to the interface to the PC ISA Bus used in the Dual Link Adapter board described in Chapter 3. The inputs to the P22V10 are **CLKOUT, DMS*,WR*,RD*, A2-A13** from the DSP2105 and **RESET*** which is from a link adapter board. The outputs are the **RnotW, NotCs*** and **Reset** signals for the C012s.

The **DMS*** strobe is used to select the data memory and the **WR*** and **RD*** signals are the write and read signals respectively. The intermediate variables **LKADP_1** and **LKADP_0** define the addresses of the C012s. Like the situation in the dual link adapter board the **NotIOW** signal has to be delayed to create the **NotCS*** signals and therefore the **NotStatWr.d** signal is created.

The intermediate variables and logic equations are very similar to those described for the dual link adapter board. The main difference is that when reading and writing to the C012s the **DMS*** signal must also be true (as well as the **RD*** or **WR*** strobes and address). To reset the C012s the **!NOTRESET** signal is inverted as the C012 reset is active high.

4.3.5.4 Other connections to ADSP-2105

All the inputs to the ADSP-2105 that are not used are tied to +5V. An 8MHz crystal oscillator is connected between the **CLKIN** and **XTAL**. To allow the C012 connected to the nodes to interrupt the ADSP-2105 when a byte has arrived the OutputInt and InputInt pins on the C012 are connected via an OR gate to the **IRQ2** (External Interrupt Request #2) input on the ADSP-2105.

4.3.5.5 Testing of Circuit

To test the booting from EEPROM code was developed which flashed an LED connected to the FLAG_OUT pin on the ADSP-2105. The programmable interval timer which can generate periodic interrupts was used as a signal to turn the light off and on.

The system specification source file (**.SYS** file)[11] which describes the target hardware is shown in Figure 4.27 on page 132. This **.SYS** file is processed by the system builder to generate an architecture description file (**.ACH** file). The **.ACH** file is interpreted by the linker in order to place relocatable code and data fragments in memory.

The .SYSTEM directive at the start of the code assigns the name *control* to the architecture description and marks the start of the file. To identify the processor type the statement .ADSP-2105 is required. The .MMAP directive specifies the state of the MMAP pin on the ADSP-2105 device. Defining MMAP as 0 indicates that boot memory is to be loaded into the chip's internal program memory beginning at address H#0000.

```
.SYSTEM                          control;
.ADSP2105;
.MMAP0;
.SEG/BOOT=0/ROM                  boot_mem1[1024];
.SEG/RAM/ABS=H#3800/DM/DATA int_dm[512];
.SEG/RAM/ABS=0/PM/DATA/CODE int_pm[1024];
.ENDSYS;
```

FIGURE 4.27. .SYS file for flashing light

The .SEG directives declare the system memory segments and their characteristics. Memory segments can be declared in any order. In this case it is only the boot memory space (*boot_mem1*), the internal program memory (*int_pm*) and the internal data memory (*int_dm*) that are declared.

To identify the 1K-word segment for one page of boot memory the *boot_mem1* segment is declared. The *int_dm* declaration identifies 512 bytes of on-chip data memory starting at absolute address H#3800. The memory space from H#0-H#3800 is reserved for external RAM and from H#3A00 to H#3FFF is reserved for control registers for the system, timer, wait state configuration and serial port operations.

The *int_pm* declaration identifies the 1K of program memory starting at absolute address 0 which can store both code and data. There is no external data or program memory in the circuit

To mark the end of the file the .ENDSYS directive is used. The system builder stops processing when it encounters the directive.

The assembler source code (**.DSP** file)[11] which flashes the LED is illustrated in Figure 4.28 on page 133. To mark the beginning of the program module and define the module name (*flash_led*) the .MODULE directive is used. The .INCLUDE directive is used to include another source file in the file being assembled. The file included in this case (DEF2105.h) initialises the memory mapped control registers and gives them symbolic names (i.e.Sys_Ctrl_Reg for the System Control Register). This makes manipulation of the registers simpler in the program.

```
.MODULE/RAM/BOOT=0flash_led;
.INCLUDE<E:\ADI_DSP\INCLUDE\DEF2105.h>;

                    JUMP restarter;NOP;NOP;NOP;
                    RTI;NOP;NOP;NOP;
                    NOP;NOP;NOP;NOP;
                    NOP;NOP;NOP;NOP;
                    NOP;NOP;NOP;NOP;
                    JUMP flash;NOP;NOP;NOP;

restarter:          CALL initialisations;
wait_loop:          IDLE;
                    JUMP wait_loop;

initialisations:    AX0=H#FFFF;
                    DM(Tperiod_Reg)=AX0;{Set counter)
                    DM(Tcount_Reg)=AX0;
                    AX0=H#1B;
                    DM(Tscale_Reg)=AX0;
                    IMASK=1;
                    ENA TIMER;
                    RTS;

flash:              TOGGLE FLAG_OUT;
                    RTI;

.ENDMOD;
```

FIGURE 4.28. Source code for flash.dsp

The first 28 addresses in program memory contain the restart and interrupt vectors (0x0000 - 0x001B). The 29th PM address (0x001C) holds the first program instruction. Since *flash_led* is declared at absolute address zero, the first 28 instructions are placed in the interrupt vector locations. As *flash_led* uses only the restart (0x0000) vector and the TIMER interrupt the remaining instructions are simply returns (RTI) or non operations (NOP).

The routine *initialisations* initialises the timer. It is the period register (Tperiod_Reg) that holds the period of the interrupt in cycles and when the timer is enabled the count register(Tcount_Reg) is decremented as often as once every instruction cycle. When the counter reaches zero an interrupt is generated. Tcount_Reg is then reloaded from Tperiod_Reg and the count begins again.

The timer scaling factor register (Tscale_Reg) stores a scaling factor that is one less than the number of cycles between decrements of Tcount_Reg. For example, if the value in Tscale_Reg is 0, the counter register decrements once every cycle. Therefore

using these three registers, interrupts from 5.24ms (when Tperiod_Reg is at maximum and Tscale_Reg at minimum with resolution of 80ns) up to 1.34 seconds (when both Tperiod_Reg and Tscale_Reg are maximum with resolution of 20.48μs) with an 80ns cycle time can be generated.

By setting Tperiod_Reg and Tcount_Reg to H#FFFF and Tscale_Reg to H#1B this provides an interrupt every 0.3s. This is enough so that the flashing of the LED can be seen by the naked eye.

The value in the IMASK register is set to one to enable the TIMER interrupt. The ENA TIMER command starts the timer decrementing logic. To return from the routine *initialisations* the RTS command is required.

The IDLE command causes the program to loop indefinitely in a low-power state, waiting for interrupts. When a timer interrupt does occur the program jumps to the *flash* routine. This routine toggles the FLAG_OUT pin and then returns to the instruction following the IDLE instructions. In this case this is JUMP instruction back to the IDLE instruction causing the program to wait for another interrupt. In this way the light flashes continuously.

This simple example shows the basics of ADSP-2105 assembler and was used to test that the ADSP-2105 was being booted successfully from the EEPROM.

4.3.5.6 Booting Program

To develop the software for controlling the switching network a program was written whereby the ADSP-2105 could be booted via one of the C012s on the control processor board[12]. The program to be downloaded is sent to the C012 from a host computer (a PC). This allows different versions of the program to be tested without having to re-program the EEPROM each time. Obviously the ADSP-2105 has to be booted from EEPROM with some initial code that loads the bytes from the C012. The pseudocode for this monitor program is illustrated in Figure 4.29 on page 135 and the source code is shown in Appendix B, pages 235 - 236.

```
Initialise registers and variables
If  input status register is not equal to zero (i.e. C012 contains data) then
   If no.of instructions < 0 then
     load no.of instructions byte at a time from C012
   If no.of instructions > 0 then
     If first byte of instruction
       load most significant byte into register SI
       decrement counter which counts bytes
     If second byte of instruction then
       load middle byte into register SR0
       decrement counter which counts bytes
    If third byte of instruction then
       load least significant byte into register PX
       load instruction (3 bytes) into program memory
       reset counter which counts bytes
       decrement instruction counter
 If no. of instructions = 0 then
       jump to start of downloaded program
```

FIGURE 4.29. Pseudocode for download program

The ADSP-2105 receives incoming instructions, loads them into program memory and when all instructions have been received executes them. However, in order to know when the download is complete the program must know the number of instructions that are to be downloaded. The first two bytes of a downloaded program therefore contain the number of instructions.

Once the program has initialised various registers and variables it monitors the input status register on the C012 waiting for data to arrive. Once data arrives it first of all checks the value of the number of instructions to be dowloaded. If the number of instructions is less than zero (it is initialised to a negative value) then this signals that the byte to be downloaded is one of the two bytes which contain the number of instructions.

The two bytes are loaded one at a time into the shifter registers and combined to give the 16-bit variable *ins_count* which contains the number of instructions.

The program then waits for another byte to arrive. If the number of instructions has been loaded previously then this byte will be a byte of the downloading program. The ADSP-2105 instructions are 24 bits wide so each instruction is three bytes long. A separate counter (*count*) is used to count the three bytes as they arrive. It is reset after each instruction is downloaded completely.

Each byte of an instruction is loaded into different registers. Whenever a program memory write occurs, the sixteen most significant bytes are supplied by the source register explicitly named in the instruction, and the eight LSBs are supplied by the PX register. The basic tactic of the monitor program is to assemble the two most significant bytes of an instruction in a data register (using the Shifter) and load PX explicitly with the least significant byte.

Once this is achieved a program memory write then writes the correct twenty-four bit instruction into memory. In order that the downloaded program avoids overwriting the monitor program while the monitor executes, the downloaded program must be placed in memory after the monitor program. This is achieved by labelling the end of the monitor program with a label (*code_start*) and using a DAG (data address generator) to generate the addresses for the downloaded program instructions.

The I register in a DAG contains the actual address used to access memory. This register is loaded with the address of *code_start*. The modify register M is loaded with the value one which causes the value in the I register to be incremented by one after each memory access. This cycles through sequential addresses starting at *code_start*. After the ADSP-2105 has received each instruction it is loaded into the program memory position pointed to by the I register.

Once each instruction has been loaded the instruction counter (*ins_count*) is decremented by one. When the instruction counter is zero (i.e. the whole program has been downloaded) a jump is made to the start of the downloaded program and it begins execution.

To test this code a program was downloaded which again flashed the LED connected to the Flag_Out pin on the ADSP-2105 (See Appendix B, page 237). However this program was slightly different to the previous example as the timer interrupt has to be loaded explicitly into the interrupt table (i.e. the monitor program must be overwritten).

This is achieved by declaring a label in the *flash* program which points to the instruction *JUMP flash* (flash is the routine which flashes the LED) and then declaring a pointer to this instruction. The value pointed to by the pointer (i.e. the bytes which make up the instruction *JUMP flash*) is then loaded into a register. The contents of the

register are loaded into the timer interrupt position in memory. This causes an interrupt to the routine *flash* every time the timer times out (i.e. the light flashes).

To download a file from the host the data must first be extracted from the Intel Hex Format file produced by the PROM splitter and then sent byte at a time to the ADSP-2105. The FORTRAN program which achieves this is shown in Appendix B page 238.

4.3.6 Software for control processor

4.3.6.1 Basic Procedure

When the source node decides it wants to communicate with the destination node, the system level software on the source node scans the links on the node for a free link to communicate with the destination node. Once a free link is found the source node sends its connection request (consisting of three bytes) to the ADSP-2105 (See Figure 4.30).The first byte contains the address of the source node and the second byte contains the link number on the source node. The third byte holds the address of the destination node. This protocol can be expanded for more processors by using two bytes for the addresses of the source and destination processors.

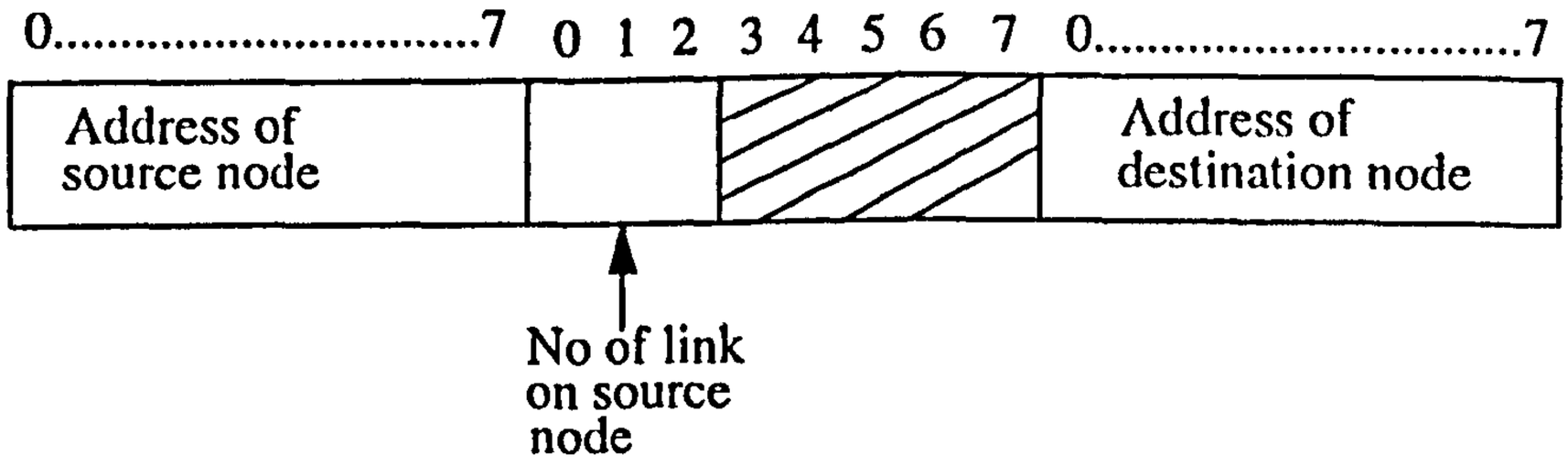


FIGURE 4.30. Connection request sent by node

The control processor has a table in memory which contains the connections from the nodes to the crossbar switch and a flag to indicate whether the connection is already in use (See Figure 4.31). When a connection request is received the ADSP-2105 scans the table to find the link on the crossbar switch that the source node is connected to. It then scans the table looking for a free link on the destination node and if one is free makes the connection on the crossbar switch which connects the source node to the

destination node. The flags in the connection table are then updated and an acknowledge is returned to the source node.

ARRAYS

Node No. [32]	Link No On Node [32]	Link No On Crossbar [32]	Connection [32] Used/Unused
0	0	10	1
0	1	25	0
0	2	12	0
0	3	30	1
1	0	8	1
.....	.....	.....	.....

FIGURE 4.31. Connection Table in Control Processor

The format of the acknowledge byte is shown in Figure 4.32. The link number of the destination node is sent in order to allow the source node to make disconnection requests (more on this later). If the value of the byte returned is greater than the number of links on the destination node (i.e greater then four for a transputer), this signifies to the source node that the connection could not be established. The byte is returned via an octal bus transceiver rather than the FIFO as only one byte is returned to the requesting node. Data from the source node to the control processor is therefore transferred via the FIFO and data is returned to the source node via an octal bus transceiver.

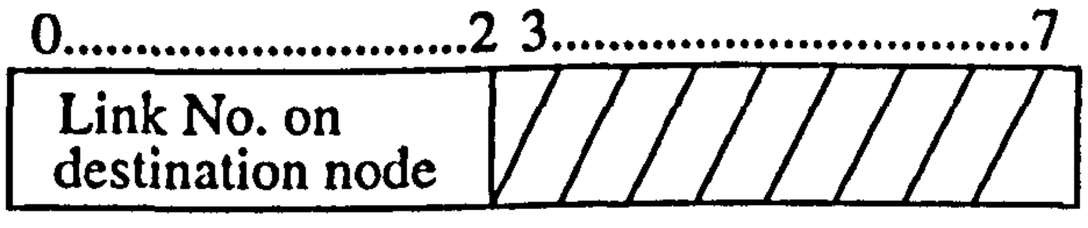


FIGURE 4.32. Acknowledge Byte returned to source node

Once a node receives a message indicating its connection request has been honoured it is free to send data via the crossbar switch to the destination node. The source node knows when the message has been successfully received by the destination node due to the link acknowledge protocol used by INMOS OSLinks. When the data has been completely transferred then the connection can be broken. The format of the disconnection request made by the source node is shown in Figure 4.33.

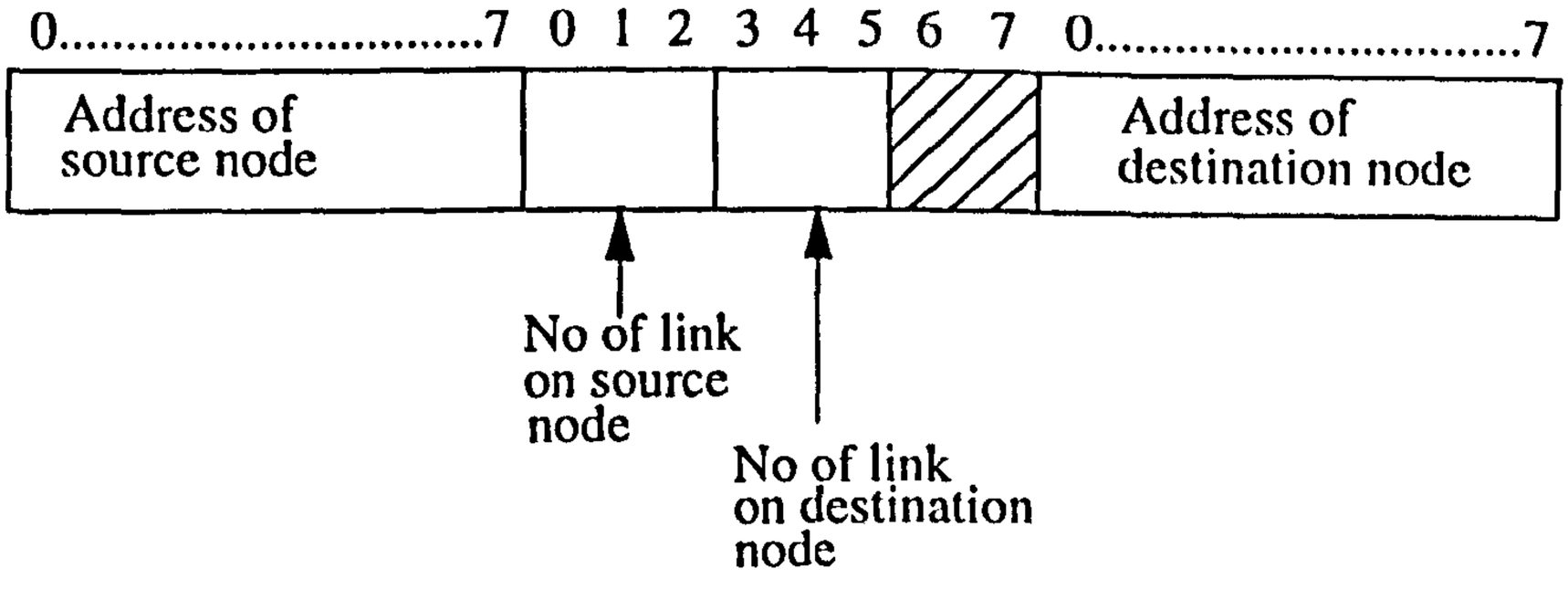


FIGURE 4.33. Disconnection Request

The message is basically the same as a connection request, except that the number of the link on the destination node is sent as well. The reason for this is that in the case where two nodes are connected by two or more links then the link numbers need to be specified in order to disconnect the correct link.

The control processor can distinguish between connection and disconnection requests by looking at the value of the second byte. If it is greater than the number of links on the source node then the request must be a disconnection request (i.e it contains the address of the destination node).

4.3.6.2 Program Structure

A diagram of the structure of the program which receives the incoming data from the nodes and programs the crossbar switch is shown in Figure 4.34 on page 140. The program consists of several modules which are linked together by the Linker to form the executable file. Appendix B, pages 239 - 251 contains the source code listings for these programs.

When developing the software for the ADSP-2105 it was assumed that the crossbar switch was an INMOS C004. Since in the case of a transputer all four links can be connected to the crossbar switch, 8 transputers can be fully connected by using a C004. A 64-way crossbar switch such as the LSILogic L64270[13]could be used which would allow 16 transputers to be fully connected. If more transputers were required then the number of crossbar switches could be increased. Each crossbar switch would have a C012 connected to it and the C012 would be addressed by the control processor. Effectively each crossbar switch would have a unique address.

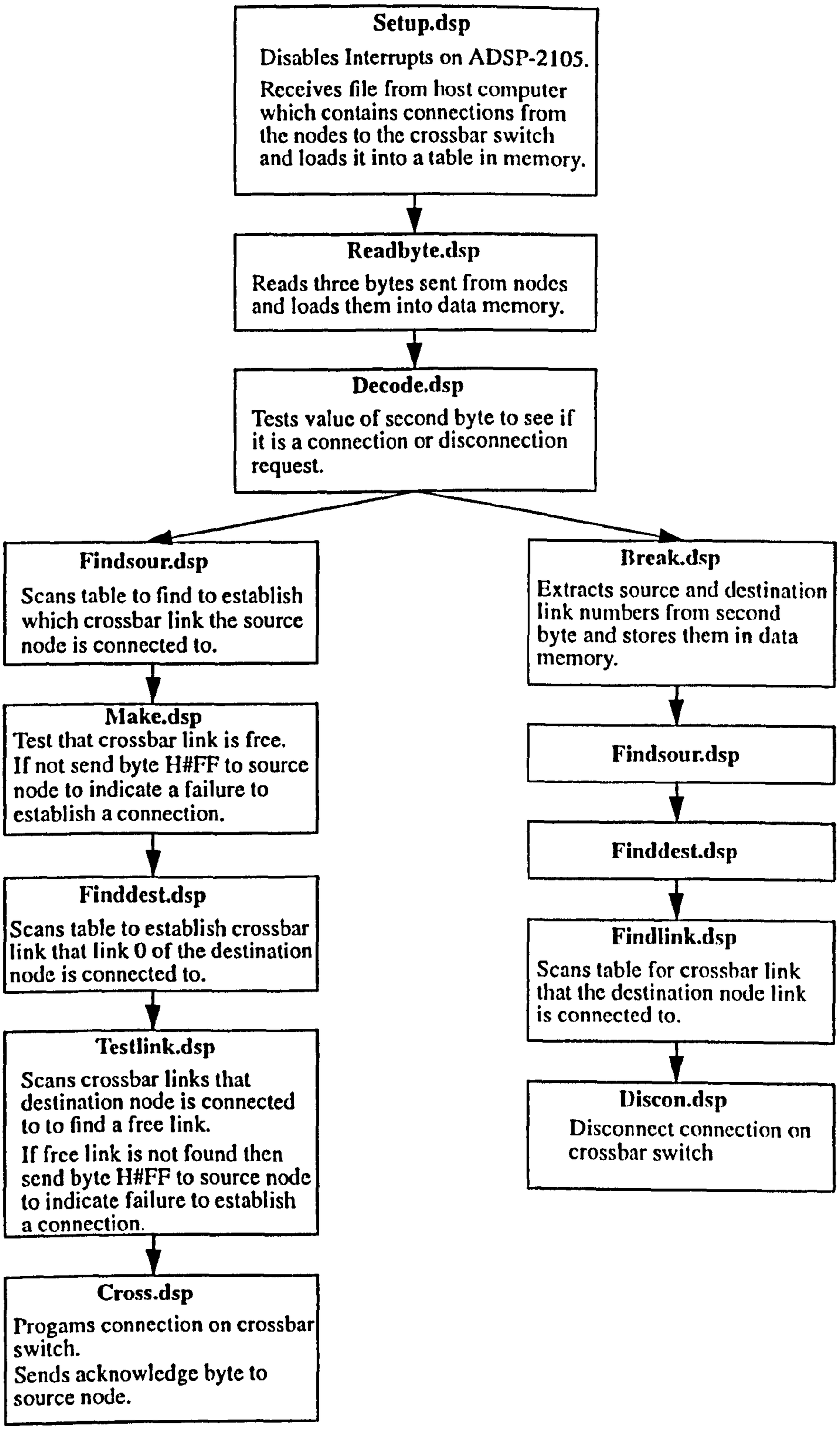


FIGURE 4.34. Program structure for ADSP-2105 software

4.3.7 Testing of overall procedure

A mock set up of the dynamic circuit-switched network using ports on a PC to emulate the nodes was constructed (See Figure 4.35 on page 142). The token passes between the two boards plugged into the PC slots and data to represent the three byte message from the nodes is written into the FIFO from the PC.

The connection requests are clocked out of the C011 to the control processor board via a serial line. The C012 which would normally be connected to the crossbar switch on the control processor board is actually connected to a link adapter board plugged into the PC. This allows the PC to read the messages normally meant for the C004 and check they are correct.

4.3.7.1 Test circuits

The circuit diagram for the PC plug in cards which emulate nodes is shown in Figure 4.36 on page 143. Two PALs (P22V10L) are used in this circuit: one for address decoding and the other for the token passing and fifo clocking state machines. A buffer ('245) is inserted between the PC data bus and the FIFO and C011 data buses. The '74 dual D-type positive edge triggered flip-flop is required to generate the **Holdtoken** and **QAck** signals. A photograph of the circuit board is shown in Appendix D, Figure 6 on page 306.

A functional block diagram of the FIFO is shown in Figure 4.37 on page 144. It is organised as a 1024 x 9 RAM with asynchronous and simultaneous read and write. The reads and writes are internally sequential through the use of ring pointers, with no address information required to load and unload data. Data is toggled in and out of the device through the use of the Write (W*) and Read (R*) pins.

The CUPL source code for P22V10L(0) which performs the address decoding is shown in Figure 4.39 on page 146. This code is similar to the address decoding for the circuit to test the clocking of the bytes out of the FIFO. The main difference here however, is that data must be read from the C011 (i.e. effectively the acknowledge byte from the control processor) and also the token must be passed on.

In order to read data from the C011 the **QAck** and **QValid** signals are controlled by a D-type flip-flop. The connections to the flip-flop, the truth table for the flip-flop and a timing diagram for the circuit are illustrated in Figure 4.38 on page 145.

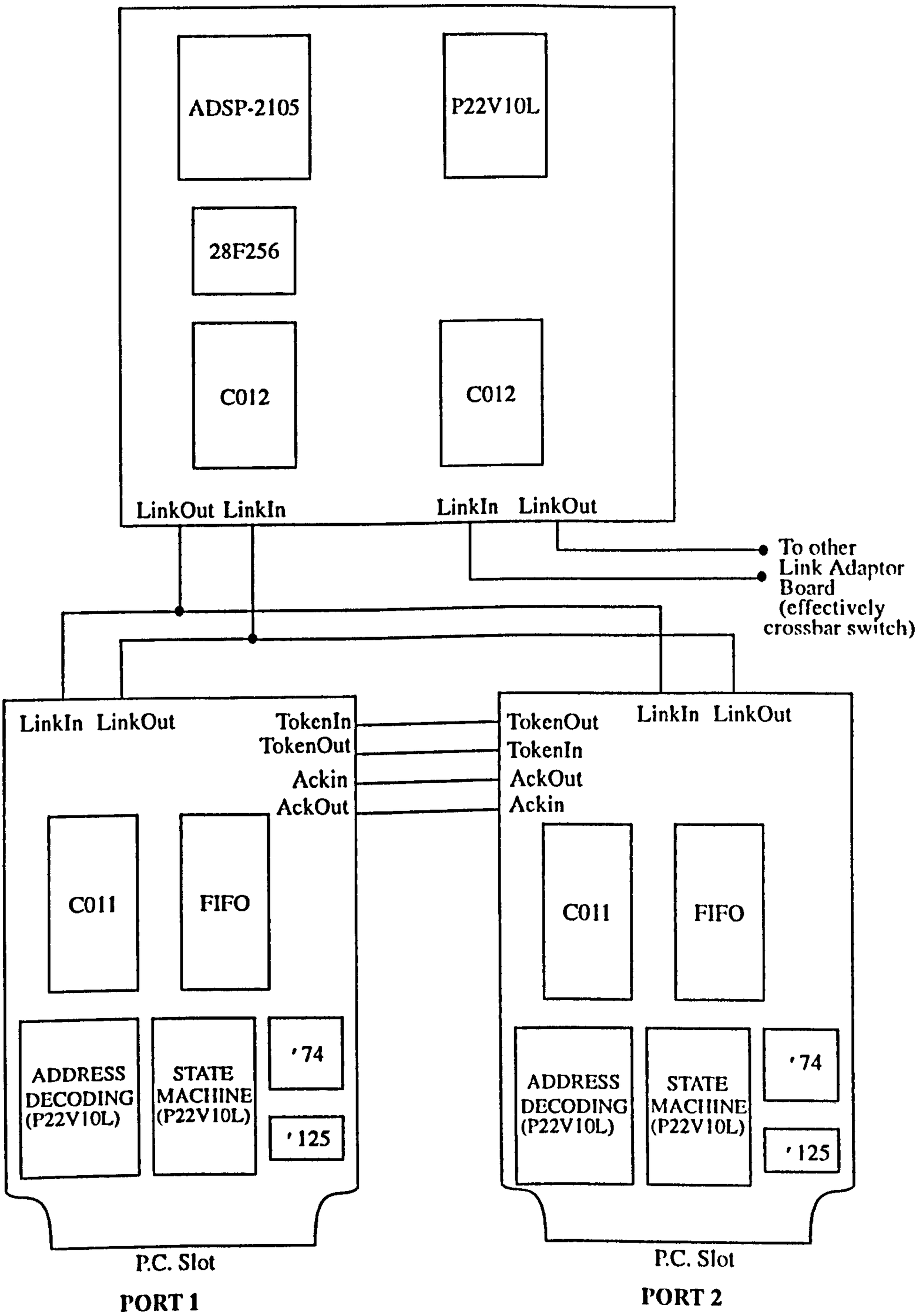


FIGURE 4.35. Set-up used to test theory of dynamic connection network

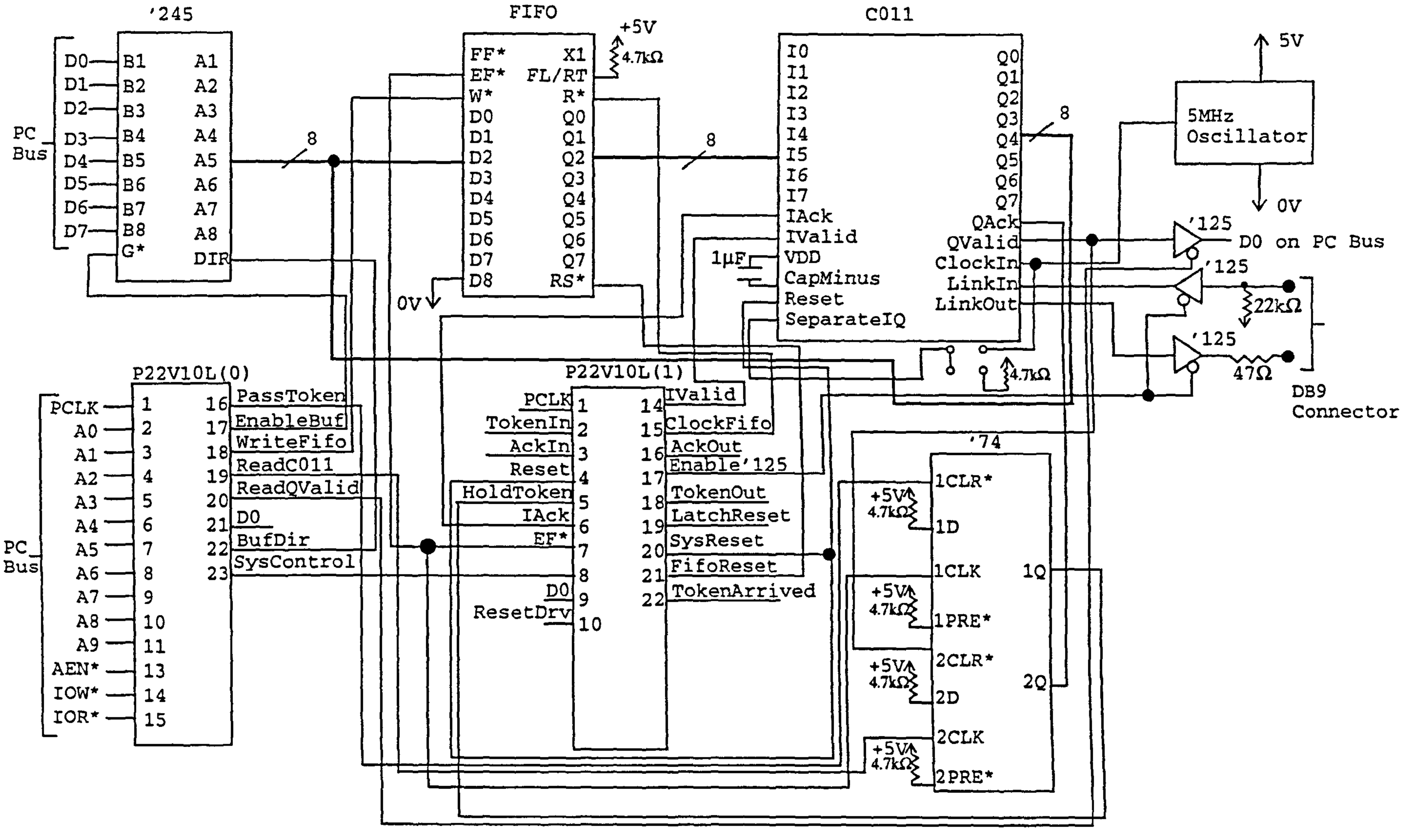


FIGURE 4.36. PC plug-in card which emulates node

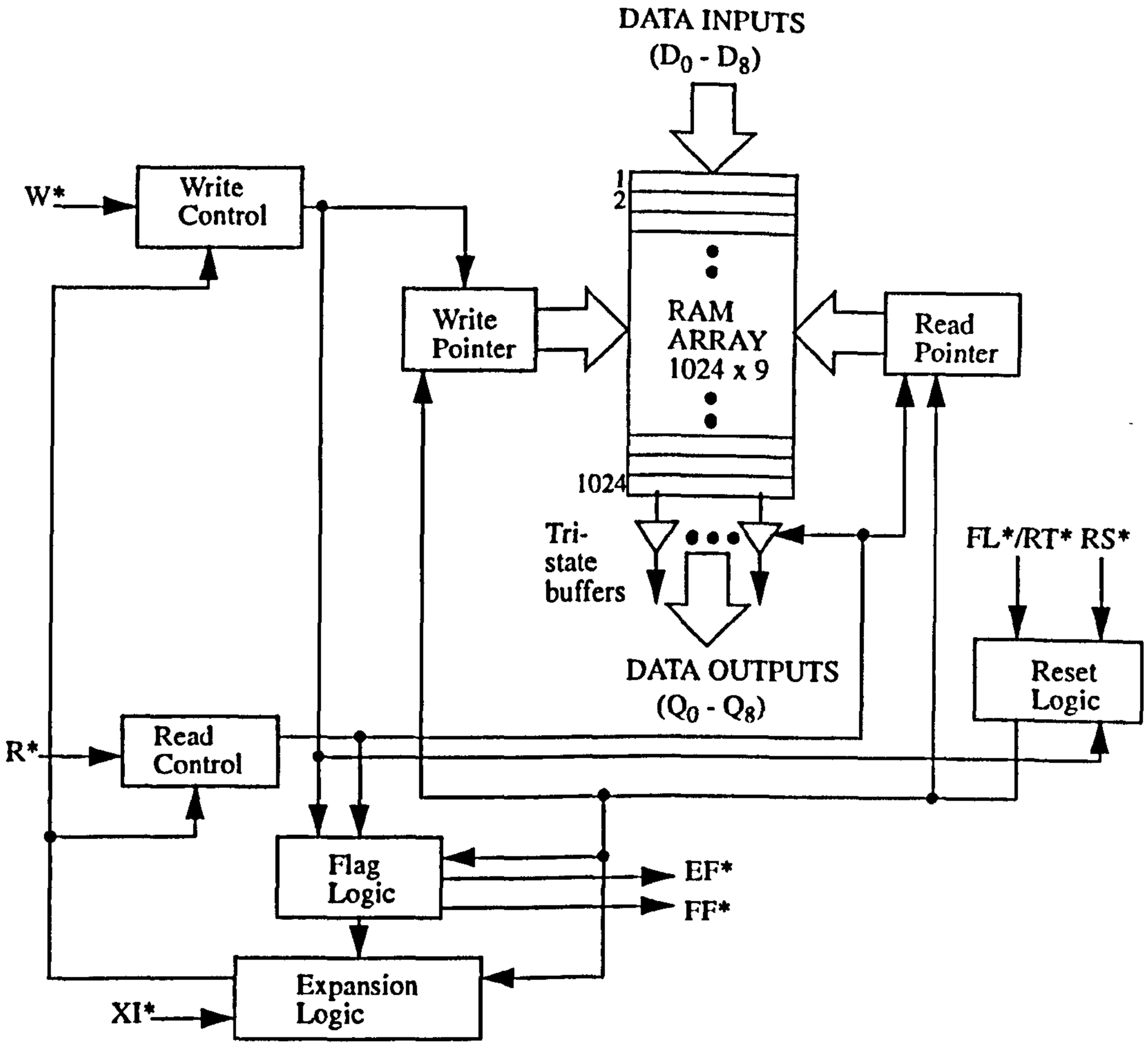
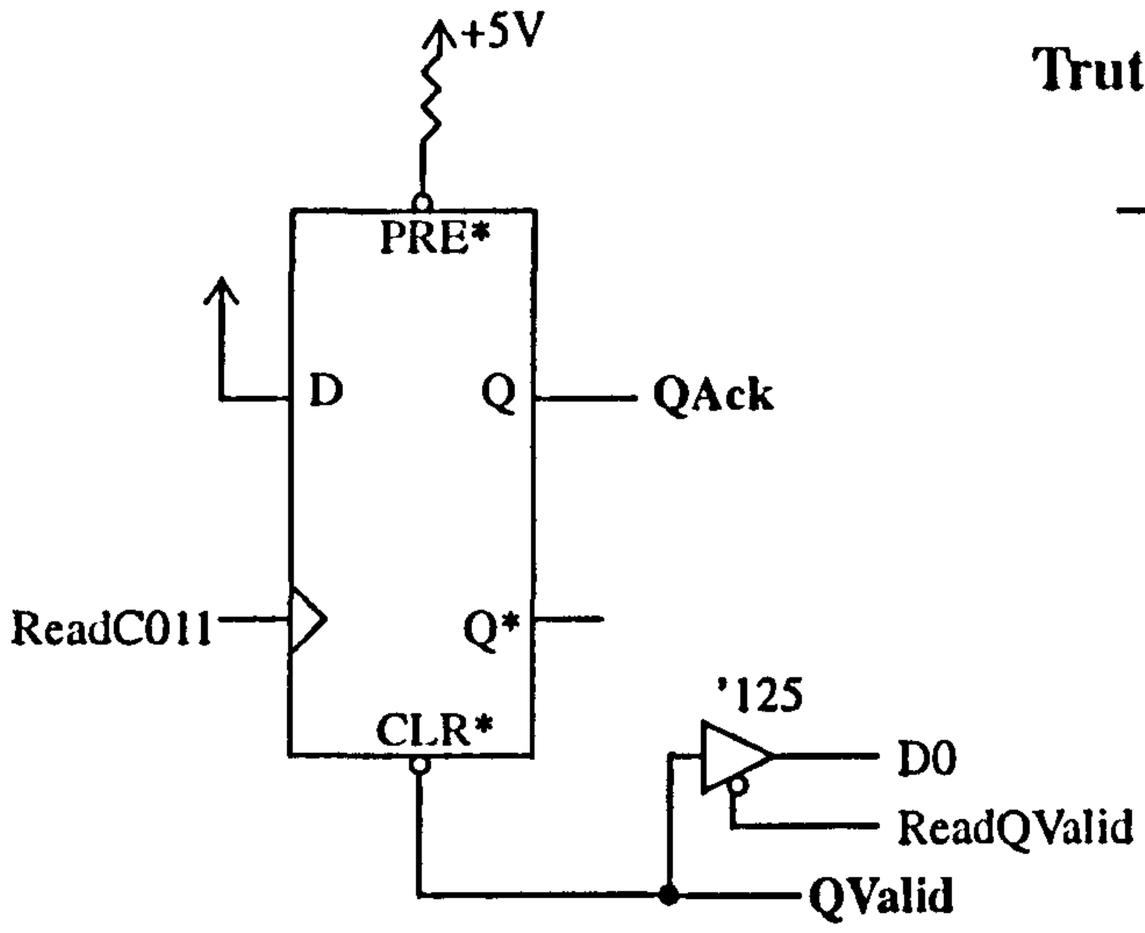


FIGURE 4.37. Functional Block Diagram of FIFO

The node (PC port in this case) monitors the state of the **Qvalid** signal via the '125 buffer. This buffer is enabled by the **ReadQvalid** signal from the P22V10L(0). When the **QValid** signal is pulled high (indicating the C011 has received data) the node (PC port) initiates a read cycle.

The signal **ReadC011** from P22V10L(0) is used to clock the flip-flop. Therefore at the end of a read cycle (i.e. on the rising edge of ReadC011) this transfers the value on the D input of the flip-flop to the Q output. Since the D input is tied high this transfers a logic high to the Q output which is connected to **QAck** on the C011. This indicates to the C011 that the write cycle is finished. The C011 then sends an acknowledge to the sending device and returns **QValid** low.



Truth Table for D-type Register

PRE	CLR	CLK	D	Q	Q*
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H	H
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	Q_0*

Timing Diagram

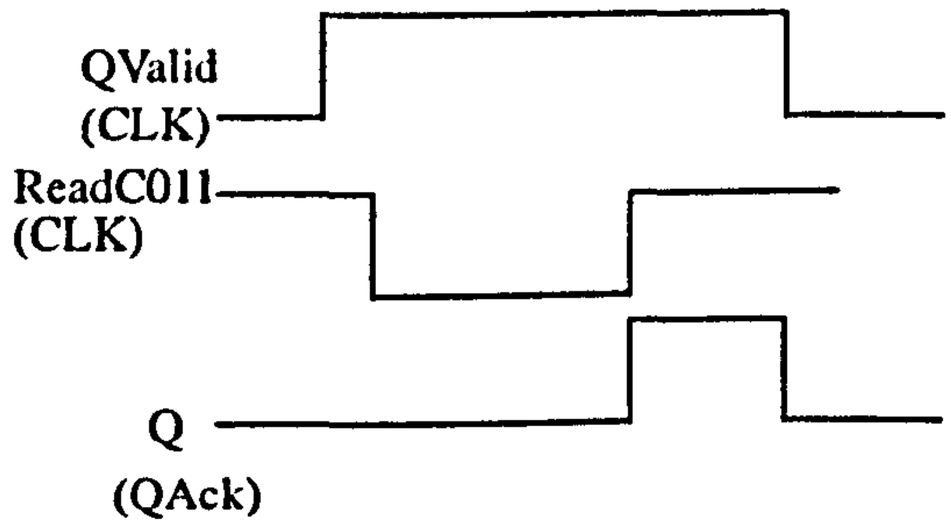


FIGURE 4.38. Control of QAck and QValid

To clear the D-type flip-flop which controls the **Holdtoken** signal (see Figure 4.14 on page 113) and thus release the token, the signal PASSTOKEN is generated by the P22V10L(0). This signal is produced by writing a value of one to the address H#102. The token is passed on once the node receives the acknowledge byte from the control processor. The signals to write data into the FIFO and to control the buffer are similar to the PC interface designs discussed previously.

P22V10L(1) runs the FIFO clocking and token passing state machines. The CUPL source code for this PAL is shown in Appendix B pages 252-253. The two state machines run separately and an output from the token passing SM called **TokenArrived** signals to the FIFO clocking state machine when the token is there (i.e. as shown in the state diagram for the FIFO clocking).

```
/**INPUTS**/

PIN 1           =    PCLK;
PIN [2..11]     =    [A0..A9];
PIN 13          =    !NOTAEN;
PIN 14          =    !NOTIOW;
PIN 15          =    !NOTIOR;
PIN 21          =    D0;

/**OUTPUTS**/

PIN 16          =    !PASSTOKEN;
PIN 17          =    !ENABLEBUF;
PIN 18          =    !WRITEFIFO;
PIN 19          =    !READC011;
PIN 20          =    !READQVALID;
PIN 22          =    BUFDIR;
PIN 23          =    SYSCONTROL;

/**DECLARATIONS AND INTERMEDIATE VARIABLE DEFINITIONS**/

FIELD ADDRESS =    [A9..A0];
FIFO            =    ADDRESS:[100] & NOTAEN;
C011            =    ADDRESS:[101] & NOTAEN;
TOKEN           =    ADDRESS:[102] & NOTAEN;
QVALID          =    ADDRESS:[103] & NOTAEN;
CONTROL         =    ADDRESS:[105] & NOTAEN;

PASSTOKEN       =    TOKEN & NOTIOW & D0;
WRITEFIFO       =    FIFO & NOTIOW;
READQVALID      =    QVALID & NOTIOR;
READC011        =    C011 & NOTIOR;
ENABLEBUF       =    WRITEFIFO # READC011;
BUFDIR          =    NOTIOR;
SYSCONTROL      =    CONTROL & NOTIOW;
```

FIGURE 4.39. CUPL source code for address decoding

This PAL is also used to generate the reset signals for the C011 and FIFO. To reset both these chips a binary '1' is sent to the SYSCONTROL address.

This mock set-up functioned successfully. The main difference between this set-up and a 'real' system is the interface between the node and FIFO. Obviously for various types of bus interface slightly different programming on the PAL may be required.

4.4 Connection Request Service Time

The four major factors involved in the time taken to service a request are:

- the time required to pass the token (0.15µs in a 2 node system)
- the time taken to clock the bytes out of the FIFO (0.15µs)
- the time to transfer the bytes from the C011 to the Control Processor (1.2µs)
- the time required to program the crossbar switch (1.2µs)

If the PALS are being clocked at 20MHz and the INMOS OSLink is operating at 20Mbits/s then the connection request service time is approximately 2.7µs minimum. Obviously this number will be larger for a greater number of nodes as the token will have further to travel and also it will vary depending on how many connection requests are to be sent to the control processor. The service time could be speeded up by using a faster token passing clock and control processor.

4.5 Conclusions and Discussion

A system has been described which allows links between nodes to be established on-demand during program run-time. All the ICs employed in the design are commercially available at relatively low cost. The control processor used achieves a much higher performance than a transputer (commonly used as a control processor) enabling it to process connection requests much faster. All the valuable communication links on the node are free for interprocessor communication and are not tied up with control information.

This cost-effective method provides deadlock free, low message latency, dynamic reconfigurability. This is especially useful in time critical applications which transmit and receive large volumes of data such as robotics and image processing. The hardware subsystem used to send data to the control processor can be used with any processor providing they possess a high speed communication mechanism.

Unfortunately it was not possible to test the system running an application. Jones[3] showed however that for an N-body simulation a dynamic switching system can out-perform a static message passing system. With a small number of simulated bodies, the performance of the dynamically switched version is poor compared to the static topology version (a ring). However, as the number of simulated bodies increases, the

performance of the dynamically switched version ultimately exceeds that of the static version.

The hardware Jones was using to implement dynamic switching was not built specifically for this purpose. Only two of the four links on the nodes could be dynamically reconfigured and the monitoring bus connected to the control processor was not designed to be fast. The results should be far better using the circuit-switching scheme described in this chapter. They do show however that where large volumes of data have to be transferred dynamic switching can give greater performance than a static system using message passing.

Dynamic switching schemes do not however out-perform systems using the T-9000 transputer and the C104 router[15]. The T-9000 is a second generation transputer which uses DS-Links operating at 100M/bits per second for communication. A major feature of the DS-Link is that it provides a physical connection over which any number of software (or 'virtual') channels may be multiplexed; these can be either be between two directly connected devices, or can be between any number of different devices, if the links are connected via (packet) routing switches.

The OS-Links described previously use only two channels, one in each direction. To map a particular piece of software onto a given hardware configuration the programmer had to map processes to processors within the constraints of available connectivity. The problem is illustrated in Figure 4.40 where 3 channels are required between two processors, but only a single link connection is available.

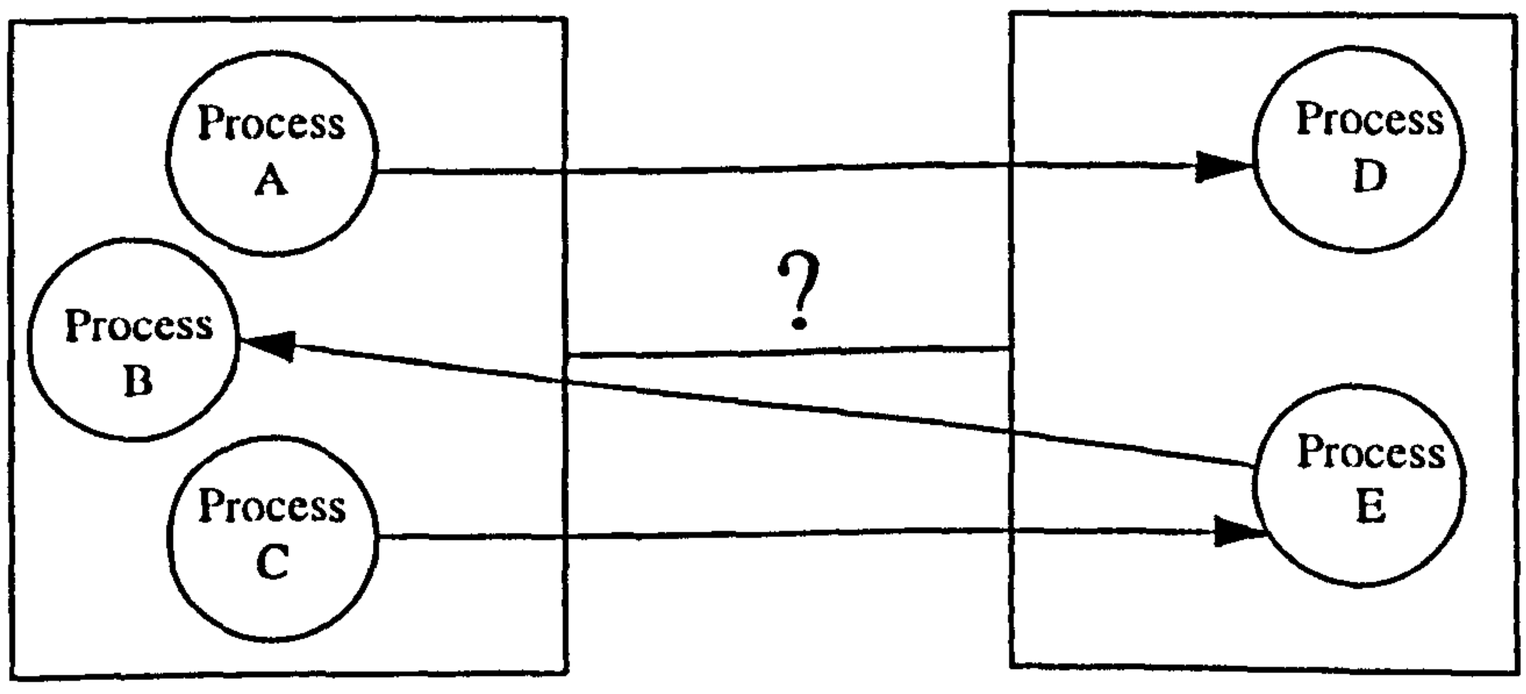


FIGURE 4.40. Multiple communications channels required between devices

This problem is solved with the T-9000 as it uses multiplexing hardware to allow any number of processes to use each link, so that physical links can be shared transparently. These channels which share a link are known as 'virtual channels'.

With DS-Links each message is divided into packets (32 bytes long). Every packet requires a header to identify its channel. Packets from messages on different channels are interleaved on the link. There are two important advantages to this:

- Channels are, generally, not busy all the time, so the multiplexing can make better use of hardware resource by keeping the links busy with messages from different channels.
- Messages from different channels can effectively be sent concurrently - the device does not have to wait for a long message to complete before sending another.

DS-Links can be used to connect devices directly together or they can be connected to a router device known as a C104 to route messages across a network. As the DS-links allow all the virtual channels of a device to use a single link, complete system wide connectivity can be provided by connecting just one link from each device to the routing network.

The IMS C104 is a full 32 x 32 non-blocking crossbar switch, enabling messages to be routed from any of its links to any other link. In order to minimize latency, the switch uses wormhole routing in which the connection through the crossbar is set up as soon as the header has been read. The header and the rest of the packet can start being transmitted from the output link immediately.

The header of each packet is used to determine the link to be used to transmit the incoming packet. This is done by a set of 32 registers associated with each link.

It is possible using 48 C104s to connect 512 nodes with only 3 routing delays. By using a special purpose router this requires no additional software on the nodes to route messages and also no additional buffering is required on the nodes to buffer the packets. The T-9000 and the C104 therefore provide an efficient fast low message latency communication system.

Even though systems using the C104 router and T9000 would be faster than the set-up described in this chapter, the latter is still more cost effective. The C104 is much more expensive than a C004. Also there are still many systems around which use T-800

transputers or other processors such as the C40. These systems could benefit from the work described in this chapter.

It is also possible that there will be some applications in which synchronisation constraints require the direct connection of processor pairs, possibly in cases when the uncertainties of message transit times in wormhole routing is unacceptable. In such a scenario, dynamic reconfiguration offers a more dependable mode of communication once a connection has been installed.

Software[16,17] has been developed for the T-800/400 transputers that emulates virtual channel routing. This however uses some of the computational power of the node for communication and also has a large message latency (~30μs). The system described in this chapter is therefore much more efficient as no additional software is required on each node and the message latency is much lower.

The design presented in this chapter provides an efficient and fast mechanism for dynamic on-demand circuit switching. It could however be enhanced by using a faster token passing clock and control processor.

References

[1] Tudruj, M., Kalinowski, T. Multi-Transputer Systems with Dynamic Link Connection Switching Controlled through a Serial Bus. *Transputer Applications and Systems '93*. Proceeding of the 1993 World Transputer Congress, Aachen, Sept. 93, pp. 803-818

[2] Tudruj, Marek. Multi-transputer architectures with the look-ahead dynamic link connection reconfiguration. *Transputer Applications and Systems '95*. Proceeding of the 1995 World Transputer Congress 1995, Harrogate, Sept. 95, pp. 52-69

[3] Jones, P. The Implementation of a Run-Time Link-Switching Environment for Multi-Transputer Machines. *Proceedings of the NATUG 2 Meeting*, Durham, Oct. 1989, pp. 107-122

[4] Murta, Alan. Support for Network-Wide Synchronous Communication via the Active Reconfiguration of Transputer Links. *Transputer Applications and Systems '93*. Proceeding of the 1993 World Transputer Congress, Aachen, Sept. 93, pp. 772-773

[5] Jin, Lan et al. Dynamically Reconfigurable Architecture of a Transputer-Based Multicomputer System. *Proceedings of the 20th international conference on parallel processing*, Vol.1, 1991. pp. I-475 - I-478

[6] Calvez, Jean Paul., Pasquier, Olivier. A Transputer Interconnection Bus for Hard Real-Time Systems. *Transputers '92*. Conference Proceedings, Besancon, France, May 20-22, 1992, pp. 273-283

[7] COM20020 ULANC Manual, *Standard Microsystems Corporation.* 1993

[8] Bissland, Lesley., White, David N. J. A Circuit-Switched Network for INMOS OSLinks. *Transputer Research and Applications 7*. Proceedings of the Seventh Conference of the North American Transputer Users Group, Atlanta, Oct. 1994, pp 133-141

[9] Ingle, Vinay K., Proakis, John G. *Digital Signal Processing Laboratory.* Analog Devices, Prentice Hall, Englewood Cliffs, 1991, ISBN 0 13 218181 9

[10] Analog Devices. *Technical Notes on ADSP-2105.*

[11] ADSP-2100 Assembler Manual, 1991.

[12] McGuire, Gerald. *Loading and ADSP-2101 Program via the Serial Port.* Analog Devices Application Note AN-243.

[13] LSI Logic Corporation 1989. *L64270 Preliminary Data*

[14] Integrated Device Technology High Performance CMOS Data Book 1988. *CMOS Parallel First-In/First-Out FIFO 1024 x 9-Bit*, pp 6-14 - 6-26

[15] May, M.D., Thompson, P.W., Welch, P.H. *Networks, Routers and Transputers.* IOS Press, 1993.

[16] Debbage, Mark., Hill, Mark B., Nicole, Denis A. The Virtual Channel Router. *Transputer Communications*, 1993, **Vol. 1**, pp 3-18

[17] Wabnig, Harald W. Virtual Channels for Deadlock-Free Communication in Transputer Networks. *Transputer Applications and Systems '93*. Proceeding of the 1993 World Transputer Congress, Aachen, Sept. 93, pp. 1035-1051

Part 2

Chapter 5

Molecular Mechanics

In recent years with the increase in computational power it has become possible to predict the chemical properties of a molecule or interactions between molecules by computational methods. This has had increasing application in drug design where the behaviour of a novel drug can be predicted by computer simulation.

One of the techniques used in this process is molecular mechanics. This chapter describes the mathematical expressions used to construct molecular mechanics calculations.

5.1 Introduction

5.1.1 What is Molecular Mechanics

Molecular mechanics(MM) is a computational method designed to give accurate structures and energies of molecules[1-4]. These properties can be determined by experimental methods such as x-ray crystallography, microwave and vibrational spectroscopy etc. These procedures however rely on having the material or crystal available. Computational methods avoid this problem and therefore can be used to predict the structure and energy of molecules that have not even been synthesised.

Molecular orbital (MO) techniques (another computational method) determine the structure of a molecule by the approximate solution of the Schrödinger equation for a given nuclear configuration, followed by a systematic adjustment of this configuration to minimise the energy of the molecule. The Born-Oppenheimer approximation is assumed which allows the nuclear and electronic motions within an atom to be separated. MO methods regard the nuclei as stationary while the electrons move relative to them.

The theoretical basis of the molecular mechanics method can be derived by taking an alternative approach to the Born-Oppenheimer approximation: in this case the nuclear motion is considered while implying a fixed electron distribution associated with each atom.

A molecule from this perspective is considered to be a collection of masses (nuclei) that are interacting with each other via (almost) harmonic forces (bonds), and it is rather analogous to a system composed of weights joined together by springs (a ball-and-spring model). Potential energy functions are used to describe the interactions between nuclei. Force constants of the springs are represented by a collection of mathematical parameters. The equations and parameters that define the energy surface of a molecule are referred to as the force field.

The origin of this method lies in vibrational spectroscopy, where the information derived from analyses of vibrational spectra required the development of potential functions to describe the overall molecular behaviour. Two different approaches were considered.

In the first, the Central Force Field (CFF)[5] method, the molecular vibrations were fitted to a function which was the sum of pairwise interactions, without reference to the covalent structure of the molecule. A disadvantage to this approach is that although such a description is correct in terms of a quantum mechanical model of a molecule, it lacks the intuitive link with structure that is required for molecular mechanics. It has also been shown to give poor results in molecular mechanics calculations.

The second method, the Valence Force Field (VFF)[5], provides a description in which the vibrational data is fitted to a potential function consisting of bond length, bond angle and torsion angle dependent terms. This is much more satisfactory than CFF and has the advantage of allowing comparisons between molecules (CFF is very molecule specific).

The major criticism of the VFF method is that the force constants produced must attempt to incorporate intramolecular interactions such as dispersion forces which result from electron correlation, and therefore are not simply a representation of the intrinsic vibrational frequency.

5.1.2 Why Molecular Mechanics

Molecular mechanics calculations are a computationally intensive task, however, they are still faster than molecular orbital methods such as the *ab initio* calculation.

The time for running an *ab initio*[2] calculation increases as approximately n^4 where n is the number of orbitals, whereas for molecular mechanics it increases as N^2 where N is the number of atoms. This allows molecular mechanics calculations to be used with large molecules such as proteins where MO calculations would be impractical.

Molecular mechanics does of course have its disadvantages. It is an empirical method, and is based on a large volume of experimental data. This data must exist for a given class of compounds before the method can be developed and applied to any particular compound in that class. On the other hand the *ab initio* method is only concerned with nuclei and electrons, few additional parameters are required. This makes it more generally applicable. However, MM is also more accurate than MO within its sphere of application.

5.2 Formulation of Molecular Mechanics

One of the fundamental principles of molecular mechanics calculations is that the total energy of a molecule can be divided into readily identifiable parts. The energy is calculated as a sum of the steric and non-bonded interactions present. Therefore each bond length, angle and torsion angle is treated individually while non-bonded interactions represent the influence of non-covalent forces.

The equation to calculate the total steric energy of a molecule (V_s) is thus given by:-

$$V_s = V_l + V_\theta + V_\omega + V_r + V_q$$

Steric Energy Equation (EQ 5.1)

where V_l represents the summation over all the bonds in the molecule of the individual potential energies due to bond stretching or compression, and V_θ, V_ω ,V_r , and V_q represent similar terms for angle bending, bond torsion, van der Waals interactions, and coulombic interactions respectively.

A more refined force field will also consider interactions or cross terms such as stretch-bend, torsion-stretch, etc. These are usually small, and they can be neglected in the first approximation. Other ad hoc terms such as out of plane bending of planar atoms types have been used to take into account phenomena that are not properly accounted for otherwise.

5.2.1 Bond Stretching

The typical vibrational behaviour of a bond is near harmonic close to its equilibrium distance but shows dissociation at longer bond lengths (See Figure 5.1 on page 157). It is most accurately described by the Morse function[6]

$$V_l = \sum D_e \left[1 - \exp\{-\alpha (l - l_o)\} \right]^2$$

Morse Function (EQ 5.2)

where l_o is the equilibrium bond length, l is the actual bond length, D_e the dissociation energy, and α a force constant. The exponential calculation is computationally intensive therefore most force fields have adopted a simple harmonic function

$$2V_l = \sum_l k_l (l - l_0)^2$$

(EQ 5.3)

where k_l is the stretching force constant, l_o is the equilibrium bond length and l is the actual bond length. The bond is effectively treated as a stretched spring. This equation only approximately describes the actual behaviour of the bond. At extended bond lengths it is much too steep (see Figure 5.1 on page 157) while it provides no representation of dissociation at very large deformations. In order to reproduce the Morse curve more accurately in the region where bonds are considerably stretched a cubic term is sometimes added to the previous expression[2]. i.e.

$$V_l = \sum_l k_l (l - l_0)^2 - k_l \left| (l - l_0)^3 \right|$$

(EQ 5.4)

Careful selection of the force constant for the cubic expression gives accurate treatment of bond length deformations in a wide variety of molecules. A problem with the cubic term however, is that as bonds are stretched to greater distances, the cubic

term will begin to dominate. At a critical point the curve reaches a maximum and the bond stretching energy then plummets downwards toward negative infinity. Attempts have been made to remedy this by adding a quartic term which reverses the inversion[7].

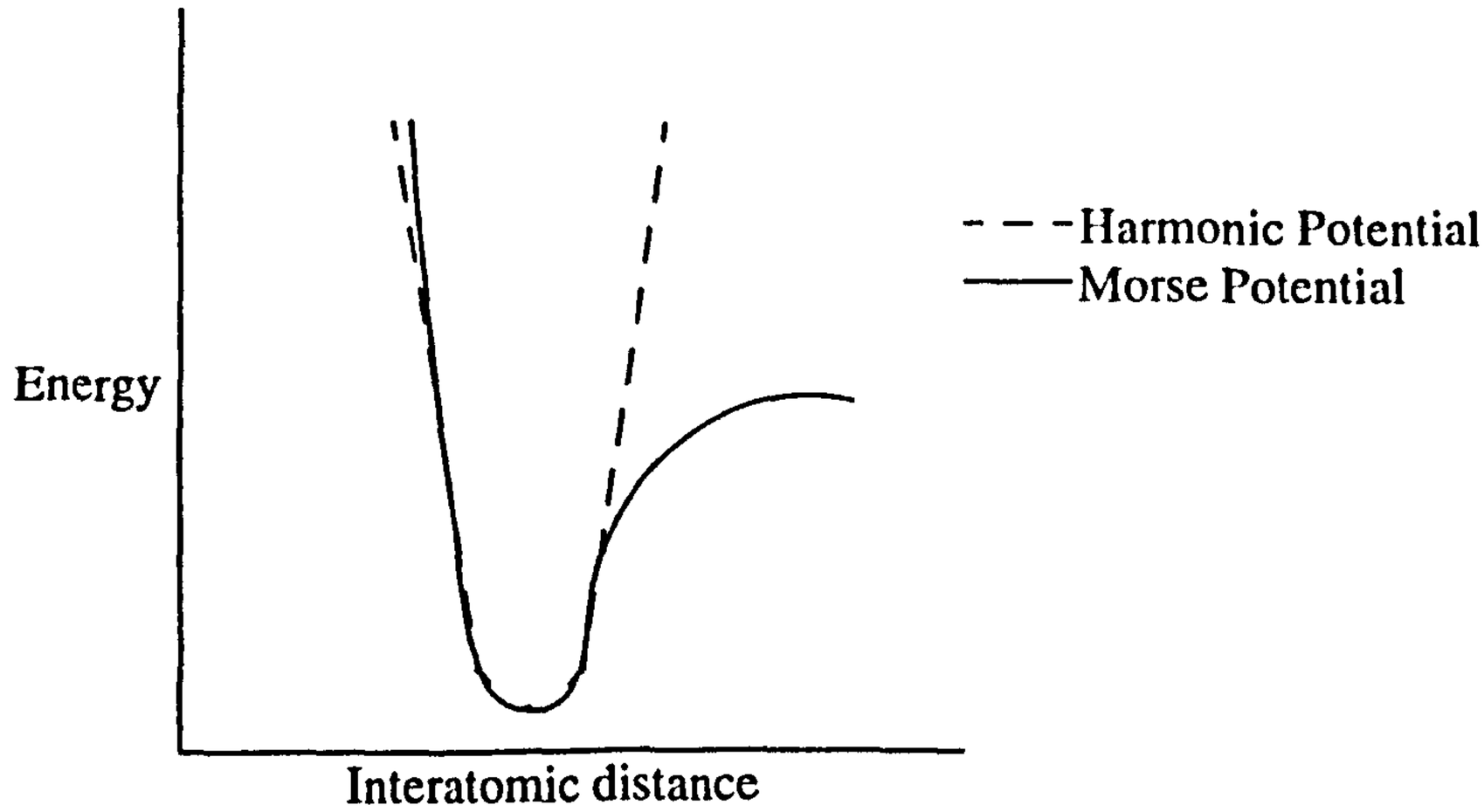


FIGURE 5.1. Curves showing the variation of bond stretch energy with distance

5.2.2 Angle Bending

Angle bending can also be described by a simple harmonic function.

$$2V_\theta = \sum_\theta k_\theta (\theta - \theta_0)^2$$

(EQ 5.5)

As before k_θ is the bending force constant, θ_o is the equilibrium bond angle and θ is the actual bond angle. This equation however is not very satisfactory as the force constants for angle bending are smaller than for stretching allowing greater distortion away from the strain free value.

The equation can be improved by adding a cubic term similar to the situation for bond stretching[16].

$$2V_\theta = \sum_\theta k_\theta (\theta - \theta_0)^2 - k'_\theta \left| (\theta - \theta_0)^3 \right|$$

(EQ 5.6)

This cubic term works well, except in the few cases where angle starts off being greatly deformed from the strain free value. Similarly to bond stretching the cubic term can

dominate at larger angles. An extra term can be added to prevent this from occurring which will reduce the effect of cubic term and force the angle back towards a more reasonable value. The equation for bond stretching is now[8]:-

$$2V_\theta = \sum_\theta k_\theta \left(\Delta\theta^2 - k'_\theta \left(|\Delta\theta^3| - 0.0004 |\Delta\theta^5| \right) \right)$$

(EQ 5.7)

where $\Delta\theta = \theta - \theta_0$, k_θ is the bending force constant and k'_θ is the anharmonic force constant. A fifth power is usually added as it can be calculated from the product of the square and cubic terms.

5.2.3 Torsion Angles

Initially the potential energy term due to bond torsion was calculated using an expression relating to the periodicity of the central bond. i.e.

$$2V_\omega = \sum_\omega V_n (1 + s\cos n\omega)$$

(EQ 5.8)

where V_n is the rotational barrier height, n is the periodicity of rotation (e.g. in ethane n=3; in ethene n =2), ω is the measured torsion angle and $s = +1$ for a staggered minimum (e.g. ethane) and $s = -1$ for an eclipsed minimum (ethene).

This equation is too simplistic for certain situations. Consider the torsion around the central C-C bond in butane. There are three kinds of torsion angle: H-C-C-H; C-C-C-H; and C-C-C-C. Whilst the periodicities of the first two are essentially threefold, the major component of C-C-C-C is onefold (i.e. the methyl-methyl eclipse occurs only once per 360°) with a minor threefold addition. In general the equation for torsional energy is written in the following form[8].

$$2V_w = \sum_\omega [V_n (1 + s\cos n\omega) + V_1 (1 + s\cos\omega)]$$

(EQ 5.9)

In most cases V_1 (the onefold component of the barrier to free rotation) is set to zero except for torsion angles such as C_{sp^3}-C_{sp^3}-C_{sp^3}-C_{sp^3} and C_{sp^3}-C_{sp^2}-Namide-C_{sp^3}.

5.2.4 van der Waals interactions

Many different equations have been used to describe the van der Waals interactions. The common one however is the Lennard-Jones 6-12 potential[9]

$$V_r = \sum_r [Ar^{-12} - Br^{-6}]$$

(EQ 5.10)

where A and B are constants that depend on the atom types (see Figure 5.2 on page 159.). The summation is over all 1,4 and higher unique pairwise non-bonded distances. Short range repulsions are accounted for by the r^{-12} term where London dispersion-attraction forces are resolved by the r^{-6} component.

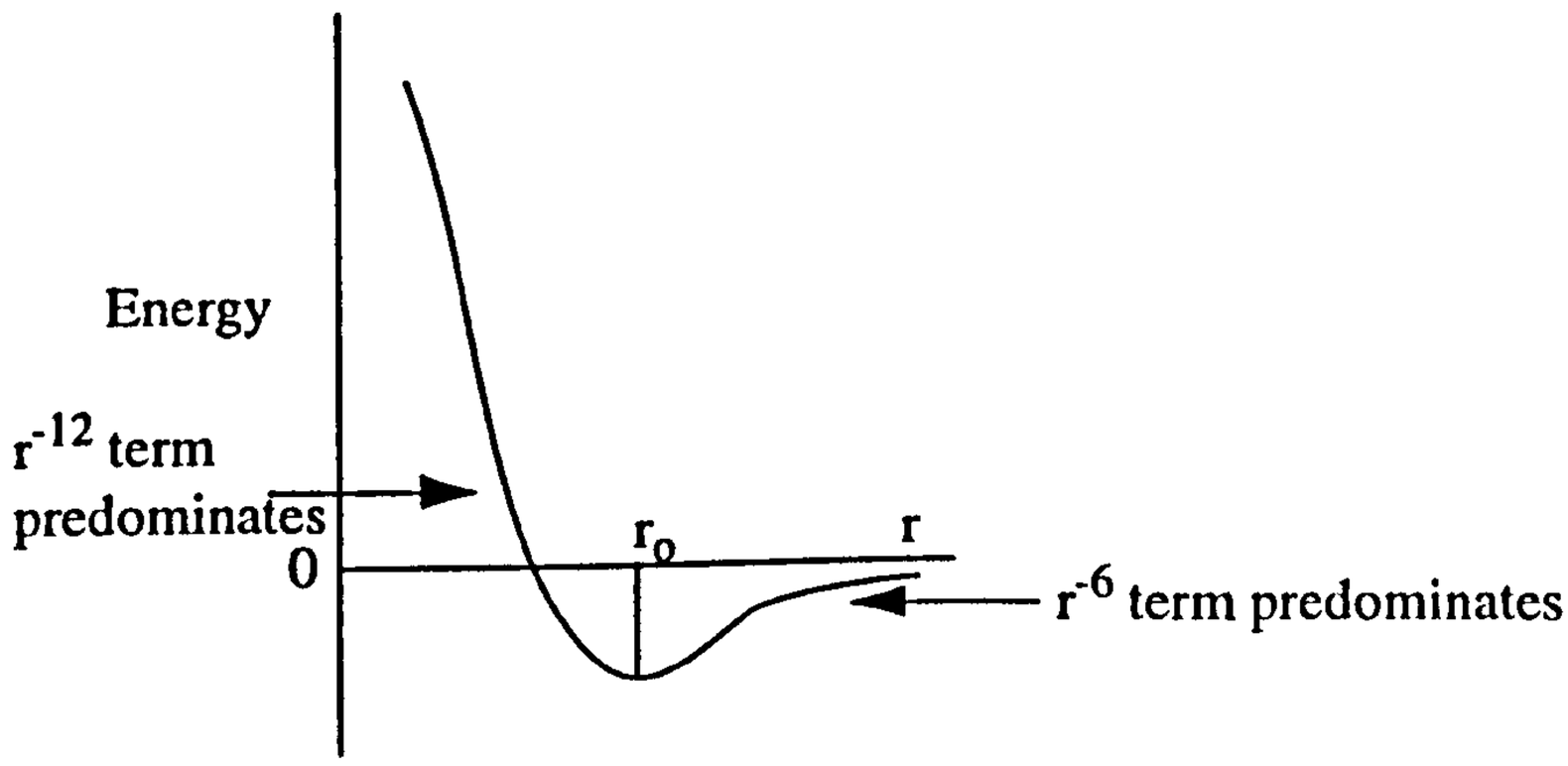


FIGURE 5.2. A typical van der Waals curve

In general the repulsive part of the Lennard Jones curve is too steep to describe interactions between atoms in organic molecules over a wide range of distances. Lifson et al.[10] showed that a power of 9 or 10 was better than 12 for organic compounds, and such values are sometimes used. The power 12 is usually used for proteins, not because it is accurate, but because it is fast to compute from the attractive r^{-6} term.

The Buckingham potential replaces the twelfth power term with an exponential, which is a better theoretical description of the repulsion expected between electron clouds[11].i.e.

$$V_r = \sum_r [A\exp(-Br) - Cr^{-6}]$$

(EQ 5.11)

In most circumstances this function behaves similarly to the Lennard-Jones equation but at very short interatomic distances the function inverts and goes to -∞, an obvious danger in poorly constructed model structures.

For protein structures the Lennard-Jones potential is usually used, as the exponential term in the Buckingham potential takes 20 times longer to compute than a floating point multiply on most computers (r^{-6} can be calculated from $r^{2)}$. For a small molecule the number of interactions is relatively small and the close range behaviour is crucial so in this case the Buckingham potential may give better results.

5.2.5 Coulombic Interactions

The earliest approach to obtaining the electrostatic energy term assigned bond dipoles to bonds between different types of atoms and calculated the electrostatic energies from dipole-dipole interactions (see Figure 5.3)

$$V_q = \sum_r \frac{\mu_i \mu_j}{D r^3_{ij}} (\cos\chi - 3\cos\alpha_i \cos\alpha_j)$$

(EQ 5.12)

where D is the dielectric constant, r_{ij} is the separation of the dipoles, χ is the angle between the dipoles, μ_i and μ_j are the values of the dipole and α_i and α_j are the angles each dipole makes to a line connecting them.

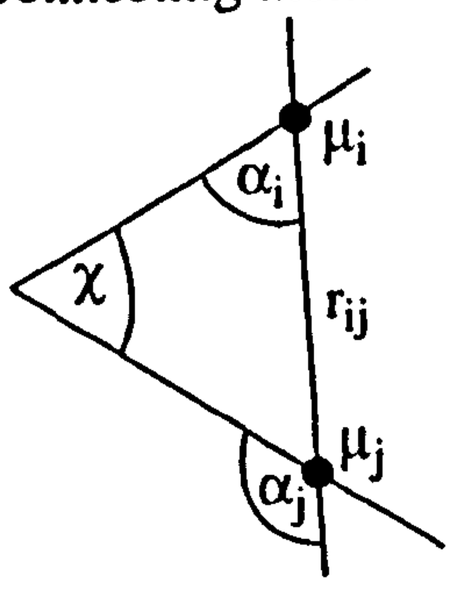


FIGURE 5.3. Single Dipole Interaction

The bond dipoles were chosen to fit known dipole moments of molecules. It was found that the effective dielectric constant of the solvent had to be taken into account to fit known experimental data. When molecular mechanics calculations were extended to large molecules with many polar bonds, it became clear that such calculations were quite time consuming.

Instead of placing point dipoles in bonds, one can place point charges at atoms, chosen so as to match the previous bond moments or as determined from *ab initio* calculations. From the point charges, Coulomb's law is used to calculate the energies[12]

$$V_q = 332 \sum_r q_i q_j / Dr$$

(EQ 5.13)

where q_i and q_j are the charges on the atoms i and j separated by the distance r_{ij}. The scaling factor 332 converts the energy to units of Kcal per mole.

If there are net charges present, as in proteins, the point charge approximation involves no extra calculation, but the dipole-dipole method requires that charge-charge and charge-dipole interactions also be carried out. The results are similar either way, but the point charge calculation can be carried out more quickly, and this method is usually used for proteins.

There are two choices for D, either a fixed value between 1 and 5 is used or a distance dependent dielectric[13,14] is used where D = 4r (sometimes D = r but this gives undue weight to coulombic interactions). Some force fields using a fixed dielectric constant (usually 1) claim to accurately represent the interaction of the collection of point charges being considered. This method, however, has undesirable computational consequences as the r term in the denominator has to be calculated from r^2. The squared term is calculated by Pythagoras and taking the square root is a relatively time consuming process.

By approximating the value of D to r this avoids taking the square root of r^2. There is also a physical justification for this procedure[8]. The value of D for a system consisting of two separated point charges in a vacuum is 1, and as matter is interspersed between the charges, the value of D becomes greater than 1 (i.e. the greater the separation the greater the chance of interspersed matter and the higher the value of D). A distance dependent dielectric is therefore not only computationally efficient but physically justified.

5.2.6 Other terms

The five terms described above are the core elements of almost all molecular mechanics force fields; in some case the entire energy function. In many situations, however, it is necessary for additional terms to be included.

5.2.6.1 Out of plane bending

This term is included to incorporate the energy increase with out of plane bending (pyramidalization) of trigonal planar systems such as carbonyl groups. The four atoms in such a grouping should be kept in the same plane, however, the branch atom (oxygen in the case of carbonyl) can be distorted. Since the deformation is likely to be very small it is possible to use a simple harmonic potential energy function[15]:-

$$2V_{\chi} = \sum_{\chi} k_{\chi} (180 - \chi)^2$$

(EQ 5.14)

where k_{χ} is the force constant for out-of-plane bending and χ is the improper torsion angle in degrees (it is 180^{o} when the conformation is planar).

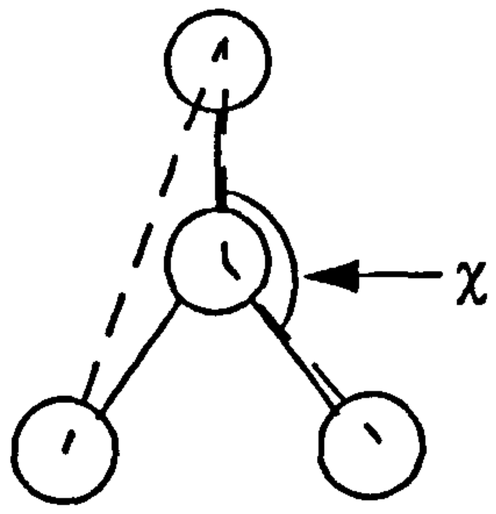


FIGURE 5.4. The Improper Torsion Angle (χ shown by dashed line)

5.2.6.2 Cross terms

These terms are usually needed when the force field is required to reproduce information on vibrational frequencies. They involve two different motions at the same time such as stretch-bend, bend-bend, torsion-bend and torsion-stretch[1].

By examining the structure of butane it is clear that there is a change in the C-C bond length and an opening of the C-C-C bond angles when changing from the *trans* to the

cis conformations (see Figure 5.5 on page 163). This can be incorporated into the force field via a stretch-bend interaction:-

$$V_{l\theta} = \sum\sum k_{l\theta}(l - l_0)\ (\theta - \theta_0)$$

(EQ 5.15)

where $k_{l\theta}$ is the force constant for stretch-bending and l, l_0, θ, θ_0 are before. This has the effect of restraining distortion of the angle through compensatory bond stretches.

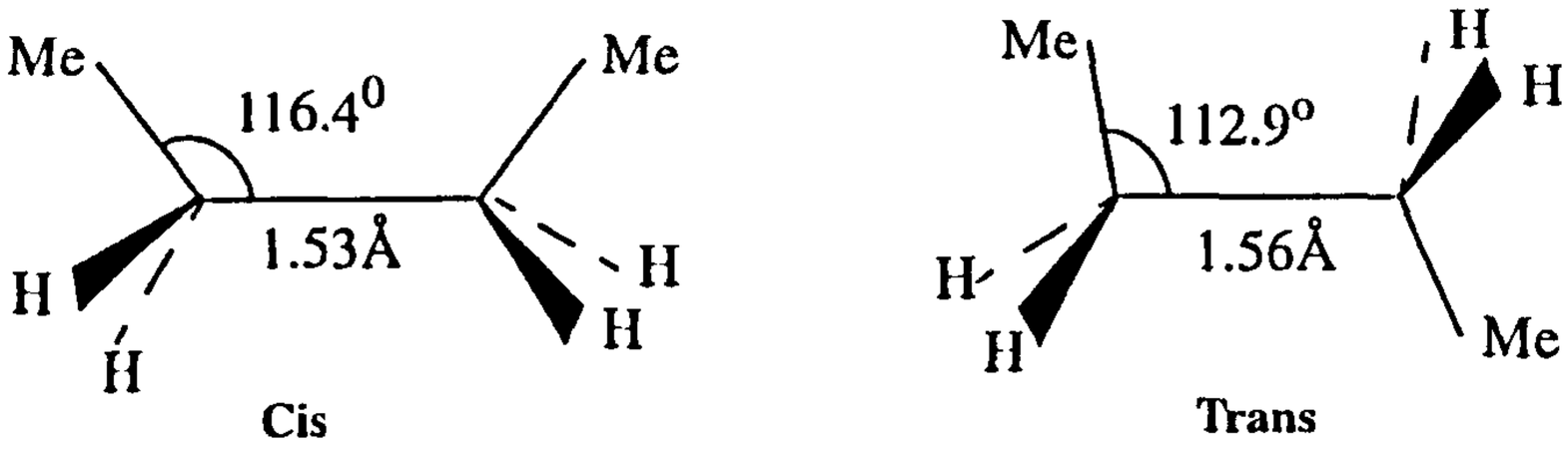


FIGURE 5.5. Molecular geometries for cis and trans butane structures

To better fit vibrational frequencies, a bend-bend interaction term is used for the bending of two angles at a common centre. This is given by an equation of the form:-

$$V_{\theta\theta'} = \sum\sum -0.021914 k_{\theta\theta'}(\theta - \theta_0)\ (\theta' - \theta'_0)$$

where θ and θ' are two valence angles on the same atom and θ_0 and θ'_0 are the appropriate strain free bond angles.

A torsion-bend term couples a torsion angle (A-B-C-D) with the two vicinal bending angles A-B-C and B-C-D.

$$V_{\theta w} = \sum\sum k_{\theta w}(\theta - \theta_0)\ (\theta' - \theta'_0)\cos\omega$$

(EQ 5.16)

This term has considerably improved the agreement between calculated and experimental frequencies of vibrational modes of atoms bonded to two adjacent carbon atoms.

A torsion-stretch term is added in situations where certain bonds eclipse each other resulting in insufficient bond stretching. The equation which helps correct this is given by the expression

$$V_{l\omega} = 11.995\left(\frac{k_{l\omega}}{2}\right)(l - l_0)\ (1 + \cos 3\omega)$$

where $k_{l\omega}$ is the force constant for torsion-stretching and the other constants are as described previously.

When trying to devise a force field for a molecule all of these additional terms are not included initially. At the start these terms are all assumed to be zero and are added as they seem to be required for some reason. If structural information is needed then few of these cross terms are big enough to cause significant changes. On the other hand if vibrational frequencies are to be considered then these terms are required to ensure high accuracy of the frequencies.

5.2.7 Force Field Parameterisation

The reliability of a molecular mechanics calculation is dependent on the potential energy equations and the numerical values of the parameters that are incorporated into those equations. In general, parameters are not transferable from one force field to another due to the different forms of equations that have been used and because of parameter "correlation" within a force field. This occurs if an error is made regarding one parameter, other parameters in the force field adjust to minimise any error that would be caused. Thus force fields that may give good results for one group of compounds may yield poor results for another group.

It is not usually possible to include all the possible parameters in a molecular mechanics program. For example consider a torsional angle of the form *a*-*b*-*c*-*d* where *a*,*b*,*c*,*d* are four different atom types. If a program contains 68 atom types then there are 68^4 possible torsion sets, and there are twice as many torsional force constants giving several million torsional parameters. A similar situation occurs with other parameter types.

It turns out that only a tiny percentage of parameters are known by experiment or calculation so far reported in the literature. For relatively simple functionalised

compounds, such as alcohols or ketones, it is likely the parameter set will be complete but for more complicated molecules containing various combinations of heteroatoms it is possible to find cases where parameters are missing.

Parameterisation can be approached from two directions: least squares optimisation[18] and trial and error[3]. Least-squares optimisation methods obtain a simultaneous best fit of calculated results to experimental data. In either case parameterisation is far from straightforward as the data sets usually available come from a variety of sources (i.e. crystal structures, vibrational spectra, quantum mechanical calculations etc.), are measured by different kinds of experiment in different units, and have relative importance that require subjective assessment.

With the least squares approach correlation between parameters can give problems. Also the derivatives involved in the calculation are extremely complex. The trial and error method is the most frequently used - mainly because it is simple to implement and does not take much longer than least squares.

The quality of a parameter is directly dependent on the quality and nature of the experimental or theoretical data available. It is also dependent on the level of accuracy required. In some cases generalised approximate parameters based on known trends are used. These can, however, lead to serious problems if an exact value is essential for understanding some property that is being studied.

In general the greater the complexity and number of parameters, the more accurate the optimised force field becomes at the expense of the time required to do the calculation. Obviously a balance has to be reached between accuracy of calculation and time taken. If the structure and energy of a molecule is being studied then significant errors in the force field parameters are often acceptable. However, if vibrational frequencies are required then a more accurate force field is required.

5.3 Energy Minimisation

Using a CAMD (Computer Aided Molecular Design) package it is possible to construct a new molecule by combining smaller molecules or fragments of molecules. The molecule can also be constructed one atom at a time using known average bond

lengths, valency angles, and torsion angles. Obviously structures built up this way are extremely crude. Before the modelled structure can be useful it must be computationally optimised by a procedure known as energy minimisation.

This involves systematically altering the geometry of the molecule (i.e. the atomic coordinates are shifted by a calculated amount) in the hope of locating the global energy minimum. Various optimisation methods can be used in the attempt to achieve this. Unfortunately though, all of these methods are prone to locating the local energy minimum closest to the starting point of the calculation, rather than the global minimum.

The difference between local and global energy minimum can be illustrated by considering the rotational potential for a 1,2 di-substituted ethane (See Figure 5.6). The *gauche* conformation is stable but the molecule's preference would be for the *anti* conformation. This trivial example illustrates one of the most difficult problems in computational chemistry: how to be sure the global minimum has been reached.

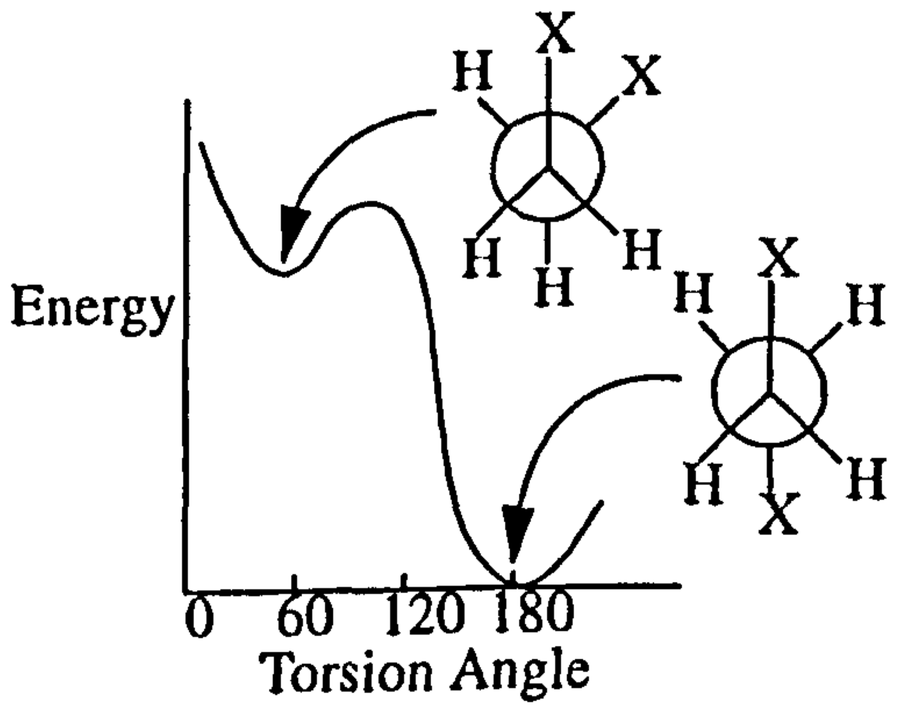


FIGURE 5.6. Shape of rotational potential for 1,2-di-substituted ethanes

There are two main categories of optimisation technique namely search and gradient methods. An example of a search method is pattern searching[16].

5.3.1 Pattern Searching

Pattern searching applies positive and negative shifts (~0.1Å) to each atom's atomic coordinates one at a time and then tests to see whether the steric energy has decreased or increased. If the energy has decreased then the atom is left in its new position and the new steric energy used as the current value. However, if the energy has increased then the atom is returned to its original position and the coordinate is then shifted in the

opposite direction. Again the steric energy is calculated and if it has decreased then the atom is left in the new position otherwise it is returned to its original position.

The whole pattern of successful shifts built up in this way is repeated and the steric energy checked for further reduction. The pattern is repeated until it no longer works and then the pattern is repeated with half the shift value and then iteratively until the shift reaches a sufficiently small value (10^{-5}Å). When the current pattern no longer works, or the shift becomes too small, a new pattern is established and the whole process repeated until a reduction in energy is no longer possible.

This method is guaranteed to find a local energy minimum and has a large radius of convergence (i.e. even with an extremely crude starting structure a local energy minimum will be reached). However the rate of convergence is slow as the same shift size is applied to each coordinate and the shift size is refined very slowly (i.e it could take hundreds of iterations to reach an energy minimum).

5.3.2 Gradient based methods

Gradient based methods again apply a shift, in the search for lower energy, to each coordinate but in this case the shift is proportional to the gradient of the steric energy at this point (i.e if the gradient of the steric energy is steep then a large shift is applied, if the steric energy function is flatter then a smaller shift is applied). These techniques are said to reach an energy minimum when the vector of first partial derivatives of the steric energy with respect to the atomic coordinates is zero. This is the case not only at energy minima but also at energy maxima and saddle points; a feature of gradient methods which can be useful when searching for transition state structures but an inconvenience when looking for minima.

Gradient based techniques have a fast rate of convergence as they calculate shifts based on the gradient of the steric energy function. However, the radius of convergence is small for the popular full matrix Newton Raphson (NR) iteration (see later for explanation). The radius of convergence can be increased by using approximations to the full NR such as the Block Diagonal Newton Raphson iteration and steepest descents, but at the expense of rate of convergence.

5.3.2.1 Steepest Descent

An example of a simple gradient based method is steepest descent[17] (a variation of the full NR iteration - almost all gradient based methods of optimisation are variants of the NR iteration). This involves calculating the gradient (the first partial derivative of the steric energy w.r.t. the atomic coordinates) of the steric energy function at a particular point. Once the gradient has been calculated the coordinates are shifted in the direction of lower energy by an amount proportional to the gradient. The constant of proportionality is determined empirically. This procedure is repeated until a local minima is reached.

Steepest descent has the disadvantage that it is only the gradient of the steric energy function that is considered and the curvature (the second partial derivative of the steric energy) of the function is not taken into account when calculating the shift. A result of this is that the rate of convergence slows down considerably near the minimum energy position. Steepest descents does however have the advantage that it converges well when the geometry is far removed from its minimum and can be used to model geometries prior to refinement by another method.

5.3.2.2 Newton Raphson

A technique which considers both the gradient and curvature of the steric energy function is the Newton Raphson iteration[19]. The proof for this procedure can be derived from simple calculus.

The minimum on a curve, at point x*, is where the first partial derivative is equal to zero. i.e.

$$f'(x^*) = 0$$

(EQ 5.17)

Since in a molecular mechanics calculation the starting point is x and not the minimum x^*,

$$x^* = x + \delta x$$

(EQ 5.18)

where δx represents the changes x must undergo to reach the minimum value. Equation 5.17) can therefore be written in terms of x

$$f'(x+\delta x) = 0$$

(EQ 5.19)

and then expanded as a Taylor series

$$f'(x+\delta x) = f'(x) + f''(x)\,\delta x + f'''(x)\,\delta x^2 + \ldots$$

(EQ 5.20)

which is also set to zero. Truncating the Taylor series after the second order term gives

$$f'(x) + f''(x)\,\delta x = 0$$

(EQ 5.21)

By rearranging Equation 5.21) an expression is given for δx (the change in x which must be made to x to reach the minimum).

$$\delta x = \frac{-f'(x)}{f''(x)}$$

(EQ 5.22)

which can be substituted into Equation 5.18) to give

$$x^* = x - \frac{f'(x)}{f''(x)}$$

(EQ 5.23)

This equation implies that the energy minimum is reached in one step. This is not the case, however, as the Taylor series was truncated. This forces the calculation to be carried out in a stepwise, iterative fashion.

This proof represents the simple one dimensional case. Molecules, in general, have 3N-6 degrees of freedom where N is the number of atoms and therefore the term $f'(x)$ is replaced by a vector containing the first partial derivatives of the steric energy with respect to the atomic coordinates. The $f''(x)$ term is replaced by a matrix containing the second partial derivatives with respect to the atomic coordinates.

The basic NR iteration which minimises the steric energy of the molecule is therefore given by:-

$$x_{k+1} = x_k - \alpha F^{+} \nabla V_s(x)$$

(EQ 5.24)

where **x** is the 3N (N = number of atoms) long vector of cartesian coordinates, α is the step length, F^{+} is the generalised inverse of the Hessian:-

$$F = \frac{\partial^2 V_s}{\partial x_i \partial x_j} \quad ; \qquad i = 1,3N,\ j = 1,3N$$

and:-

$$\nabla V_s(x) = \frac{\partial V_s}{\partial x_j}; \qquad j = 1,3N$$

The calculation of the complete Hessian (a 3N x 3N matrix) is a very time consuming procedure and is not really suitable for molecules with over 200 atoms. Therefore an approximation known as the Block Diagonal Newton Raphson (BDNR) is used. This is so called because only the second partial derivatives in each 3x3 block along the leading diagonal of the Hessian are calculated. Therefore F is given by:-

$$F = \left(\frac{\partial^2 V_s}{\partial x_i \partial x_j} \right); \qquad i = 3m+1, 3m+3;\ j = 3m+1, 3m+3;\ m = 0, N-1$$

Each block contains second partial derivatives of the steric energy with respect to the coordinates of only one atom. The BDNR iteration can therefore be applied one atom at a time, allowing each atom to be moved to its corrected position before the calculations for the next atom are started. Each atom's position is therefore calculated on the basis of the best structure available at the time.

The BDNR iteration converges faster (usually in 50-200 iterations) than the steepest descent or pattern based methods and has a reasonable radius of convergence.

5.3.2.3 Calculation of Derivatives

The derivatives can be calculated in two ways: numerically or analytically[20]. Numerical methods use finite difference calculations to calculate the derivatives (i.e.

the coordinates are shifted by a small amount and the energy re-calculated). The equations for the first and second derivatives by numerical methods are therefore:-

$$\frac{\partial V_s}{\partial x_i} = \frac{V_s(x_i + \delta x) - V_s(x_i - \delta x)}{2\delta x}$$

Central Difference (EQ 5.25)

$$\frac{\partial^2 V_s}{\partial x_i x_j} = \frac{V_s(x_i + \delta x, x_j + \delta x) - V_s(x_i, x_j + \delta x) - V_s(x_i + \delta x, x_j) + V_s(x_i, x_j)}{\delta x^2}$$

Forward Difference followed by Reverse Difference (EQ 5.26)

$$\frac{\partial^2 V_s}{\partial x_i^2} = \frac{V_s(x_i + \delta x) + V_s(x_i - \delta x) - 2V_s(x_i)}{\delta x^2}$$

(EQ 5.27)

where δx is a small value (i.e. 0.001) and i = 1,3N, j = 1,3N. The steric energy is therefore calculated at (x_i), $(x_i + \delta x)$, $(x_i - \delta x)$, $(x_j + \delta x)$ and $(x_i + \delta x, x_j + \delta x)$. The second partial derivatives vary by so little after each iteration that it is sufficient to calculate them after only every 4 or 5 iterations for the Newton Raphson method.

Analytical derivatives are determined by applying calculus to the various steric energy terms. The following sum of derivatives is required.

$$\sum_{i=1}^{3N} \frac{\partial V_s}{\partial x_i}$$

(EQ 5.28)

Using the chain rule the first partial derivatives of energy (V) with respect to the cartesian coordinates can be expressed as

$$\frac{\partial V}{\partial x_i} = \frac{\partial V}{\partial q} \cdot \frac{\partial q}{\partial x_i}$$

(EQ 5.29)

and the second partial derivatives as

$$\frac{\partial^2 V}{\partial x_i x_j} = \frac{\partial^2 V}{\partial q^2} \cdot \frac{\partial q}{\partial x_i} \cdot \frac{\partial q}{\partial x_j} + \frac{\partial V}{\partial q} \cdot \frac{\partial^2 q}{\partial x_i x_j}$$

(EQ 5.30)

where q are the internal coordinates (i.e. lengths, angles and torsion angles)

For example for bond stretching where the original equation is given by:-

$$V_l = \frac{1}{2} k_l (l - l_0)^2$$

$$= \frac{1}{2} k_l (l^2 - 2ll_o + l_o^2)$$

(EQ 5.31)

then the derivative of V_l with respect to the internal coordinate l is

$$\frac{\partial V_l}{\partial l} = \frac{1}{2} k_l (2l - 2l_0) = k_l (l - l_0)$$

(EQ 5.32)

and the second derivative w.r.t l is

$$\frac{\partial^2 V_l}{\partial l^2} = k_l$$

(EQ 5.33)

The first partial derivative of l w.r.t the cartesian co-ordinates is fairly simple to calculate, however, the derivatives of θ and ω are not quite as easy. They are derived by taking the first partial derivative of the cosine of the angle w.r.t. the atomic coordinates, as this is more straightforward than differentiating the angle itself.

The equation for the first partial derivatives of θ w.r.t the cartesian coordinates is

$$\frac{\partial \theta}{\partial x_i} = \frac{-1}{\sin\theta} \cdot \frac{\partial}{\partial x_i} (\cos\theta)$$

(EQ 5.34)

and the equation for the second partial derivatives is

$$\frac{\partial^2\theta}{\partial x_i \partial x_j} = \frac{-\cos\theta}{\sin^3\theta} \cdot \frac{\partial}{\partial x_i}(\cos\theta) \cdot \frac{\partial}{\partial x_j}(\cos\theta) - \left(\frac{1}{\sin\theta} \cdot \frac{\partial^2}{\partial x_i \partial x_j}(\cos\theta) \right)$$

Minimisation is moderately faster using analytical derivatives as it does not require multiple energy calculations. In a situation, however, where the form of the force field is constantly being changed (i.e. optimisation of the force field) numerical derivatives are more useful as there is no need to know the form of the force field.

5.4 Conclusions

The basics of molecular mechanics have been described: the components of the steric energy equation and the energy minimisation techniques. Depending on the information required (i.e structural, thermodynamic) from the force field calculation, different forms of force field are used.

When far from an energy minimum, the simple steepest descents based minimisation, and a less complicated force field are the methods preferred. Close to the energy minimum, more sophisticated procedures such as the Newton Raphson iteration are more commonly used.

Even with the increase in computational power it is still the case that *ab initio* calculations are only feasible on molecules with up to 100 atoms. Molecular mechanics calculations, however, can be conducted on molecules with thousands of atoms. This makes them suitable for studying large biological molecules and hence they are often used in drug design.

References

[1] Grant, Guy H. and Richards, W. Graham. *Computational Chemistry.* Oxford University Press, London, 1995. ISBN 0 19 855740 X

[2] Bowen, J. Phillip., Allinger, Norman L.Molecular Mechanics: The Art and Science of Parameterisation. *Reviews in Computational Chemistry.* 1991, **Vol.2**, pp. 81-95. ISBN 1 56081 515 9

[3] Allinger, N.L. Calculation of Molecular Structure and Energy by Force-Field Methods. *Adv.Phys.Org.Chem.* 1976, 13, pp. 1-76. ISBN 0 12 033513 1

[4] White, D.N.J. Molecular Mechanics Calculations. *Mol. Struct. Diffr. Methods.* 1978, **No.6**, pp. 38-62

[5] Wilson et al. *Molecular Vibrations.* McGraw-Hill, London, 1955

[6] Dinur, Uri., Hagler, Arnold, T. New Approaches to Empirical Force Fields. *Reviews in Computational Chemistry.* 1991, **Vol.2**, pp. 99-164. ISBN 1 56081 515 9

[7] Allinger, N.L., Yuh, Y.H., Lii, J.H. Molecular Mechanics. The MM3 Force Field for Hydrocarbons. *J.Am. Chem. Soc.* 1989, **111**(23), pp. 8551-8582

[8] White, David N.J., Ruddock, Noel J. and Edgington, Paul R. Molecular Design with Transparallel Supercomputers. *Molecular Simulation.* 1989, **Vol. 3**, pp. 71-100

[9] Lennard-Jones, J.E. Cohesion. *Proc. Phys. Soc. (London)*, Ser. A. 1931, 43, 461

[10] Warshel, A and Lifson, S. Consistent Force Field Calculations. II. Crystal Structures Sublimation Energies, Molecular and Lattice Vibrations, Molecular Conformations and Enthalpies of Alkanes. *J.Chem.Phys.* 1970, 53, pp. 582-594

[11] Hill, T.L. *J.Chem.Phys.* 1948, **Vol 16**, 399

[12] Meyer, A.Y., Forrest, F.R.F. Towards the Convergence of Molecular Mechanics Force Fields. *J. Comput. Chem.* 1985, **6**, pp. 1-4

[13] Weiner et al. A New Force Field for Molecular Mechanical Simulation of Nucleic Acids and Proteins. *J.Am.Chem.Soc.* 1984, **106**, pp. 765-784

[14] Lennart, Nilsson., Karplus, Martin., Empirical Energy Functions for Energy Minimisation and Dynamic of Nucleic Acids. *J. Comput. Chem.* 1986, **7**, pp. 591-616

[15] Warshel, A., Levitt, M., Lifson, S. Consistent Force Field Calculations of Vibrational Spectra and Conformations of some amides and Lactam Rings. *J.Mol.Spectroscopy*, 1970, Vol.33, pp. 84-99

[16] Engler, E.M., Andose, J.D., Schleyer, P. von R. Critical Evaluation of Molecular Mechanics. *J.Amer.Chem.Soc.* 1973, **95**, pp. 8005-8025

[17] Wiberg, K.B. A Scheme for Strain Energy Minimisation. Application to Cycloalkanes. *J.Amer. Chem. Soc.* 1965, **87**, pp. 1070-1078

[18] Lifson, S., Warshel. A. Consistent Force Field Calculations of Conformations, Vibrational Spectra, and Enthalpies of Cycloalkane and n-Alkane Molecules. *J.Chem.Phys.* 1968, **49**, pp. 5116-5129

[19] White, D.N.J., and Ermer, O. Molecular Mechanics - A Cautionary Note. *Chem. Phys. Letters.* 1975, 31, pp. 111-112

[20] Niketic, Svetozar R., Rasmussen, Kjeld. *The Consistent Force Field: A Documentation.* Springer-Verlag, Berlin, 1977.

Chapter 6

Parallel Molecular Mechanics Calculations using COMFORT and the BB08

This chapter describes the parallelisation of a sequential FORTRAN molecular mechanics program to run on novel hardware, where each node processor has a dedicated high speed link to the host processor, and to all of the other nodes. Code/data can be broadcast from the host to the nodes over these direct links using an overhead free hardware mechanism. The broadcast hardware (the BB08) is supported by the COMFORT message passing subroutine library. The calculation is executed on a PC host computer with four T414 nodes on a BB08.

First of all the main features of CAMD (Computer Aided Molecular Design) are described before explaining the BB08 (the broadcast hardware) and COMFORT. The structure of the sequential molecular mechanics program CHEMMIN is detailed and then the parallelisation strategies for molecular mechanics calculations are discussed. The parallelisation of CHEMMIN is described along with the implementation of COMFORT into it. Finally the addition of a graphical interface to the parallel molecular mechanics program is detailed.

6.1 Introduction

Molecular mechanics(MM) calculations as described in the previous chapter are a very computationally intensive task. Even for a small molecule (~100 atoms) there are thousands of parameters to be evaluated. The advent of powerful workstations and parallel computers have made it possible to execute MM calculations on large protein structures comprising thousands of atoms.

Usually the MM calculation is incorporated into a CAMD package[1-4]. This is a program with a graphical interface which allows the user to construct molecules by combining smaller molecules or fragments of molecules into a larger overall structure. The molecule can also be constructed one atom at a time using known average bond lengths, valency angles, and torsion angles.

Once the molecular model has been built it can be manipulated in various ways: bonds can be broken and joined; lengths, angles and torsion angles can be altered; the image of the molecule can be altered from ball and stick representation to space-filling, 3-D stereo etc.; the whole molecule can be rotated, scaled etc. There is usually a range of computational procedures available as well, such as molecular mechanics, molecular dynamics and conformational search procedures like Monte Carlo.

Even on the fastest workstations that are available today it can still take a long time (several hours or even days) to execute the computational procedures mentioned above. Many implementations of parallel molecular dynamics[5-9] (the simulation of molecular motions with time) have been attempted, however little work has been published on parallel energy minimisation.

This chapter describes a parallel implementation of an energy minimiser which utilises the COMFORT host/node programming environment and BB08 octal broadcast link interface[13]. Each node has a direct link to the host computer, down which code and/or data can be transmitted, received or broadcast.

6.2 The BB08 and COMFORT

6.2.1 The BB08 Broadcast Link Interface

The BB08 octal broadcast link interface is a printed circuit board with eight C012 link adapters interfaced to a microprocessor style bus. Data can be broadcast simultaneously to all the C012s. The links are either routable to size one TRAM slots or to a DIN41612 connector (See Figure 6.1).

By routing the links to the connector this allows microprocessors such as PCs to be used as nodes. Using one BB08 board per node this can provide full connectivity between eight nodes each with a direct connection to the host. Figure 6.2 shows the connections from Node 3. The other nodes have connections that are analogous to this. It is possible to connect 16, 32 etc. nodes if more BB08 boards are added per node.

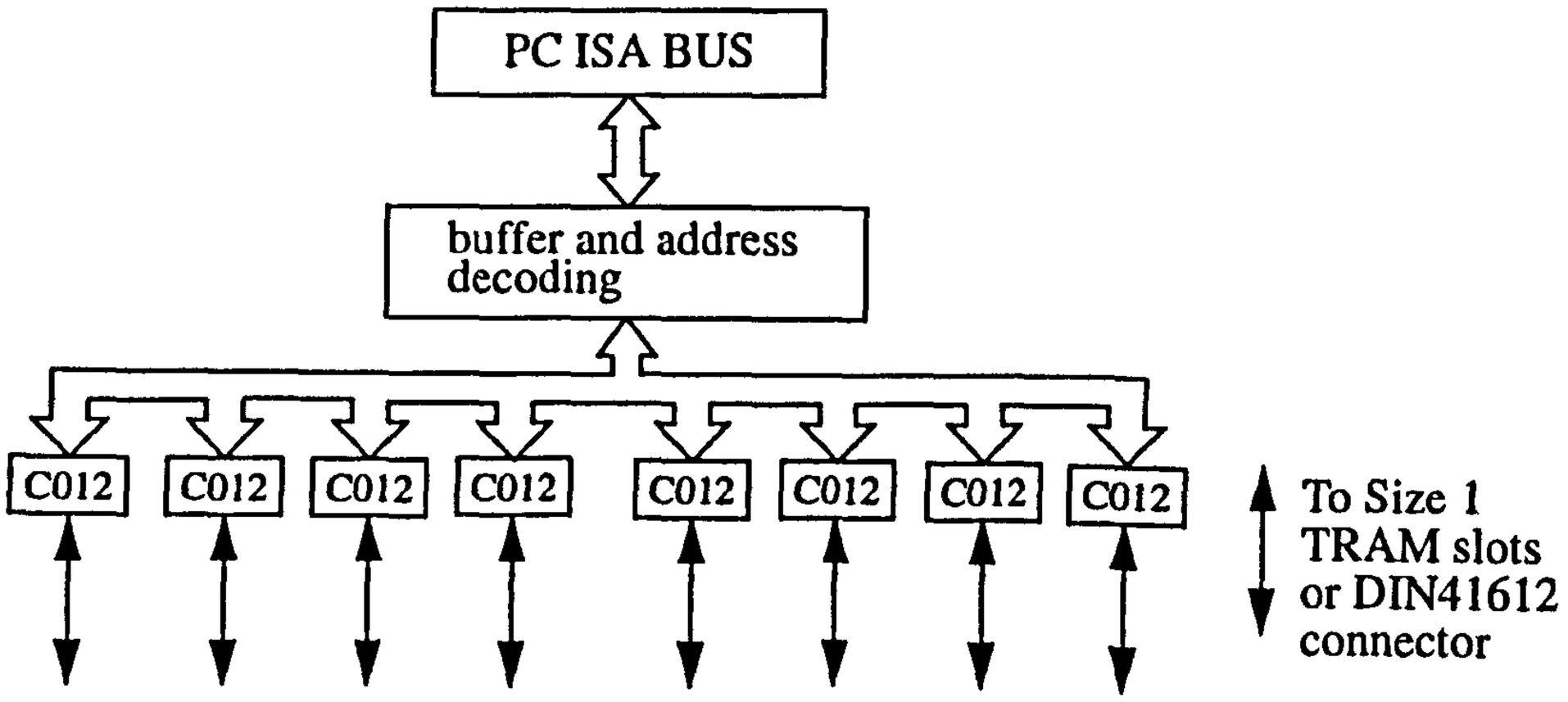


FIGURE 6.1. Basic layout of BB08 board

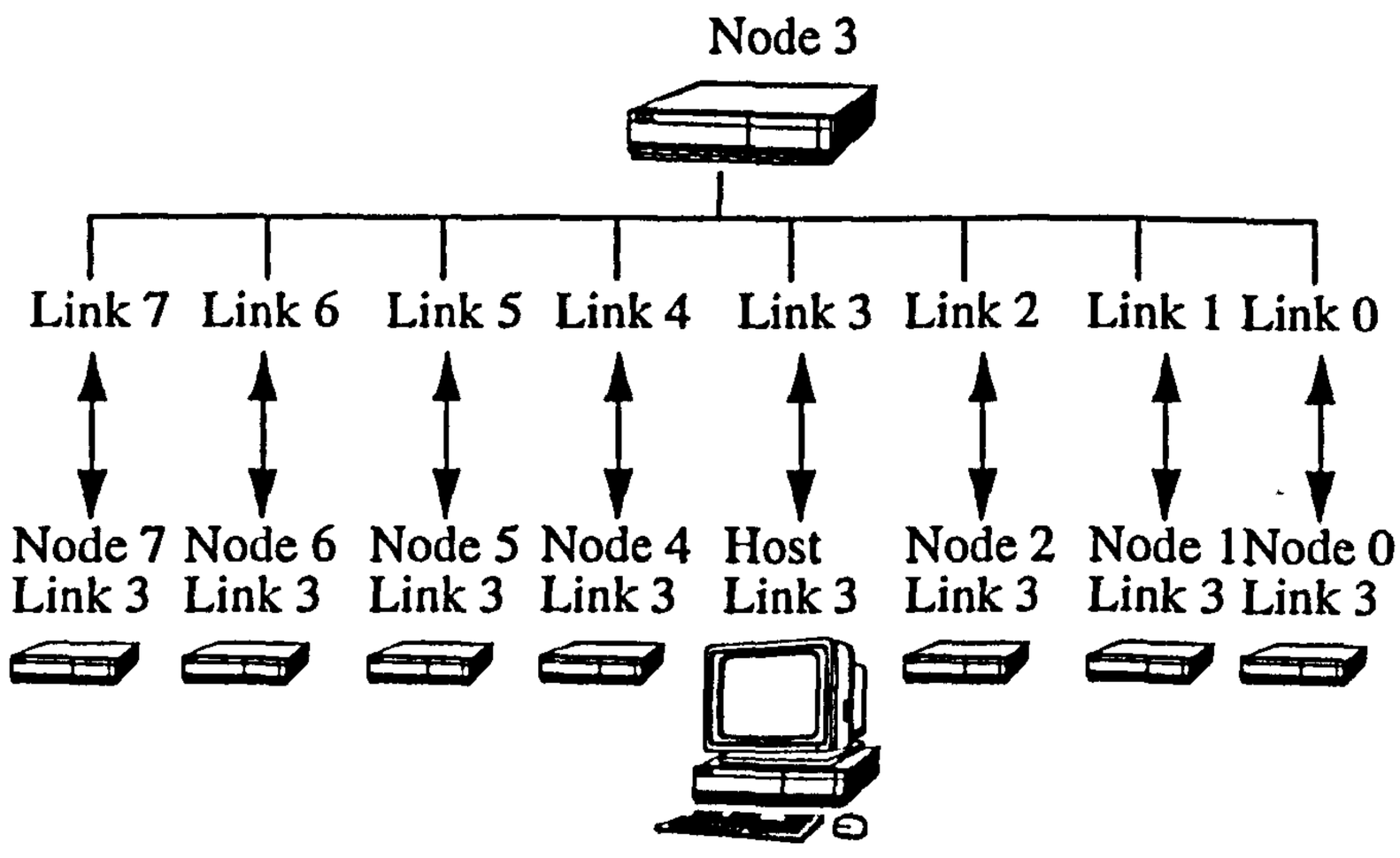


FIGURE 6.2. Connections from BB08 board on Node 3

Using PC motherboards as nodes provides an easily up-gradeable, cost-effective, fast parallel computer. It is only the host computer that requires a keyboard and monitor as it loads/runs programs on the node processors. Another advantage of using PC motherboards as node processors is that it is simple, and inexpensive, to add a hard disc and CD ROM drive to every node.

The TRAM slots on the BB08 allow transputers or other TRAM based processors to be used as nodes. Again each node has a direct physical link to the host down which code/ data can be broadcast. In the case of a T-800 transputer that has only four links complete connectivity between the nodes cannot be achieved using the TRAM slots.

The interface between the C012s and the PC bus is similar to the dual link adapter board discussed in Chapter 3. The eight C012 link adapters are accessible in the normal way via their read, write, input status, and output status registers. In addition, however, the write registers can all be accessed simultaneously via a single broadcast data register. During a broadcast write the D0-D7 data lines, the chip select signals, and the read/write signal of all eight C012 link adaptors are activated simultaneously with the same data. This simultaneously outputs the same byte of data down all eight links.

When the parallel minimiser was being developed, a parallel computer that uses PCs as nodes was still under construction. The prototype version of the minimiser was therefore run on four size 1 TRAMs each with a T4XX transputer and 1Mbyte of memory. The same principles could be applied when using PCs as nodes with little alteration to the software.

6.2.2 The COMFORT Programming Environment

COMFORT is a library of FORTRAN subroutines similar to those provided by EXPRESS (Chap. 1 Ref. 11) and MPI (Chap. 1 Ref. 16), which allow the host computer to broadcast load code onto the nodes and also facilitate communication between the nodes, amongst other things. The host/node methodology allows the host to participate in the calculation rather than act merely as a facilities server. COMFORT makes parallelisation easier because no communication tasks are required on the nodes and no configuration (in the 3L FORTRAN sense) is required.

The loading and running of programs on the nodes is the responsibility of the host processor. A program is loaded onto the nodes and run, either from the node's hard disc if it is already there, or via the host to node link with a hard disc copy being made for future use.

For speed and efficiency COMFORT is built around a library of low level subroutines which are written in assembly language. An example of some of these subroutines is shown below in Table 6.1.

The name of the subroutines reflect their function. The user does not usually need to access these low level subroutines although they are available and documented. These

routines are not required in the application discussed in this chapter as it does not require inter-node communication, only host-node.

Fortran Call	Definition of Arguments
Call LinkInByte (LinkBase, Data-Byte	LinkBase - Base address of C012 link adaptor registers DataByte - Data byte read from C012 read data register
Call LinkOutByte (LinkBase, DataByte)	LinkBase - Base address of C012 link adaptor registers DataByte - Data byte written to C012 write data register
Call LinkInWord (LinkBase, DataWord, ByteRev)	LinkBase - Base address of C012 link adaptor registers DataWord - Int*2 value assembled from input bytes ByteRev - Enables big/little endian byte ordering
Call LinkOutWord (LinkBase, DataWord, ByteRev)	LinkBase - Base address of C012 link adaptor registers DataWord - Int*2 value to output as bytes ByteRev - Enables big/little endian byte ordering
Call LinkInLongWord (Link-Base, DataLongWord, ByteRev)	LinkBase - Base address of C012 link adaptor registers DataWord - Int*4 value assembled from input bytes ByteRev - Enables big/little endian byte ordering
Call LinkOutLongWord (Link-Base, DataLongWord, ByteRev)	LinkBase - Base address of C012 link adaptor registers DataLongWord - Int*2 value to output as bytes ByteRev - Enables big/little endian byte ordering
Call LinkInMessage (LinkBase, Massage, MessageLength)	LinkBase - Base address of C012 link adaptor registers Message - Message assembled from input bytes MessageLength - Number of bytes to input
Call LinkOutMessage (LinkBase, Message, MessageLength)	LinkBase - Base address of C012 link adaptor registers Message - Message of bytes to output MessageLength - Number of bytes to output

TABLE 6.1. COMFORT low-level subroutines

These low level subroutines are designed to explicitly support message passing. In addition to these, COMFORT has a library of subroutines designed to give the FORTRAN programmer access, amongst other things, to the rich set of run time facilities usually available when programming in C. These include access to a range of directory services, file services and the PC extended memory. Examples of some of these routines are shown in Table 6.2..

Fortran Call	Definition of Arguments
Call ChangeDirectory (DirectoryName, Error)	DirectoryName - Name of directory to change to Error - Error number
Call DeleteDirectory (DirectoryName, Error)	DirectoryName - Name of directory to delete Error - Error number

TABLE 6.2. COMFORT run-time libraries

Fortran Call	Definition of Arguments
Call MakeDirectory (DirectoryName, Error)	DirectoryName - Name of directory to create Error - Error number
Call DeleteFile (FileName, Error)	FileName - Name of file to delete Error - Error number
Call FindFileFirst (FileMask, Attributes, FileName, Error)	FileMask - File mask for search Attributes - File Attributes FileName - First filename which matches file mask Error - Error number
Call GetFileAttribs (FileName, FileAttribs, Error)	FileName - Filename whose attributes are required File Attribs - File attributes Error - Error number
Call AllocateXm (BlockSize, Handle, Error_	Blocksize - Size of extended memory block requested Handle - Handle of extended memory block Error - Error number
Call FreeXmBlk (Handle, Error)	Handle - Xm handle Error - Error number

TABLE 6.2. COMFORT run-time libraries

The application detailed in this chapter uses the higher level subroutines shown below:-

Configure (BoardBase, NumberProcs, TimeoutRes)

Reset (Node)

Load (Node, ExeFileName, Error)

Initialize (Error)

Send (Destination, Buffer, BuffType, BuffLen, Error)

Receive (Source, Buffer, BuffType, BuffLen, Error)

Table 6.3. shows the purpose of each subroutine and defines the arguments.With the exception of Load, subroutines with the same name as their host counterparts are used by the node, although in one or two cases they operate slightly differently. For instance in the case of Load and Receive, on the host these routines possess an extra Timeout argument which specifies the time before a timeout occurs. If there is a problem with host/node communication, Receive will time out after the specified number of clock ticks and return the timeout error number in Error.

It should also be noted that in the Load, Send and Receive routines if the Node ID is set equal to -1 then this broadcasts loads code/data simultaneously onto all the nodes.

The use of these subroutines will become clear in the explanation of their implementation in the parallel energy minimiser.

Subroutine	Definition of Arguments	Purpose
Configure	BoardBase - Base PC i/o address of BB08 NumberProcs - Number of node processors TimeoutRes - Resolution of timeout clock	Hardware setup
Reset	Node - Node ID number (= -1 = all nodes)	Resets one or more nodes prior to loading
Load	Node - Node ID number (= -1 = all nodes) ExeFileName - Name of file containing program code Error - Error number	Loads program onto specified node/nodes
Initialize	Error - Error number	Send each node a message containing the total number of nodes, the host identification number, and the identification number by which the node itself will be known
Send	Destination - ID number of node to receive message Buffer - Integer*1 array of data to transmit BuffType - User defined code for type of message BuffLen - Length of message, in bytes Error - Error number	Sends data in Buffer to Destination
Receive	Source - ID number of node originating the message Buffer - Integer*1 array which holds received message BuffType - User defined code for type of message BuffLen - Length of message, in bytes Error - Error number	Receives data from Source and stores it in Buffer

TABLE 6.3. Description of COMFORT routines

6.3 The Molecular Mechanics Program

The parallel minimiser was derived from a sequential stand-alone minimiser, Chemmin, which was developed in-house at Glasgow University[12]. A stand-alone minimiser usually loads a file containing information on a molecule such as number of atoms, atom types and positions etc., and then performs a single calculation such as energy minimisation. The stand-alone minimiser Chemmin has been integrated into the COMMET and CHEMMOD molecular mechanics packages[3].

6.3.1 The Chemmin Minimiser

The basic structure of Chemmin is shown in Figure 6.3 and its pseudocode is shown in Figure 6.4 on page 185 with the appropriate subroutines highlighted in bold. Chemmin has a modular structure and was written in FORTRAN.

The program can essentially be divided into two parts; initialisation and calculation. **Mindat, Mininit1, Getcop, Getopb** and **Asboml** are responsible for the initialisation.

The pseudocode for **Mindat** is shown in Figure 6.5 on page 186. First of all an integer value is given to each atom type (i.e. H, Csp3, Csp2 etc.). Reference bond angles and electronegativities values are then assigned to each atom type by storing the values in arrays (i.e. EN(1) stores the electronegativity value for atom type 1, EN(2) stores the electronegativity value for atom type 2 etc.). The number of the atom types which are aromatic or involved in double bonds are stored in arrays ARTYPS and DBTYPES respectively.

Several arrays are then constructed which contain:-

- reference bond lengths between atom types
- reference bond lengths for conjugated single bonds between atom types
- reference periodicities for each bond between atom types
- reference barrier to free rotation for each bond between atom types
- A6 and B12 values for Lennard-Jones potential for interactions between atoms of the same type
- reference barrier to free rotation values for conjugated single bonds.

These arrays are two dimensional and arranged such that entry (**i**,**j**) in the arrays contains the correct value for the bond length etc. between **i** and **j** (where **i** and **j** are atom type numbers).

The pseudocode for **Mininit1**is shown in Figure 6.6 on page 186. Again several arrays are constructed which contain:-

- bond stretching constants for each bond between atom types
- bond stretching constants for conjugated single bonds
- A6 and B12 values for the Lennard-Jones potential for interactions between atoms of different types

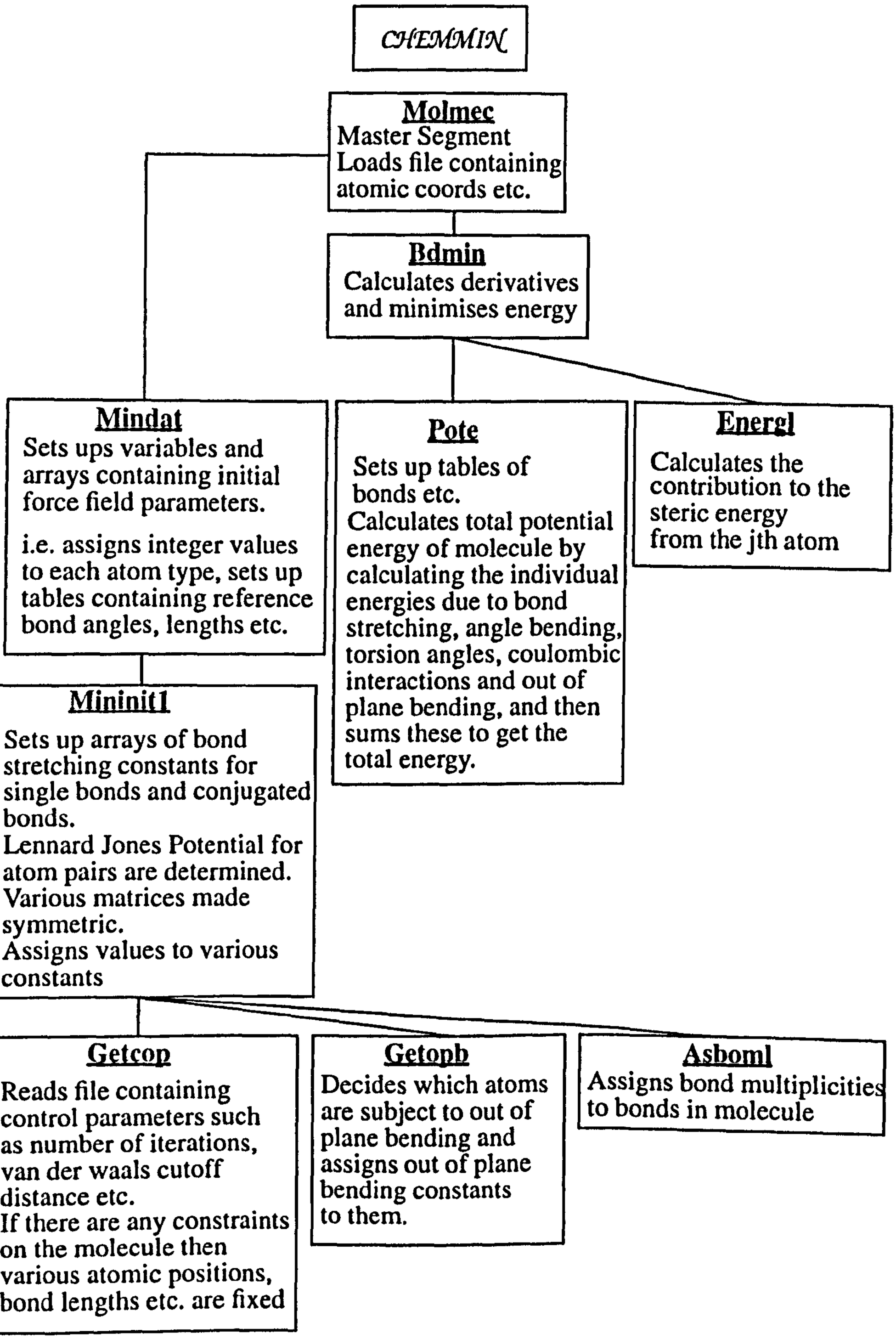


FIGURE 6.3. Program Structure of Chemmin

Initialise various reference values (**Mindat**)
Initialise various constants (**Mininit1**)
Read in file containing data on molecule (**Molmec**)
Assign bond multiplicities to each bond in molecule (**Asboml**)
Read in file containing control parameters (**Getcop**)
Assign out of plane bending constants for planar groups of atoms in molecule (**Getopb**)
Calculate what bonds,angles, torsion angles and nonbonded interactions each atom
in the molecule is involved in and store in tables (**Pote**)
Calculate the total potential energy of the molecule (**Pote**) using

$$V_s = V_l + V_\theta + V_\omega + V_r + V_q + V_{opb}$$

Do j = 1,number of atoms(**Bdmin**)
Calculate the energy of the jth atom using **Energl**
Do k = 1,3
Increment kth coordinate of jth atom and recalculate energy using **Energl**
Decrement kth coordinate of jth atom and recalculate energy using **Energl**
Calculate first derivative using

$$\frac{\partial V_s}{\partial x_k} = \frac{V_s(x_k + \delta x) - V_s(x_k - \delta x)}{2\delta x}$$

Do k = 1,3
Calculate sum of squares of first derivatives
If on 1st, 5th, 9th etc. iteration
Do 1 L =1,3
Increment lth coordinate of jth atom
Calculate $V_s(x_l + \delta x) - V_s(x_l)$
Calculate $V_s(x_l + \delta x) - 2V_s(x_l)$
Do 2 M = L,3
If (L.EQ.M) then goto 3
Increment mth coordinate of jth atom and recalculate energy
using **Energl**
Calculate 2nd derivative using

$$\frac{\partial^2 V_s}{\partial x_l x_m} = \frac{V_s(x_l + \delta x, x_m + \delta x) - V_s(x_l, x_m + \delta x) - V_s(x_l + \delta x, x_m) + V_s(x_l, x_m)}{\delta x^2}$$

Goto 2
3 Calculate 2nd derivative using

$$\frac{\partial^2 V_s}{\partial x_l^2} = \frac{V_s(x_l + \delta x) + V_s(x_l - \delta x) - 2V_s(x_l)}{\delta x^2}$$

2 Continue
1 Continue

FIGURE 6.4. Pseudocode for Chemmin

Assign integer values to each atom type

Assign reference bond angles to each atom type

Assign reference electronegativities to each atom type

Define which atom types are aromatic

Define which atom types are double bonded

Assign reference bond lengths to each bond between atom types

Assign reference bond lengths for conjugated bonds

Assign reference periodicities for each bond between atom types

Assign reference barrier to free rotation values for each bond between atom types

Assign A6 and B12 values for Lennard-Jones potential for each atom type

Assign reference barrier to free rotation values for conjugated single bonds

FIGURE 6.5. Pseudocode for Mindat

Calculate bond stretching constants for each bond between atom types

Calculate bond stretching constants for conjugated single bonds

Calculate the A6 and B12 values for the Lennard-Jones potential for each atom pair

Make matrices containing reference bond lengths, periodicities and barrier to free rotation symmetrical

Set value of barrier periodicity for conjugated single bond

Set value of δx used to calculate numerical derivatives

FIGURE 6.6. Pseudocode for Mininit1.dat

The bond stretching constants are calculated from the reference bond lengths and the B6 and B12 parameters are calculated from the B6 and B12 values for the individual atoms.

Mininit1 also makes the matrices (arrays) set up in **Mindat** containing reference values symmetrical. When constructed in **Mindat** these matrices are upper triangular. They are made symmetrical in order that when they are accessed it is immaterial which way round the indices are (i.e REFLEN(**x**,**y**) is equivalent to REFLEN(**y**,**x**)).

Then the values of various constants are set:-

- barrier periodicity for conjugated single bond
- value of δx used to calculate numerical derivatives
- bond length tolerance

After **Mindat** and **Mininit1** have been executed, the file is read which contains the atomic coordinates etc. of the molecule that has to be minimised. This allows arrays to

be initialised that contain information specific to the molecule. The first of these is constructed in **Asboml**.

This routine constructs a two dimensional array that contains pseudo bond orders for each bond in the molecule. A single bond is given a value of one, a double bond a value of two and a conjugated bond a value of 1.1 or 1.5 depending on the length of the bond.

The next routine to be called is **Getcop**. This reads a file containing a number of control parameters for the minimisation. These are:-

- number of iterations
- van der Waals cutoff distance
- energy threshold for printing
- maximum allowed shift
- long, abbreviated or short printed output
- constraints on atoms, lengths, angles, torsion angles and molecule

Getcop also assigns a value to NDERIV which determines whether the second derivatives are calculated every iteration or not.

Initialisation is completed by the routine **Getopb** which assigns out of plane bending constants for atoms that are subject to this constraint.

The routines which execute the calculation are **Bdmin, Pote** and **Energl. Bdmin** calls **Pote** which, using the equations detailed in chapter 5, calculates the initial potential energy of molecule. **Pote** also sets up the three two dimensional arrays NBMAT, NAMAT and NTMAT that contain bonded/nonbonded interactions for each atom pair, angles each atom is involved in and torsion angles each atom is involved in. These are used in **Energl**.

The entries in the NBMAT array are integer values that indicate the type of interactions between atom pairs. A value of two indicates the two atoms are bonded, a value of four indicates a nonbonded interaction, a value of three indicate a 1,3 interaction and a value of five indicates no interaction between the pair of atoms.

Each angle in the molecule is assigned a number and the array NAMAT contains the number(s) of the angle(s) each atom is involved in. The array NTMAT is analogous to NAMAT and is for torsion angles.

After **Pote** has finished, **Bdmin** calculates the derivatives and hence the corrected coordinates using the BDNR method, for each atom at a time. Once the specified number of iterations is complete the new potential energy is calculated by **Pote** and printed to the screen.

In order to calculate the derivatives **Bdmin** calls **Energl** which calculates the contribution to the steric energy from the jth atom. Since the derivatives are calculated by numerical methods, the steric energy is evaluated for each atom at (x,y,z), (x + δx, y ,z)), (x. y + δy, z), (x, y, z + δz), (x - δx, y, z)), (x, y - δy, z)), (x, y, z - δz), (x + δx, y + δy, z), (x + δx, y, z + δz), (x, y + δy, z + δz). The code in **Energl** is very similar to **Pote** except that it is only the steric energy of one atom that is calculated. The second derivatives are only calculated every 1st, 5th, 9th iteration or every iteration if specified in **Getcop**.

Chemmin is a sequential minimiser designed to run on PCs. Even if it was run on a fast workstation the speed of the minimisation on a large protein structure is not fast enough to give a good cost/performance ratio. Parallel versions of molecular mechanics have been constructed for use with arrays of transputers[12] (and other processors) and also using clusters of workstations to improve the cost/performance ratio.

6.3.2 Parallelisation Strategies for Energy Minimisation

As stated previously, many parallel versions of molecular dynamics have been implemented. This calculation is very similar to energy minimisation as it involves calculating the steric energy of all the atoms in a molecule and then calculating the first derivative of this energy (the second derivatives are not required). The parallelisation strategies used are therefore comparable to those used in molecular mechanics.

Swanson and Lybrand[8] parallelised the AMBER molecular modelling package by distributing the calculation of nonbonded energies and forces across a collection of Unix workstations linked by Ethernet. The reasoning behind this is that nonbonded calculations typically consume over 90% of the total execution time of an energy calculation compared to about 1% for the bonded forces.

AMBER calculates the nonbonded pair list (i.e. the atoms involved in nonbonded interactions) on an amino-acid residue basis (AMBER is only used for proteins). If the distance between any two atoms in two different residues is within the cutoff distance, then all atoms of each residue are considered to have pairwise nonbonded interactions, with some exceptions.

Swanson and Lybrand distributed the calculation evenly among the nodes by giving each node a portion of atoms which contained an equal number of nonbonded pairs. The nodes only work on the nonbonded interactions and the host works on the bonded interactions. This ensures that the host is finished calculating in sufficient time to receive the results back from the nodes.

By only allowing the host to calculate the bonded interactions this results in the host being idle for some time waiting for the results from the nodes. It would seem more efficient to include the host in calculation of some of the nonbonded forces and energies. Also by only including the nonbonded calculation on the nodes this means that the node code is significantly different from the sequential version of the code. Thus more work is required to parallelise the code.

This parallel molecular dynamics code does however give good efficiency when run on a network of workstations using PVM to implement message passing (~88% on four Indigos). It is also highly portable and has been run successfully on clusters of Silicon Graphics, IBM RS6000, DEC ALPHA, and HP workstations as well as CRAY T3D and Kendal Square KSR2 parallel supercomputers. When using Ethernet connections between the processors, interprocessor communication is slow compared to the high speed connections used in special purpose multiprocessor machines such as the CRAY T3D.

Vincent and Merz[9] parallelised the molecular dynamics calculation in AMBER by dividing the calculation of both the bonded and nonbonded interactions between the nodes. All message passing was compliant with the MPI (Message Passing Interface) Standard.

In this case the nonbonded interactions are divided between the nodes in a residue fashion (i.e. each node is responsible for a certain number of residues). If the residues are simply divided up evenly between the nodes then this results in a load imbalance

between the nodes as each node would be responsible for a different number of nonbonded pairs. This is due to the fact that AMBER calculates the nonbonded pair list by assuming that if any two atoms on different residues are within the specified distance, then all the atoms on the two residues are said to interact.

Vincent and Merz rectify the load imbalance by manually redistributing the pairs among the nodes after each has generated its own pair list. This involves each node sending a count of its nonbonded pair list to all the other nodes and from this determining a target pair count. The pairs are then redistributed among the nodes until the target pair count is reached. This results in an increase in the amount of interprocessor communication which, depending on the speed of the hardware links, will reduce the efficiency of the system.

Schweitzer et al.[10] parallelised the molecular mechanics MM2 package by splitting four computationally intensive subroutines over four processors on a shared memory computer. The subroutines were DVDWCG, DDIPOL, DOMGA and DRANG which calculate the derivatives for van der Waals energy, bond dipole interactions, torsional energy and stretching and bending energy respectively. Each subroutine executes on a separate node which results in uneven load balancing between the processors as some subroutines take longer than others. Using this method an improvement of only 50% in program execution speed is achieved.

The parallel version of CHEMMIN divides the data domain onto the available processors. The atoms are divided between the available nodes so that each node works on a 'slice' of atoms. For each atom both the bonded and nonbonded energies, and derivatives are calculated. Each node has a copy of the atomic coordinates of all the atoms, as some of the atoms in its 'slice' may interact with atoms on other nodes. The host calculates the initial and final steric energies of the molecule and the nodes execute the energy minimisation.

The nodes consider each atom in their 'slice' one at a time. For each atom the first and second partial derivatives of the steric energy with respect to the atomic coordinates are calculated. The atom's corrected coordinates are then computed using the Newton Raphson iteration. Once the corrected coordinates for all the atoms on a node have been computed, they are sent back to the host. The host assembles a complete set of

new 'improved' coordinates from the 'slices' returned by the nodes and broadcasts this set back to all the nodes ready for the next iteration.

In CHEMMIN the nonbonded interactions are calculated differently from in AMBER. They are not calculated by considering interactions between residues but instead each atom is considered in turn and if it is within a certain distance to any other atom then they are said to interact (i.e if two atoms on different residues are within the cutoff distance then all the atoms in the two residues are not assumed to interact).

The number of atoms allocated to each node is equal (some nodes have an extra atom if the number of atoms is not exactly divisible by the number of nodes). This gives efficient load balancing as the nonbonded interactions which take up most time, will be distributed reasonably evenly across the nodes (at least any differences in the number of nonbonded interactions are likely to be small compared to the total number of nonbonded interactions). This method of parallelisation was pioneered by White et al[12] and is similar to that of Vincent and Merz.

6.3.3 Hostmin and Nodemin

CHEMMIN is essentially divided into two parts: Hostmin which runs on the host (a PC) and Nodemin which runs on the nodes. The partition of the subroutines used in CHEMMIN between the host (Hostmin) and the nodes (Nodemin) is shown in Figure 6.7 on page 192. A full program listing of Hostmin and Nodemin is shown in Appendix C pages 255 - 282.

The subroutine **BDMIN** on the host is different from the version used in the sequential minimiser CHEMMIN. In the parallel version it is still responsible for calculating the initial and final steric energies. It does not however calculate the derivatives as it is the nodes that execute the minimisation. Instead it loads the node program onto the nodes and sends the required data to the nodes.

To calculate the first and second derivatives, the nodes require a substantial amount of data that the host has obtained or already calculated. Some of this information is sent to the nodes but most of it is recalculated by the nodes. This is thought to be quicker than sending this data to the nodes (if the node processors were extremely slow then this may not be the case).

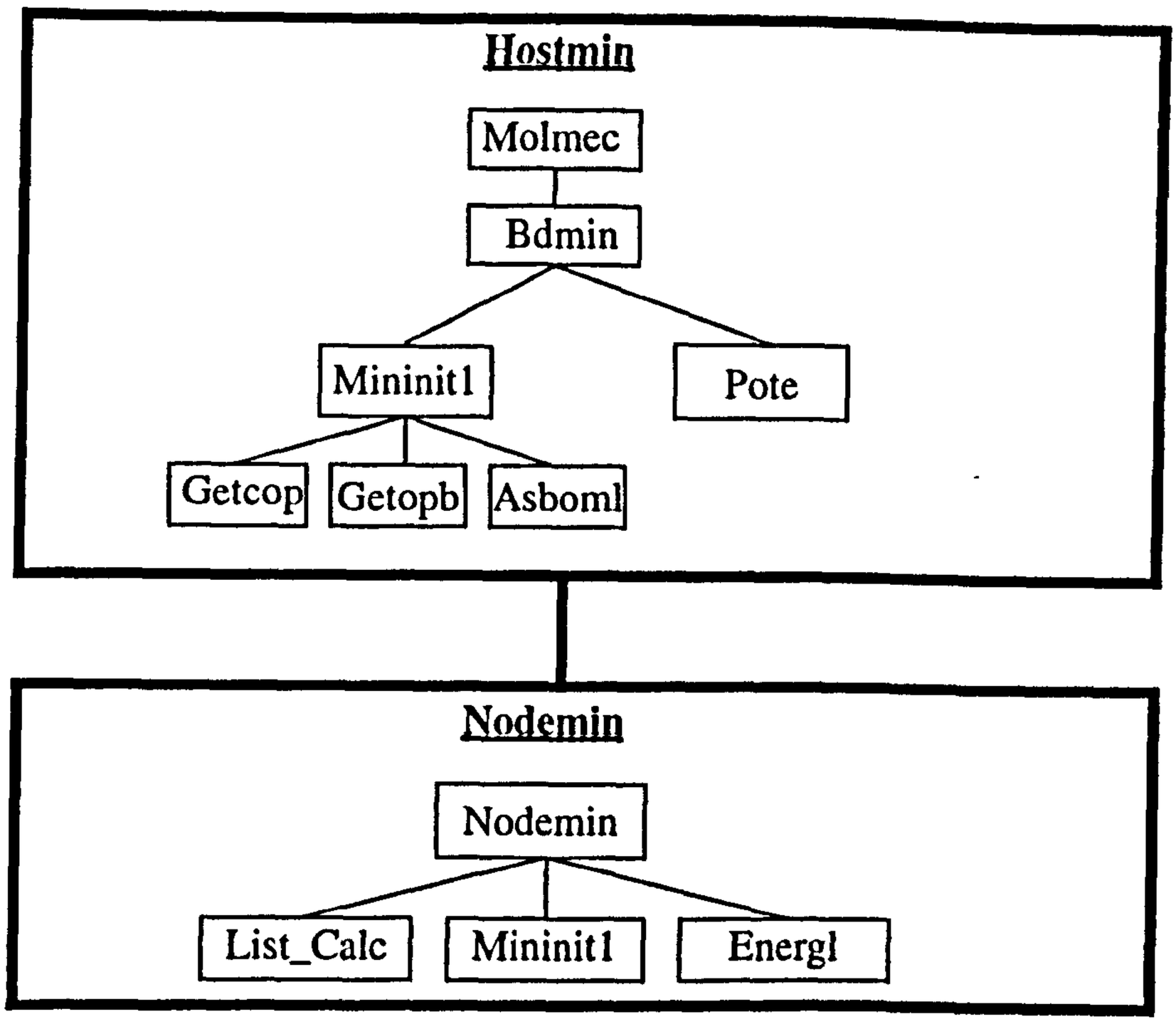


FIGURE 6.7. Partition of subroutines between host and nodes

The data downloaded to the nodes is mainly information about the molecule (number of atoms, bonds etc.), various constants, the tables containing fixed bond lengths, angles etc. and the atomic coordinates. From this data the other required information is calculated. The routine Nodemin on the nodes receives the data from the host, calculates the derivatives and sends the corrected coordinates back to the host.

The routine **List_Calc** on the nodes is basically a cut-down version of **Pote** that only calculates the arrays NBMAT, NAMAT and NTMAT that contain bonded/ nonbonded interactions for each atom pair, angles each atom is involved in and torsion angles each atom is involved in. **Energl** is identical to the version used in the sequential minimiser.

In the parallel version of the minimiser the routines **Mindat** and **Mininit1** were combined into the one routine **(Mininit1).**

Figure 6.9 and Figure 6.10 on page 194 show the pseudocode for Hostmin and Nodemin. The *NFIRST* and *LAST* variables referred to in Nodemin are the first and last atoms in the nodes slice. These are calculated using the code shown in Figure 6.8; *me* is the id number of the node, *NUMPROC* is the number of nodes and *NUMATS* is the

number of atoms. Each node is allocated *NUMATS/NUMPROC* atoms, with the first *NMOD* nodes being allocated an extra atom. This distributes the atoms as evenly as possible across the nodes and provides efficient node balancing. The number of atoms in a nodes 'slice' is stored in BFLENG.

```
NDIV = NUMATS / NUMPROC
NMOD = MOD (NUMATS,NUMPROC)

IF(me.lt.NMOD)THEN
  NFIRST = (me*NDIV)+me+1
  LAST = ((me+1)*NDIV)+me+1
ELSE IF(me.eq.NMOD)THEN
   NFIRST = (me*NDIV)+me+1
   LAST = ((me+1)*NDIV)+me
ELSE IF(me.gt.NMOD)THEN
   NFIRST = (me*NDIV)+NMOD+1
   LAST = ((me+1)*NDIV)+NMOD
ENDIF

BFLENG=((LAST+1)-NFIRST)
nfirst4 = (nfirst * 4) - 3
```

FIGURE 6.8. Code to allocate atoms to node

6.3.4 Implementation of host/node communication using COMFORT and the BB08

In a conventional 3L FORTRAN[11] (a Parallel Fortran) based implementation of parallel molecular mechanics the nodes would be connected together in a pipeline or more complex topology and the code would be loaded onto the nodes in the standard 'store and forward' manner[12]. Any data exchanged between the host and nodes will generally have to pass through one or more intermediate nodes before it reaches its destination.This requires the nodes to run communication tasks which reduces the raw computational power deliverable to the application.

```
Read file containing atomic coordinates
Read file containing various parameters
Set up various tables required for the calculation
(i.e. bond lengths, bond angles etc.)
Calculate the total potential energy of the molecule
Configure,reset,load and initialize nodes
Send arrays of data to the nodes
While (no.of iterations not complete) do
    Send atomic coordinates to nodes
    Receive modified coordinates from nodes
Calculate Final Steric Energy of the Molecule
```

FIGURE 6.9. Pseudocode for Hostmin

```
Initialize node
Receive data from host
Set up various tables required for the calculation
(i.e. bond lengths, bond angles etc.)
Decide which atoms the nodes will work on
Receive atomic coordinates from host
For J = NFIRST, LAST do
    Calculate Energy of Jth atom
    For k =1 , 3 do
        Increment kth coordinate of jth atom and recalculate energy
        Decrement kth coordinate and recalculate energy
        Calculate first derivative for kth coordinate
    For k = 1,3 do
        sum of squares of first derivatives =
        sum of squares of first derivatives + (first derivative for kth coordinate)^2
    If Mod(Iteration,4) = 0 then
        Calculate second derivatives for jth atom
        Calculate corrections to coordinates for jth atom
        For k = 1,3 do
            Calculate new value for kth coordinate of jth atom
Send modified coordinates and sum of squares of first derivatives back to host
```

FIGURE 6.10. Pseudocode for Nodemin

This overhead can be eliminated by using the BB08 board which allows code/data to be broadcast simultaneously onto all the nodes. Initial parallel versions of the

minimiser used a non-standard C library to implement host/node communication via the BB08[14]. The following sections describes how COMFORT was implemented instead of the this C library in Hostmin and Nodemin.

6.3.4.1 The Implementation of COMFORT in HOSTMIN

An early version of COMFORT was used. The host code was written in Microsoft (16-bit) FORTRAN(Chap 3, Ref. 9) (current versions of COMFORT use Microsoft 32-bit FORTRAN) and the node code with 3L parallel FORTRAN (the 3L FORTRAN node programs are configured with the stand alone FORTRAN run time library). Although this methodology is not without its problems (more on this later) it does result in reasonably portable programs[15].

COMFORT is only used in the communication between the host and the nodes: the rest of the code remains unchanged (compared to the earlier version that used the C library). Figure 6.11 on page 196 shows FORTRAN code to broadcast code and data onto the nodes.

The nodes are loaded with code via the configure, reset, load and initialize routines that were explained in Table 6.3. on page 182. The initialize routine is slightly different as it sends a matrix 'ProcConn' to the nodes. This tells each node the link interconnection topology between nodes (as explained in Section 6.2.2 on page 179 the latest versions of COMFORT use complete connectivity and the topology maps are unnecessary). The variable NETCAST used to specify the node id numbers in the SEND and LOAD routines is set to -1. This indicates a broadcast code or data onto all the nodes.

The data required by the nodes is sent in several arrays via the SEND routine which broadcasts the data to all the nodes simultaneously via the BB08 board. The variables/ arrays downloaded to the nodes are stored in common blocks. A sample of the common block declarations is shown in Figure 6.12 on page 196 and a definition of the variables is presented in Table 6.4. on page 197. Sending this data to the nodes is not as simple as it might first appear, mainly due to restrictions imposed by the Microsoft 16-bit FORTRAN which are not present with the 32-bit version.

```
NETCAST = -1
file= 'c:\comfort\lesley\min\nodemin.app'//char(0)
call configure(#180, 4, #976f)
call reset(NETCAST)
call load(NETCAST, file, 100, error)
do i=1,4
   ProcConn(1,i)=4
   ProcConn(2,i)=-1
   ProcConn(3,i)=-1
   ProcConn(4,i)=-1
end do
call initialize(ProcConn, 100, error)
call send (NETCAST,buffer_atmdat0,1,total_atmdat0,100,error)
call send (NETCAST,buffer_atmdat1,2,total_atmdat1,100,error)
call send (NETCAST,buffer_moldat,3,total_moldat,100,error)
call send (NETCAST,buffer_ffp,4,total_ffp,100,error)
call send (NETCAST,buffer_cffp,5,total_cffp,100,error)
call send (NETCAST,buffer_contrl,6,total_contrl,100,error)
call send (NETCAST,buffer_constn,7,total_constn,100,error)
C      SEND INTEGER*1 VARIABLES/ARRAYS SEPARATELY
       call send (NETCAST,ATYNUM,8,LENGTH9,100,ERROR)
       call send (NETCAST,BONDML,9,LENGTH10,100,ERROR)
       call send (NETCAST,MOLNUM,10,LENGTH9,100,ERROR)

999    write (5,*)'No of iterations =',itrcmp +1

C      SENDS COORDINATES TO NODES
       call send(NETCAST,XO1,42,INT2(length7),100,error)
```

FIGURE 6.11. Host Code that broadcasts arrays to node

```
      COMMON/ATMPRP/ EN(MAXTYP)
      COMMON/MOLDAT/ NUMATS,NMOLS
      COMMON/FILDAT/ DLUNIN,DLNOUT,LUNOUT
      COMMON/FILCHR/ INFILE,OUTFIL,FILTYP
      COMMON/HEADER/ TITLE
      COMMON/FFP/ REFLEN(MAXTYP,MAXTYP),STRCON(MAXTYP,MAXTYP)
     1,A6(MAXTYP,MAXTYP),B12(MAXTYP,MAXTYP),REFANG(MAXTYP)
     2,PERIOD(MAXTYP,MAXTYP),BARIER(MAXTYP,MAXTYP)
      COMMON/CFFP/ CREFLN(MXCNJ,MXCNJ),CSTCON(MXCNJ,MXCNJ)
     1,CBARR(MXCNJ,MXCNJ),CPRIOD
      COMMON/CONJTP/ ARTYPS(NARTYP),DBTYPS(NDBTYP)
```

FIGURE 6.12. Common Block Declarations

Variable Name	Definition
EN	Array containing electronegativity values
NUMATS, NMOLS	Number of atoms, number of molecules
DLUNIN, DLNOUT,LUNOUT	Unit file identifiers
INFILE, OUTFIL, FILTYP	File names
TITLE	Title of file
REFLEN	Array containing reference lengths
STRCON	Array containing stretching constants
A6,B12	Arrays containing A6 and B12 values for nonbonded energy
REFANG	Array containing reference angles
PERIOD	Array containing periodicity values
BARIER	Array containing barrier to free rotation values
CREFLN, CSTCON, CBARR, CPRIOD	Arrays containing reference lengths, stretching constants, barrier to free rotation, periodicity for conjugated bonds
ARTYPS, DBTYPS	Arrays containing values of aromatic and double bond atom types

TABLE 6.4. Variable names and definitions

The simplest approach may appear to be to send a large array whose start address is the address of the first variable in the first common block. The length (in bytes) of this array would be equal to the total length of all the common blocks. This approach is not possible as although the common blocks will be stored contiguously in memory, they are each assigned to a different 64kbyte wide segment by the FORTRAN compiler and addresses do not automatically roll over from one segment to the next. Also, unless the the molecule under investigation contained the maximum number of atoms, then the array would not be full resulting in a waste of space.

Another possible approach might be to dispense with the individual common blocks and put all of the data into one large common block. This is not possible as there is more than 64kbytes of data and the compiler limits each common block to a maximum of 64kbytes in length. In addition to this restriction the COMFORT SEND subroutine imposes a maximum message length of 64kbytes.

A dummy array is therefore EQUIVALENCED to the start of each common block (or the position in the common block where the required data starts). This dummy array is

dimensioned to encompass the data by calculating the combined size (in bytes) of all the variables/arrays required from the common block (See Figure 6.13).

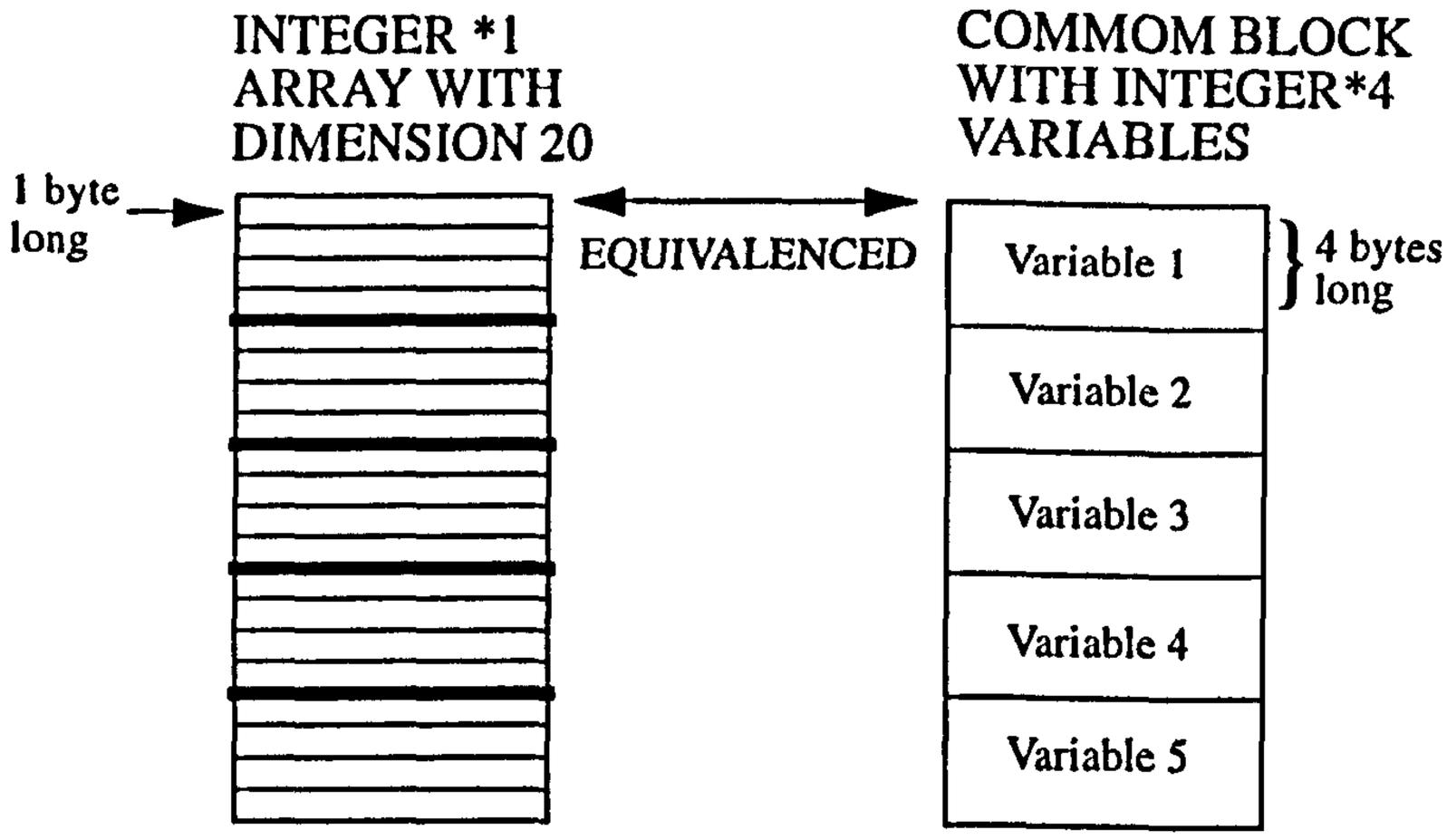


FIGURE 6.13. Graphical Representation of Equivalence Statements

The include file which EQUIVALENCES the arrays/variables to dummy arrays is illustrated in Figure 6.14 on page 199. In order to make the calculation of the lengths of the dummy arrays simpler the first block of PARAMETER statements assigns values to the different lengths (length1, length2 etc.). These lengths (in bytes) are of the arrays/variables stored in the common block. They are all multiplied by four as the arrays/variables are INTEGER*4.

The second block of parameter statements assign values to parameters that specify the length of the common blocks (i.e. common block 'contrl' contains eight variables of length3). The dummy arrays (buffer_atmdat0, buffer_atmdat1, etc.) are then dimensioned and EQUIVALENCED to the first variable in each common block (or the first variable that is required).

A further difficulty arises from the fact that the Microsoft FORTRAN compiler adheres rigidly to the FORTRAN standard. If the SEND subroutine is called with a message of one data type then any subsequent call with a message of a different data type will result in a run-time error. In order to overcome this difficulty SEND is always called with messages of type INTEGER*1 which are EQUIVALENCED to the real data array (which contains data of many types).

```
**************************************************************
c*EQUIVALENCES ARRAYS IN COMMON BLOCKS TO DUMMY ARRAYS     *
c*                                                          *
c**************************************************************

      integer length1,length2,length3,length4,length5,length6,
     1        length7,length8,length9,length10,length11

      integer*2 total_atmdat0,total_atmdat1,total_moldat,
     1          total_ffp,total_cffp,total_contrl,total_constn

      integer*1 buffer_atmdat0,xo1,buffer_atmdat1,buffer_moldat,
     1          buffer_ffp,buffer_cffp,buffer_contrl,buffer_constn,
     2          xo2,xo3

      parameter (length1 = MXAT*MXCN*4)
      parameter (length2 = MXAT*4)
      parameter (length3 = 4)
      parameter (length4 = MAXTYP*MAXTYP*4)
      parameter (length5 = MXCNJ*MXCNJ*4)
      parameter (length6 = MXAT*4*4)
      parameter (length7 = MXAT*3*4)
      parameter (length8 = MAXCNS*4)
      parameter (length9 = MXAT)
      parameter (length10 = MXAT * MXCN)
      parameter (length11 = MXAT)

      parameter (total_atmdat0 = length1)
      parameter (total_atmdat1 = length2)
      parameter (total_moldat = length3 * 2)
      parameter (total_ffp = length4)
      parameter (total_cffp = length5)
      parameter (total_contrl = length3 * 8)
      parameter (total_constn = (5 * length3) + length6 + length7
     1                        + length2 + (16 *length8))

      dimension buffer_atmdat0(1:total_atmdat0)
      dimension buffer_atmdat1(1:total_atmdat1)
      dimension buffer_moldat(1:total_moldat)
      dimension buffer_ffp(1:total_ffp)
      dimension buffer_cffp(1:total_cffp)
      dimension buffer_contrl(1:total_contrl)
      dimension buffer_constn(1:total_constn)
      dimension xo1(length2)
      dimension xo2(length2)
      dimension xo3(length2)

      equivalence (buffer_atmdat0(1),atmcon)
      equivalence (buffer_atmdat1(1),charge)
      equivalence (buffer_moldat(1),numats)
      equivalence (buffer_ffp(1),strcon)
      equivalence (buffer_cffp(1),cstcon)
      equivalence (buffer_contrl(1),shiftx)
      equivalence (buffer_constn(1),conmin)
      equivalence (xo1(1),xo(1,1))
      equivalence (xo2(1),xo(1,2))
      equivalence (xo3(1),xo(1,3))
```

FIGURE 6.14. Include file that equivalences arrays/variables to dummy arrays

Three of the variables/arrays that are stored in the common blocks are INTEGER*1. These are sent separately as if they were included in equivalence statements they would disrupt the alignment of the dummy array with the common block. In the case where the INTEGER*1 variable/array is in the middle of a common block, then two dummy arrays must be equivalenced to that common block: one starting at the beginning of the common block and ending before the INTEGER*1 variable/array and the other beginning after the INTEGER*1 variable/array and ending at the end of the common block.

6.3.4.2 The Implementation of COMFORT in Nodemin

The arrays/variables sent from the host are received using the COMFORT receive routine (See Figure 6.15). There must be an equivalent receive on the nodes for every send on the host.

```
C  INITIALIZE NODES
   call initialize
C  RECEIVES BUFFERS FROM HOST.
   call receive(host,buffer_atmdat0,1,total_atmdat0,error)
   call receive(host,buffer_atmdat1,2,total_atmdat1,error)
   call receive(host,buffer_moldat,3,total_moldat,error)
   call receive(host,buffer_ffp,4,total_ffp,error)
   call receive(host,buffer_cffp,5,total_cffp,error)
   call receive(host,buffer_contrl,6,total_contrl,error)
   call receive(host,buffer_constn,7,total_constn,error)

c      RECEIVE BYTE VALUES SEPARATELY

   call receive(HOST,ATYNUM,8,LENGTH9,ERROR)
   call receive(HOST,BONDML,9,LENGTH10,ERROR)
   call receive(HOST,MOLNUM,10,LENGTH9,ERROR)
    :
    :
 191 call receive(HOST,XO1,42,length7,error)
```

FIGURE 6.15. Code on node which receives data from host

The nodes also have a copy of the include file which equivalences the dummy arrays to the variables/arrays. This allows the nodes to effectively decode the information sent from the host.

6.3.4.3 Transfer of atomic coordinates between host and nodes

The atomic coordinates are stored in an INTEGER*4 array (XO (MXAT,3)) on the host which is effectively arranged as three columns one for each of the x,y and z coordinates. This array is EQUIVALENCED to three INTEGER*1 arrays X01,X02 and X03 (See Figure 6.14 on page 199); X01 contains the x coordinates, and X02,X03 the y and z coordinates respectively.

To send the atomic coordinates to the nodes the X01 array is used in the SEND routine (See Figure 6.12). X01 is EQUIVALENCED to the start of X0 and the buffer length in the SEND statement is four times the length of X0 (as X01,X02 and X03 are INTEGER*1 arrays). An equivalent RECEIVE statement is required on the nodes (See Figure 6.15 on page 200).

When sending the coordinates back from the nodes to the host only the coordinates in the node's 'slice' must be returned and the host must put the returned coordinates in the correct place in XO. The code on the nodes and host which achieves this is shown in Figure 6.16 and Figure 6.17 respectively.

```
call send(HOST,xo1(nfirst4),43,bfleng*4,error)
call send(HOST,xo2(nfirst4),44,bfleng*4,error)
call send(HOST,xo3(nfirst4),45,bfleng*4,error)
call send(HOST,sgd1sq,46,4,error)
```

FIGURE 6.16. Node code to return 'improved' coordinates to host

The x,y and z coordinates are sent separately in X01,X02 and X03. Nfirst4 specifies the position of the first atom in the nodes 'slice' in X01 etc. This value is not just equal to nfirst (the first atom in a nodes slice) as X01 etc. are INTEGER*1 arrays so the value of nfirst needs to be recalculated (i.e. nfirst4 = (nfirst*4) -3). The length of X01,X02 and XO3 is set to BFLENG *4; i.e the number of atoms in a nodes slice multiplied by 4.

```
do 321 l=0,numproc-1
      if(l.lt.nmod)then
       nfirst = (l*ndiv)+l+1
       last = ((l+1)*ndiv)+l+1
      else if(l.eq.nmod)then
       nfirst = (l*ndiv)+l+1
       last = ((l+1)*ndiv)+l
      else if(l.gt.nmod)then
        nfirst = (l*ndiv)+nmod+1
        last = ((l+1)*ndiv)+nmod
      endif

      bfleng =((last+1)-nfirst)
      nfirst = nfirst*4 - 3
C   RECALCULATE NFIRST FOR XO1(INTEGER*1 SIZE ARRAY)
      call receive(L,xo1(nfirst),43,INT2(bfleng*4),100,error)
      call receive(L,xo2(nfirst),44,INT2(bfleng*4),100,error)
      call receive(L,xo3(nfirst),45,INT2(bfleng*4),100,error)
      call receive(L,temp1,46,4,100,error)
      sgd1sq = sgd1sq + temp
321 continue
```

FIGURE 6.17. Host code to receive 'improved' coordinates

6.3.5 Minimisation times

Table 6.5. shows the run-time of the parallel minimiser on one node compared to four nodes for 24 and 45 atom molecules. The results illustrate that for a 24 atom molecule a speed-up of approximately 2.5 is obtained whereas for a 45 atom molecule a speed-up of approximately 3 is achieved. The difference in the results is due to the set-up time (i.e. the loading of the required data onto the nodes etc.) which becomes more significant for smaller numbers of atoms.

Number of atoms	Number of Nodes	Run-time of Minimiser
24	1	320s
24	4	129s
45	1	743s
45	4	243s

TABLE 6.5. Optimisation times for 30 iterations

One of the reasons that a speed up of closer to 4 is not achieved is that a large amount of redundant information is sent to the nodes. This is due to the fact that the dummy arrays sent to the nodes are dimensioned to encompass the maximum number of atoms. In the case of a small molecule the arrays would contain a large amount of zero values.

The problem could be overcome by calculating the size of the arrays based on the number of atoms. This would be feasible for one dimensional arrays but the situation is not so simple for multidimensional arrays due to the way arrays are arranged in memory.

Figure 6.18 shows how the array FATXYZ, that contains the coordinates of fixed atoms in the molecule, is arranged in memory. The parameter MXAT is the maximum number of atoms and the arrows in the diagram indicate the continuation of memory addresses.

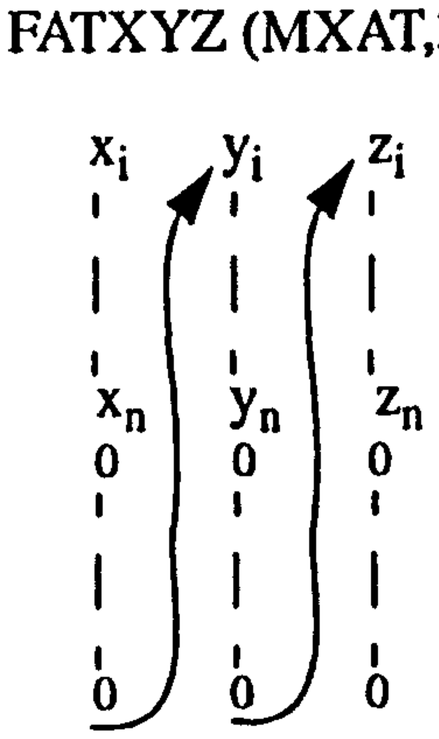


FIGURE 6.18. Arrangement of FATXYZ in memory

If this array was equivalenced to a one dimensional dummy array dimensioned NUMATS x 3, where NUMATS is the number of atoms, then the dummy array would not contain the correct data. This problem can be solved by reversing the indices of the array FATXYZ (See Figure 6.19). By arranging the array in this manner and equivalencing it to a dummy array with dimension NUMATS x 3, the dummy array will contain all the relevant data and no zeros.

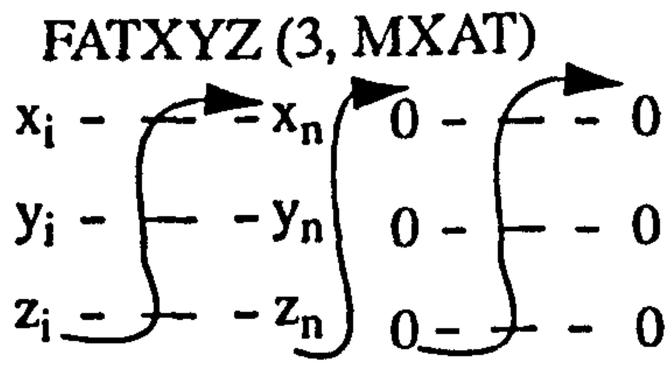


FIGURE 6.19. Arrangement of FATXYZ in memory with reversed indices

Since the arrays were already dimensioned with the first index as MXAT it was decided for the prototype version of the minimiser to leave the arrays the way they were and therefore to send the redundant data to the nodes. Changing the order of the indices

would involve altering every instance of the arrays in the program - a time consuming task.

6.4 Graphical Interface

In order to make the minimiser more user friendly, a graphical interface was developed that allows the user to enter parameters such as number of iterations, van der Waals cut-off distance etc. via selecting boxes on the screen. These were previously entered in a file that was read in the subroutine **Getcop**. In this version of the minimiser instead of calling **Getcop**, the routine **nrm_get_control_parmeters** is called which initiates the graphical interface. The graphics were drawn using the Microsoft FORTRAN graphics library (Chap. 3 Ref. 10).

Several screen shots of the graphical interface are shown in Appendix D pages 308 - 310. The pseudocode for it is shown in Figure 6.20 on page 206 and the FORTRAN code is shown in Appendix C pages 283 - 302.

The values of the parameters are entered via a number pad, which is printed at the side of the screen, once a parameter has been selected. In the case of fixing parameters, a stick representation of the molecule is drawn to allow the user to select atoms, lengths etc. The user also defines the severity of the restraint via the number pad. For lengths and angles the user enters the value via the number pad.

6.5 Conclusions

The use of the BB08 broadcast link interface and the COMFORT programming environment within a parallel molecular mechanics program has been detailed. An advantage of using the BB08 is that the data required by the nodes can be broadcast simultaneously to all the nodes and therefore no additional software is required on the nodes to manage the passing of the data. Also broadcasting the code and data to all the nodes at once reduces the run time of the minimisation considerably compared to the version that uses 3L FORTRAN and a pipeline of transputers.

```
Calculate position of option boxes and text
Calculate position of number pad boxes and text
While .NOT. start do
        Draw option box and text
        Show mouse cursor
        Find which box mouse clicked on
        If clicked on first box then
                Write message 'Enter no. of iterations' on screen
                Draw number pad boxes and text
                Decide what number was entered and store it
        Else If clicked on second box then
                Write message 'Enter van der Waals cutoff distance'
                Draw number pad boxes and text
                Decide what number was entered and store it
Else If clicked on third box and type of output is 'long' then
        Write 'Enter print threshold energy'
        Draw number pad boxes and text
        Decide what number was entered and store it
Else If clicked on fourth box then
        Write 'Enter maximum coordinate shift'
        Draw number pad boxes and text
        Decide what number was entered and store it
Else If clicked on fifth box then
        If type of output is 'short' then
                set type of output to 'abbreviated'
        Else If type of output is 'long' then
                set type of output to 'short'
        Else If type of output is 'abbreviated' then
                set type of output to 'long'
Else If clicked on sixth box then
        Calculate screen coordinates for molecule
        Draw simple stick molecule
        Label atoms with numbers
        Draw menu for fixing atoms or parameters
        Find which box in menu was selected
        If first box picked then
                Write 'Pick atom to fix'
                Find which atom selected
                Write 'Enter severity'
                Draw number pad boxes and text
                Decide what number was entered
                Enter coordinates of atom and severity of constraint in appropriate arrays
        Else If second box picked then
                Write 'Pick atoms defining fixed length'
                Find which atoms selected
                Write 'Choose value of fixed length'
                Draw number pad boxes and text
                Decide what number was entered
                Write 'Enter severity'
                Draw number pad boxes and text
                Decide what number was entered
                Enter coordinates of atoms, fixed value and severity of constraint in
                appropriate arrays
```

```
Else If third box picked then
        Write 'Pick atoms defining fixed angle'
        Find which atoms selected
        Write 'Choose value of fixed angle'
        Draw number pad boxes and text
        Decide what number was entered
        Write 'Enter severity'
        Draw number pad boxes and text
        Decide what number was entered
        Enter coordinates of atoms, fixed value and severity of constraint in
        appropriate arrays
    Else If fourth box picked then
            Write 'Pick atoms defining torsion angle'
            Find which atoms selected
            Write 'Choose value of torsion angle'
            Draw number pad boxes and text
            Decide what number was entered
            Write 'Enter severity'
            Draw number pad boxes and text
            Decide what number was entered
            Enter coordinates of atoms, fixed value and severity of constraint in
            appropriate arrays
    Else If fifth box picked then
            Write 'Pick any atom in molecule to fix'
            Find which atom selected
            Enter no.of molecule in appropriate array
Else If clicked on seventh box then
        start = .TRUE.
Else If clicked on eighth box then
        STOP
```

FIGURE 6.20. Pseudocode for graphical interface

Further improvements to this algorithm could include using the host to carry out the Newton Raphson iteration on a 'slice' of atoms rather than it remaining idle while the nodes are computing. Also in this version the nodes recalculate data for all the atoms where they only need to calculate the data for the atoms in their slice. For example the arrays NBMAT, NAMAT, NTMAT which contain the bonded/nonbonded interactions, angles and torsion angles respectively for each atom in the molecule need only be calculated for the atoms in the node's slice.

A more recent version of the software uses Microsoft Powerstation (32-bit) Fortran. Using this version with eight 486 PCs as nodes and a 45 atom molecule, gives a speed-up factor of 6.1 compared to a single node.

References

[1] Chem-X. Chemical Design Inc.

[2] Sybyl. Tripos Associates

[3] White, D.N.J. Computer methods for molecular design. *Phil. Trans. R. Soc. Lond.* 1986. A **317,** 359 - 369

[4] Weiner, P.K., Kollman, P.A., Amber: Assisted Model Building with Energy Refinement. A General Program for Modeling Molecules and their Interactions. *J.Comp.Chem* 1981, **2,** pp. 287-303

[5] Mertz, John.E., Tobias, Douglas, J., Brooks, Charles. L., Singh, U.C. Vector and Parallel Algorithms for the Molecular Dynamics Simulation of Macromolecules on Shared Memory Computers. *J.Comp.Chem* 1991, **12,** pp. 1270 - 1277

[6] Clark, Terry.W., McCammon, J.Andrew. Parallelisation of a Molecular Dynamics non-bonded force algorithm for MIMD architecture. *Computers & Chem.* 1990, Vol.**14,** No3, 219

[7] Hwang, Yuan-Shin., Das, Raja., and Saltz. Joel. H. Parallelizing Molecular Dynamics Programs for Distributed-Memory Machines. IEEE Computational Science and Engineering, Summer 1995, pp. 18-29

[8] Swanson, Eric., Lybrand, Terry P. PVM-AMBER: A Parallel Implementation of the AMBER Molecular Mechanics Package for Workstation Clusters. *J.Comp.Chem* 1995, **16,** pp. 1131 - 1140

[9] Vincent, James J., Merz, Kenneth M. A Highly Parallel Implementation of AMBER4 Using the Message Passing Interface Standard. *J.Comp.Chem* 1995, **16,** pp. 1420 - 1427

[10] Schweitzer, Robert.C., Small, Gary W., Application of Parallel Processing Techniques to Improving the Efficiency of the MM2 Molecular Mechanics Calculations. *J.Comp.Chem.* 1993, **14,** 977 - 985

[11] Parallel Fortran User Guide, 3L Ltd, 1990

[12] White, D.N.J., Ruddock, J. Noel., Edgington, Paul R. Molecular Design with Transparallel Supercomputers. Molecular Simulation 1989, Vol. 3, 71

[13] White, David N.J. A Hardware & Software Environment for Parallel Processing with PCs. In press, *Computers & Chemistry.*

[14] Harkins, Andrew. Molecular Mechanics Calculations using Parallel Computers. *Final Year Project.* 1992-93

[15] Bissland, Lesley., White, David N.J. Parallel Molecular Mechanics Calcualtions. *Transputer Applications and Systems '95.* Proceeding of the 1995 World Transputer Congress 1995, Harrogate, Sept. 95, pp. 473-487

Appendix A:

Source code for command line and graphical interfaces.

```
C*************************************************************************
C       NAME: C0041.FOR                                                  *
C                                                                        *
C       FUNCTION: A COMMAND LINE INTERFACE THAT ALLOWS CONNECTIONS TO*   *
C                 BE SPECIFIED BETWEEN PROCESSORS                        *
C*************************************************************************

      INTERFACE TO INTEGER*2 FUNCTION LINKIN
     1          [C,ALIAS:'_linkin'] (C)
      INTEGER*2 C
      END

      INTERFACE TO SUBROUTINE LINKOUT[C,ALIAS:'_linkout'] (A,B)
      INTEGER*2 A,B
                END

      INTERFACE TO SUBROUTINE RUN[C,ALIAS:'_run'] (E,F)
      INTEGER*2 E,F
      END

      PROGRAM PROCESSOR C004

      CHARACTER*10 PROC1,LINK1,PROC2,LINK2
      CHARACTER*80 STATEMENT
      CHARACTER*65 SUBSTA
      CHARACTER*1  ANS,SREP,REP
      LOGICAL TEST,ZERO0
                INTEGER IPROC1,ILINK1,ITO,IPROC2,ILINK2,
     1         LMAX,PMAX,NP1(32),NP2(32),NL1(32),NL2(32),
     1         C0040(32,6),C0041(32,6),LMIN,LKAD(4),N,J,
     1         C40IN(32),C41OUT(32),C40OUT(32),C41IN(32),
     1         X,Y,A3,A4,P1,P2,L1,L2,BADD,T,CPROC1,CPROC2,
     1         SPROC1,SPROC2,CLINK1,CLINK2,SLINK1,SLINK2,
     1         IN,OUTPUT,C4,M,P,SSREP,IN0,IN1
      INTEGER*2 LINKIN
      ZERO0 = .FALSE.

C     INSTRUCTIONS FOR ENTERING CONNECTIONS

      WRITE(*,*)
      WRITE(*,3)
3     FORMAT(20X,'LINK CONNECTIONS')
      WRITE(*,4)
4     FORMAT(20X,'*****************')
      WRITE(*,*)
                WRITE(*,6)
6     FORMAT(1X, 'ENTER THE LINKS BETWEEN THE PROCESSORS IN THE',1X,
     1       'FOLLOWING FORM:')
      WRITE(*,*)
                WRITE(*,7)
7     FORMAT(1X, 'CONNECT PROCESSOR A LINK B TO PROCESSOR',1X,
     1      'C LINK D')
      WRITE(*,*)
                WRITE(*,8)
8     FORMAT(1X, 'EACH STATEMENT MUST BE ON A NEW LINE AND',1X,
     1           'EACH WORD')
                WRITE(*,9)
9     FORMAT(1X, 'MUST BE TYPED IN ALL THE SAME CASE.')
      WRITE(*,21)
21    FORMAT(1X,'ONCE FINISHED TYPE ''QUIT''.')

C     SET COUNTER

                K = 0

C     READ STATEMENT OF CONNECTIONS

10              READ(*,'(A80)') STATEMENT

C     TEST FOR END

                IF ((STATEMENT.EQ.'QUIT').OR.(STATEMENT.EQ.'quit')) THEN
   GOTO 20
END IF
                K = K + 1

C                FIND POSITIONS OF BEGINNING OF KEY WORDS

                CPROC1 = INDEX(STATEMENT,'PROCESSOR')
SPROC1 = INDEX(STATEMENT,'processor')
                CLINK1 = INDEX(STATEMENT,'LINK')
SLINK1 = INDEX(STATEMENT,'link')
                CTO = INDEX(STATEMENT,'TO')
STO = INDEX(STATEMENT,'to')

IF (CPROC1.GT.0) THEN
    IPROC1 = CPROC1
  ELSE IF (SPROC1.GT.0) THEN
```

```
      IPROC1 = SPROC1
    ELSE
      IPROC1 = 0
  END IF
  IF (CLINK1.GT.0) THEN
      ILINK1 = CLINK1
    ELSE IF (SLINK1.GT.0) THEN
      ILINK1 = SLINK1
    ELSE
      ILINK1 = 0
  END IF
  IF (CTO.GT.0) THEN
      ITO = CTO
    ELSE IF (STO.GT.0) THEN
      ITO = STO
    ELSE
      ITO = 0
  END IF
```

```
                    SUBSTA = STATEMENT(ITO:80)
        CPROC2 = INDEX(SUBSTA,'PROCESSOR')
        SPROC2 = INDEX(SUBSTA,'processor')
                    CLINK2 = INDEX(SUBSTA,'LINK')
        SLINK2 = INDEX(SUBSTA,'link')

        IF (CPROC2.GT.0) THEN
            IPROC2 = CPROC2
          ELSE IF (SPROC2.GT.0) THEN
            IPROC2 = SPROC2
          ELSE
            IPROC2 = 0
        END IF
        IF (CLINK2.GT.0) THEN
            ILINK2 = CLINK2
          ELSE IF (SLINK2.GT.0) THEN
            ILINK2 = SLINK2
          ELSE
            ILINK2 = 0
        END IF

C       TEST FOR SPELLING MISTAKES

                    IF ((IPROC1.EQ.0).OR.(ILINK1.EQ.0).OR.
     1      (IPROC2.EQ.0).OR.(ILINK2.EQ.0).OR.(ITO.EQ.0)) THEN
                      WRITE(*,*)'SYNTAX ERROR. RE-ENTER STATEMENT'
           K = K - 1
           GOTO 10
                    END IF

C       FOR EACH ENTRY IN THE ARRAY STATEMENT FIND THE
C       RELEVANT PROCESSOR AND LINK NUMBERS


        PROC1 = STATEMENT((IPROC1 + 9):(ILINK1 - 1))
        LINK1 = STATEMENT((ILINK1 + 4):(ITO - 1))
        PROC2 = SUBSTA((IPROC2 + 9):(ILINK2 - 1))
                    LINK2 = SUBSTA((ILINK2 + 4):)

C       CALL SUBROUTINE WHICH WILL CONVERT CHARACTER VALUES INTO
C       INTEGER VALUES

                    CALL INTEG (K,PROC1,NP1)
                    CALL INTEG (K,LINK1,NL1)
                    CALL INTEG (K,PROC2,NP2)
                    CALL INTEG (K,LINK2,NL2)

C       CONSTANTS FOR MAXIMUM NUMBER OF PROCESSORS
C       PMAX = MAX NO OF PROCESSSORS
C       LMAX = MAX VALUE OF LINK NO
C       LMIN = MIN VALUE OF LINK NO

        PMAX = 31
        LMAX = 2
        LMIN = 1

        P1 = NP1(K)
        P2 = NP2(K)
        L1 = NL1(K)
        L2 = NL2(K)

        TEST = .FALSE.

C       CHECK IF PROCESSOR NUMBER IS TOO HIGH

        IF (NP1(K).GT.PMAX) THEN
           TEST = .TRUE.
           WRITE(*,11)
11         FORMAT(1X,'THE NUMBER FOR THE FIRST PROCESSOR',1X,
     1           'IS TOO HIGH.')
```

```
      END IF
      IF (NP2(K).GT.PMAX) THEN
         TEST = .TRUE.
         WRITE(*,12)
12       FORMAT(1X,'THE NUMBER OF THR SECOND PROCESSOR',1X,
     1           'IS TOO HIGH.')
      END IF

C     CHECK IF LINK NUMBER IS CORRECT

      IF ((NL1(K).GT.LMAX).OR.(NL1(K).LT.LMIN)) THEN
         TEST = .TRUE.
         WRITE(*,13)
13       FORMAT(1X,'THE NUMBER OF THE FIRST LINK IS',1X,
     1           'NOT CORRECT.')
      END IF
      IF ((NL2(K).GT.LMAX).OR.(NL2(K).LT.LMIN)) THEN
         TEST = .TRUE.
         WRITE(*,14)
14       FORMAT(1X,'THE NUMBER OF THE SECOND LINK IS',1X,
     1           'NOT CORRECT.')
      END IF

C     CHECK LINK NUMBERS ARE NOT EQUAL

      IF (NL1(K).EQ.NL2(K)) THEN
         TEST = .TRUE.
         WRITE(*,15)
15       FORMAT(1X,'THE LINK NUMBERS CANNOT BE',1X,
     1           'THE SAME.')
      END IF

C     CHECK LINK AND PROCESSOR NUMBER NOT USED
C     BEFORE

      DO 50 J = 1,K - 1
        IF(((P1.EQ.NP1(J)).AND.(L1.EQ.NL1(J))).OR.
     1     ((P1.EQ.NP2(J)).AND.(L1.EQ.NL2(J)))) THEN
          TEST = .TRUE.
          WRITE(*,22)
22       FORMAT(1X,'THE VALUES FOR THE FIRST PROCESSOR',1X,
     1           'AND LINK NUMBERS HAVE BEEN USED BEFORE.')
        END IF
        IF(((P2.EQ.NP1(J)).AND.(L2.EQ.NL1(J))).OR.
     1     ((P2.EQ.NP2(J)).AND.(L2.EQ.NL2(J)))) THEN
          TEST = .TRUE.
          WRITE(*,23)
23        FORMAT(1X,'THE VALUES FOR THE SECOND',1X,
     1           'PROCESSOR AND LINK NUMBERS HAVE BEEN.')
          WRITE(*,*) 'USED BEFORE'
        END IF
50    CONTINUE

      IF (TEST) THEN
          WRITE(*,*) 'PLEASE RE-ENTER STATEMENT.'
          K = K-1
      END IF

      GOTO 10

C     FIND CORRESPONDING LINK NUMBERS ON CO04

20    CALL TABLE(CO041,CO040)
      CALL LINKS(K,NP1,NL1,CO040,CO041,
     1           C41IN,C41OUT,C40IN,C40OUT)
      CALL LINKS(K,NP2,NL2,CO040,CO041,
     1           C41IN,C41OUT,C40IN,C40OUT)

C     CREATE AN ARRAY CONTAINING BASE ADDRESSES
C     OF LINK ADAPTORS

      LKAD(1) = 384
      LKAD(2) = 388
      LKAD(3) = 392
      LKAD(4) = 396

C     SET BOARD ADDRESS

      BADD = 424

C     PUT BOARD INTO RUN STATE

      CALL RUN(1,BADD)
      DO 60 T = 1,5000
60    CONTINUE
      CALL RUN(0,BADD)

C     CALL SUBROUTINE TO OUTPUT BYTE AT LINK ADAPTOR
```

```
      CALL LINKOUT(4,LKAD(1))
      CALL LINKOUT(4,LKAD(2))

      DO 30 N = 1,K
         CALL LINKOUT(0,LKAD(1))
         CALL LINKOUT(C40IN(N),LKAD(1))
         CALL LINKOUT(C40OUT(N),LKAD(1))
         CALL LINKOUT(3,LKAD(1))
         CALL LINKOUT(0,LKAD(2))
         CALL LINKOUT(C41IN(N),LKAD(2))
         CALL LINKOUT(C41OUT(N),LKAD(2))
         CALL LINKOUT(3,LKAD(2))
30    CONTINUE

C     OFFER OPTION TO INTERROGATE COO4

55    WRITE(*,*) 'DO YOU WANT TO INTERROGATE COO4S?(Y/N)'
      READ(*,'(A)') REP
      WRITE(*,*) REP
      IF ((REP.NE.'N').AND.(REP.NE.'n').AND.
     1     (REP.NE.'Y').AND.(REP.NE.'y')) THEN
         GOTO 55
      END IF
      IF ((REP.EQ.'N').OR.(REP.EQ.'n')) THEN
         GOTO 110
      END IF
70    WRITE(*,*) 'WHICH COO4 DO YOU WANT TO INTERROGATE?'
      READ(*,*) C4
      IF ((C4.NE.0).AND.(C4.NE.1)) THEN
         GOTO 70
      END IF
80    WRITE(*,*) 'WHICH OUTPUT TO YOU WANT TO INTERROGATE?'
      READ(*,*) OUTPUT
      IF (OUTPUT.GT.31) THEN
         WRITE(*,*) 'NUMBER IS TOO HIGH.'
         GOTO 80
      END IF

      IF (C4.EQ.0) THEN
         CALL LINKOUT(2,LKAD(1))
         CALL LINKOUT(OUTPUT,LKAD(1))
         IN = LINKIN(LKAD(1))
         IF (((128 - IN).LT.0).OR.((128 - IN).EQ.0)) THEN
             IN = IN - 128
             WRITE(*,*) 'THIS OUTPUT IS CONNECTED TO INPUT',IN
         ELSE
             WRITE(*,*) 'THIS OUTPUT IS NOT CONNECTED'
         END IF
        ELSE
         CALL LINKOUT(2,LKAD(2))
         CALL LINKOUT(OUTPUT,LKAD(2))
         IN = LINKIN(LKAD(2))
         IF (((128 - IN).LT.0).OR.((128 - IN).EQ.0)) THEN
             IN = IN - 128
             WRITE(*,*) 'THIS OUTPUT IS CONNECTED TO INPUT',IN
         ELSE
             WRITE(*,*) 'THIS OUTPUT IS NOT CONNECTED'
         END IF
      END IF


90    WRITE(*,*) 'DO YOU WANT TO INTERROGATE THE COO4S FURTHER?(Y/N)'
      READ(*,'(A)') SREP
      IF ((SREP.NE.'N').AND.(SREP.NE.'n').AND.
     1     (SREP.NE.'Y').AND.(SREP.NE.'y')) THEN
          GOTO 90
      END IF
      IF ((SREP.EQ.'Y').OR.(SREP.EQ.'y')) THEN
         GOTO 70
      END IF

C     DISPLAY CONNECTIONS MADE BETWEEN PROCESSORS

110   WRITE(*,25)
25    FORMAT(1X,'DO YOU WANT TO SEE ALL THE CONNECTIONS',1X,
     1        'BETWEEN THE PROCESSORS?(Y/N)')
      READ(*,'(A)') SSREP
      IF ((SSREP.NE.'N').AND.(SSREP.NE.'n').AND.
     1     (SSREP.NE.'Y').AND.(SSREP.NE.'y')) THEN
          GOTO 110
      END IF
      IF ((SSREP.EQ.'N').OR.(SSREP.EQ.'n')) THEN
          GOTO 100
      END IF
C     FIND WHICH INPUT EACH OUTPUT IS CONNECTED TO
C     TEST MSB TO SEE IF SET
```

```
40      DO 130 P = 1,32
          CALL LINKOUT(2,LKAD(1))
          CALL LINKOUT(COO40(P,6),LKAD(1))
          IN0 = LINKIN(LKAD(1))
          CALL LINKOUT(2,LKAD(2))
          CALL LINKOUT(IN0,LKAD(2))
          IN1 = LINKIN(LKAD(2))
          IF (((128 - IN0).LT.0).OR.((128 - IN0).EQ.0)) THEN
               IN0 = IN0 - 128
               IN1 = IN1 - 128
               WRITE(*,26) COO40(P,5),COO40(P,4),COO40(IN0 + 1,2),
     1                     COO40(IN0 + 1,1)
26             FORMAT(1X,'PROCESSOR',1X,I2,1X,'LINKIN',1X,I2,1X,
     1          'IS CONNECTED TO PROCESSOR',1X,I2,1X,
     1           'LINKOUT',1X,I2)
               WRITE(*,26) COO41(IN0 + 1,5),COO41(IN0 + 1,4),
     1                     COO41(IN1 + 1,2),COO41(IN1 + 1,1)
          END IF
130     CONTINUE

C       CODE FOR TESTING COO4

100     WRITE(*,*) 'DO YOU WANT TO TEST THE C004''s?(Y/N)'
        READ(*,'(A1)') ANS

C       TEST FOR ANSWER

        IF ((ANS.EQ.'N').OR.(ANS.EQ.'n')) THEN
           STOP
        END IF
        WRITE(*,16)
16      FORMAT(1X,'ENTER NUMBER YOU WANT TO SEND TO LINK',1X,
       1           'ADAPTOR 2')
        READ(*,*) X
        CALL LINKOUT(X,LKAD(3))
        WRITE(*,17)
17      FORMAT(1X,'ENTER NUMBER YOU WANT TO SEND TO LINK',1X,
       1        'ADAPTOR 3')
        READ(*,*) Y
        CALL LINKOUT(Y,LKAD(4))

        A3 = LINKIN(LKAD(3))
        WRITE(*,18) A3
18      FORMAT(1X,'THE NUMBER RECEIVED AT LINK ADAPTOR',1X,
       1       '2 WAS',I3)

        A4 = LINKIN(LKAD(4))
        WRITE(*,19) A4
19      FORMAT(1X,'THE NUMBER RECEIVED AT LINK ADAPTOR',1X,
       1         '3 WAS',I3)

        END

        SUBROUTINE INTEG(D,DUMR,X)

        CHARACTER*10 DUMR
        INTEGER D,X(*)

C       CONVERT CHARACTERS INTO INTEGERS

        READ(DUMR,'(I10)') X(D)

        END

        SUBROUTINE TABLE(COO41,COO40)

        INTEGER COO41(32,6),COO40(32,6)

C       SET UP ARRAYS CONTAINING CONNECTIONS
C       ON COO4-1 AND COO4-0

        DO 10 I = 1,32
             COO41(I,1) = 1
             COO41(I,2) = I - 1
             COO41(I,3) = I - 1
             COO41(I,4) = 2
             COO41(I,5) = I - 1
             COO41(I,6) = I - 1
             COO40(I,1) = 2
             COO40(I,2) = I - 1
             COO40(I,3) = I - 1
             COO40(I,4) = 1
             COO40(I,5) = I - 1
             COO40(I,6) = I - 1

10           CONTINUE
             END
```

```
      SUBROUTINE LINKS(A,PNO,LNO,ARRAY0,ARRAY1,C41I,
     1                 C41O,C40I,C40O)

      INTEGER A,J,I,PNO(32),LNO(32),ARRAY0(32,6),ARRAY1(32,6),
     1        C41I(32),C41O(32),C40I(32),C40O(32)
        DO 10 I = 1,A
          DO 20 J = 1,32
            IF((LNO(I).EQ.ARRAY1(J,1)).AND.(PNO(I).EQ.ARRAY1(J,2)))
     1        THEN
              C41I(I) = ARRAY1(J,3)
            END IF
            IF ((LNO(I).EQ.ARRAY1(J,4)).AND.(PNO(I).EQ.ARRAY1(J,5)))
     1        THEN
              C41O(I) = ARRAY1(J,6)
            END IF
            IF((LNO(I).EQ.ARRAY0(J,1)).AND.(PNO(I).EQ.ARRAY0(J,2)))
     1        THEN
              C40I(I) = ARRAY0(J,3)
            END IF
            IF((LNO(I).EQ.ARRAY0(J,4)).AND.(PNO(I).EQ.ARRAY0(J,5)))
     1        THEN
              C40O(I) = ARRAY0(J,6)
            END IF
20        CONTINUE
10      CONTINUE
        END

C       SUBROUTINE LINKOUT(A,B)
C       INTEGER*2 A,B
C       WRITE(*,*) A,B
C       END
```

```
C*************************************************************************
C       NAME: GRAPH.FOR                                                  *
C                                                                        *
C       FUNCTION: A GRAPHICAL INTERFACE THAT ALLOWS CONNECTIONS TO       *
C                 BE DRAWN BETWEEN PROCESSORS                            *
C*************************************************************************

      INCLUDE 'MOUSE.FI'
      INCLUDE 'FGRAPH.FI'

                  INTERFACE TO INTEGER*2 FUNCTION LINKIN
     1          [C,ALIAS:'_linkin'] (BASEADDRESS)
      INTEGER*2 BASEADDRESS
      END

      INTERFACE TO SUBROUTINE LINKOUT[C,ALIAS:'_linkout']
     1                        (OUTPUTBYTE,BASEADDRESS)
      INTEGER*2 OUTPUTBYTE,BASEADDRESS
                  END

      INTERFACE TO SUBROUTINE RUN[C,ALIAS:'_run']
     1                                (OUTPUTBYTE,BOARDADDRESS)
      INTEGER*2 OUTPUTBYTE,BOARDADDRESS
      END

                  PROGRAM GRAPHICS19


      INCLUDE 'GRAPH.INC'

      INTEGER*4 NUMARGS
      INTEGER*2 N,STATUS,K,J
      CHARACTER*3 BUFFER

C     SET RESET ADDRESS AND BASE ADDRESSES FOR LINK ADAPS

      BADD = 424
      LKAD(1) = 384
      LKAD(2) = 388
      LKAD(3) = 392
      LKAD(4) = 396
      NUMARGS = NARGS()

C     SET FLAGS FOR PIPELINE AND WHETHER BETWEEN 1 AND 3 OR
C     1 AND 2

      LINKS = .FALSE.
      PIPE = .FALSE.

C     LOOK AT COMMAND LINE AND SEE IF INSTRUCTIONS
C     WANTED AND/OR PIPELINE SPECIFIED

      DO 40 K = 1,3
         CALL GETARG(K,BUFFER,STATUS)
         IF (STATUS.NE.-1) THEN
             IF (BUFFER.EQ.'\I') THEN
                CALL INSTRUCTIONS()
             END IF
             IF (BUFFER.EQ.'\HP') THEN
                 PIPE = .TRUE.
                 DO 20 J = 1,3
                    CALL GETARG(J,BUFFER,STATUS)
                    IF (STATUS.NE.-1) THEN
                       IF (BUFFER.EQ.'\03') THEN
                          LINKS = .TRUE.
                       END IF
                    END IF
20                CONTINUE
             END IF
          END IF
40    CONTINUE

C     PUT INTO GRAPHICSMODE AND DRAW ENDBOX,PROCESSORS
C     AND PIPELINE

      CALL GRAPHICSMODE()
      CALL ENDBOX(50,35)
      CALL DRAWPROCESSOR()
      IF (PIPE) THEN
         CALL PIPELINE()
      END IF

C     INITIALISE MOUSECURSOR

10    CALL MOUSE()

C     FIND WHERE MOUSE HAS BEEN PRESSED
```

```
      CALL PRESSMOUSE()

C     CALL ROUTINE TO SET UP CONNECTIONS ON COO4S

      CALL COO4CONNECTION()

C     CALL ROUTINE TO FIND CONNECTIONS ON COO4

      CALL CONNECTIONS(NP1,NL1)
      CALL CONNECTIONS(NP2,NL2)

C     PUT BOARD INTO RUN STATE

      CALL RUN(1,BADD)
      DO 60 T = 1,5000
60    CONTINUE
      CALL RUN(0,BADD)

C     CALL SUBROUTINE TO OUTPUT BYTE AT LINK ADAPTOR

      CALL LINKOUT(4,LKAD(1))
      CALL LINKOUT(4,LKAD(2))

      DO 30 N = 1,LINKNO
        CALL LINKOUT(0,LKAD(1))
        CALL LINKOUT(C40IN(N),LKAD(1))
        CALL LINKOUT(C40OUT(N),LKAD(1))
        CALL LINKOUT(3,LKAD(1))
        CALL LINKOUT(0,LKAD(2))
        CALL LINKOUT(C41IN(N),LKAD(2))
        CALL LINKOUT(C41OUT(N),LKAD(2))
        CALL LINKOUT(3,LKAD(2))
30    CONTINUE

C     CALL SUBROUTINE TO INTERROGATE COO4S

      CALL INTERROGATE()

C     CALL SUBROUTINE TO DISPLAY CONNECTIONS

      CALL DISPLAYCONNECTIONS()

C     CALL ROUTINE TO TEST COO4

      CALL TESTCOO4()

      END

C     ****WRITE OUT INSTRUCTIONS****

      SUBROUTINE INSTRUCTIONS()

      WRITE(*,5)
5     FORMAT(20X,'COO4 PROGRAMMER')
      WRITE(*,10)
10    FORMAT(20X,'*****************')
      WRITE(*,*)
      WRITE(*,20)
20    FORMAT(1X,'TO MAKE A CONNECTION BETWEEN TWO',1X,
     1          'PROCESSORS YOU CLICK ON THE LINK')
      WRITE(*,30)
30    FORMAT(1X,'YOU WANT TO USE, HOLD DOWN THE MOUSE',1X,
     1          'BUTTON AND DRAG THE')
      WRITE(*,40)
40    FORMAT(1X,'MOUSE TO WHERE YOU WANT IT TO BE RELEASED.',1X,
     1          'IF YOU WANT, SEVERAL')
      WRITE(*,50)
50    FORMAT(1X,'LINES CAN BE USED TO MAKE A CONNECTION. IF',1X,
     1          'YOU WANT TO DELETE')
      WRITE(*,60)
60    FORMAT(1X,'A CONNECTION THEN CLICK ON EITHER END OF ',1X,
     1          'THE CONNECTION.')
      WRITE(*,70)
70    FORMAT(1X,'PRESS RETURN TO CONTINUE.')

      READ(*,*)

      END


C     ****SET GRAPHICS MODE****

      SUBROUTINE GRAPHICSMODE()

      INCLUDE 'FGRAPH.FD'
      INCLUDE 'GRAPH.INC'

      RECORD/VIDEOCONFIG/MYSCREEN
```

```
                    INTEGER * 2 MODESTATUS,DUMMY,STATUS

C       SET VIDEOMODE TO MAXRESOLUTION

        MODESTATUS = SETVIDEOMODE($MAXRESMODE)
        IF (MODESTATUS.EQ.0) STOP 'ERROR :
     1     CANNOT SET GRAPHICS MODE'

        CALL CLEARSCREEN($GCLEARSCREEN)

C       SET FONTS

        DUMMY = REGISTERFONTS('C:\MSF\LIB\*.FON')

        IF (DUMMY.LT.0) THEN
         STOP 'ERROR:CANNOT FIND FONT FILES'
        END IF

        STATUS = SETFONT('T'COURIER'*//'h10w8b')

        IF (STATUS.LT.0) THEN
           STOP 'ERROR:CANNOT SET FONT'
        END IF

C       FIND RESOLUTION OF SCREEN

        CALL GETVIDEOCONFIG(MYSCREEN)
        MAXX = MYSCREEN.NUMXPIXELS - 1
        MAXY = MYSCREEN.NUMYPIXELS - 1

C       SCALE TO 1000

                    SCALEY = FLOAT(MAXY) / 1000.0
                    SCALEX = SCALEY * (FLOAT(MAXX) / FLOAT(MAXY)) * (3.0 / 4.0)
        END

C        ****ENDGRAPHICS****

        SUBROUTINE ENDGRAPHICS()

        INCLUDE 'FGRAPH.FD'
        INCLUDE 'GRAPH.INC'

        INTEGER * 2 MODESTATUS

        MODESTATUS = SETVIDEOMODE($DEFAULTMODE)

        END

C       ****FIND XCOORDINATES FROM SCREEN COORDS****

        INTEGER*2 FUNCTION NEWX(XCOORD)

        INTEGER XCOORD

        INCLUDE 'GRAPH.INC'

        REAL TEMPX

        TEMPX = FLOAT(XCOORD) * SCALEX
        NEWX = INT2(TEMPX + 0.5)
        END

C       ****FIND SCREEN COORDS FROM XCOORDS****

        INTEGER*2 FUNCTION CONVERTX(XCOORD)

        INTEGER*2  XCOORD

        INCLUDE 'GRAPH.INC'

        REAL TEMPX

        TEMPX = FLOAT(XCOORD) / SCALEX
        CONVERTX = INT2(TEMPX + 0.5)
        END

C       ****FIND YCOORDS FROM SCREEN COORDS****

        INTEGER*2 FUNCTION NEWY(YCOORD)

        INTEGER YCOORD

        INCLUDE 'GRAPH.INC'

        REAL TEMPY

        TEMPY = FLOAT(YCOORD) * SCALEY
```

```
      NEWY = INT2(TEMPY + 0.5)
      END

C     ****FIND SCREEN COORDS FROM YCOORDS****

      INTEGER*2 FUNCTION CONVERTY(YCOORD)

      INTEGER*2 YCOORD

      INCLUDE 'GRAPH.INC'

      REAL TEMPY

      TEMPY = FLOAT(YCOORD) / SCALEY
      CONVERTY = INT2(TEMPY + 0.5)
      END

C     ****DRAW BOXES FOR PROCESSORS****

      SUBROUTINE BOX(XCENT,YCENT,PROCNO)

                  INTEGER XCENT,YCENT
      INTEGER PROCNO

      INCLUDE 'GRAPH.INC'
                  INCLUDE 'NEWXY.INC'
      INCLUDE 'FGRAPH.FD'

                  INTEGER *2 STATUS
      CHARACTER*2 CHAR
      RECORD/XYCOORD/XY

C     MAKE PROCESSOR NUMBER A CHARACTER

      WRITE (CHAR,'(I2)') PROCNO
      CALL COLOUR(9)

C     DRAWBOX

      STATUS = RECTANGLE($GBORDER,NEWX(XCENT - 50),NEWY(YCENT - 50),
     1             NEWX(XCENT + 50),NEWY(YCENT + 50))

      CALL COLOUR(12)

C     PUT  PROCESSOR NUMBER IN BOX

      CALL TEXT(XCENT - 19,YCENT - 11,CHAR)

      END

C     ****DRAW LINES****

      SUBROUTINE DRAWLINE(STARTX,STARTY,ENDX,ENDY)

      INCLUDE 'GRAPH.INC'
                  INCLUDE 'NEWXY.INC'
      INCLUDE 'FGRAPH.FD'

      INTEGER STARTX,STARTY,ENDX,ENDY
      RECORD/XYCOORD/XY
      INTEGER*2 LINE

C     MOVE CURSOR TO WHERE YOU WANT LINE TO START
C     AND THEN DRAW LINE TO NEW POSITION

      CALL MOVETO(NEWX(STARTX),NEWY(STARTY),XY)
      LINE = LINETO(NEWX(ENDX),NEWY(ENDY))
      END

C     ****INITIALISE AND SHOW MOUSE****

      SUBROUTINE MOUSE

      INCLUDE 'MOUSE.FD'
      INCLUDE 'FGRAPH.FD'
      INCLUDE 'GRAPH.INC'
      INCLUDE 'NEWXY.INC'

      INTEGER START
      INTEGER*2 BUTTONS

      CALL COLOUR(7)
      START = INITIALISEMOUSE(BUTTONS)

      IF (START.EQ.0) THEN
         CALL ENDGRAPHICS()
```

```
            WRITE(*,*) 'MOUSE DRIVER NOT INSTALLED'
          END IF


          CALL SHOWMOUSECURSOR()

          END

C         ****CHECK FOR MOUSE PRESSES AND THEN TAKE
C             APPROPRIATE  ACTION****

          SUBROUTINE PRESSMOUSE()

          INCLUDE 'MOUSE.FD'
          INCLUDE 'FGRAPH.FD'
          INCLUDE 'GRAPH.INC'
          INCLUDE 'NEWXY.INC'
            INTEGER I,XPOINT,YPOINT,K,IXPOS,IYPOS,OLDXPOS,
         1             OLDYPOS,NIXPOS,NIYPOS,COLUMN,N,COUNTER,CONNO
            INTEGER*2 XPOS,YPOS,BPOS,BCOUNT

            LINKNO = 0
            COLUMN = 2
            COUNTER = 0

            CONNECT = .TRUE.
            CHOOSELINK1 = .TRUE.
            CHOOSELINK2 = .TRUE.
            CHANGELINKNO = .TRUE.

            CALL COLOUR(11)

10          CALL GETMOUSECURSORPOSITION(XPOS,YPOS,BPOS)

C           IF BUTTON PRESSED DOWN

            IF (BPOS.EQ.1) THEN
                CALLMOUSETRUE = .FALSE.

C               FIND SCRENN COORDS OF MOUSE POSITION

                XPOINT = CONVERTX(INT2(XPOS))
                YPOINT = CONVERTY(INT2(YPOS))

C               IF IN MIDDLE OF CONNECTION THEN MAKE
C               INITIAL COORDINATES PREVIOUS ONES

                IF (.NOT.CONNECT) THEN
                   XPOINT = NIXPOS
                   YPOINT = NIYPOS
                END IF

C               IF MADE BAD CONNECTION AND CONNECTION
C               OF MORE THAN ONE  LINE THEN MAKE INITIAL
C               COORDS AT END OF PREVIOUS LINE

                IF (.NOT.CHANGELINKNO) THEN
                    XPOINT = POINTS(LINKNO,COLUMN - 2)
                    YPOINT = POINTS(LINKNO,COLUMN - 1)
                END IF

C               STORE LINKNO BEFORE YOU FIND NEW LINKNO
                CONNO = LINKNO

C               CHECK TO SEE IF MOUSE HAS BEEN PRESSED AND RELEASED
C               ON A PROCESSOR AND STORE PROCESSOR NUMBER AND LINKNO
C               IF IT HAS
                CALL FINDPROCNO1(NP1,NL1,XPOINT,YPOINT)

C               IF MOUSE PRESSED ON PROCESSOR BUT NOT ON LINK THEN
C               EXIT LINK

                IF (.NOT.CHOOSELINK1) THEN
                   GOTO 10
                END IF

C               CHECK TO SEE WHETHER END BOX COULD HAVE BEEN PRESSED

                 IF ((CONNO.EQ.LINKNO).AND.(COUNTER.EQ.0)) THEN
                    IF ((XPOINT.LT.110).AND.(XPOINT.GT.2).AND.
         1              (YPOINT.LT.52).AND.(YPOINT.GT.10)) THEN
                        CALL CLEARSCREEN($GCLEARSCREEN)
                        CALL ENDGRAPHICS
                        STOP
                        RETURN
                     END IF
```

```
            GOTO 10
            END IF

C           IF PIPELINE CHOSEN CHECK THAT LINKS USED FOR
C           PIPELINE WERE NOT PRESSED

            IF ((PIPE).AND.(LINKS)) THEN
               IF ((NL1(LINKNO).EQ.0).OR.(NL1(LINKNO).EQ.3)) THEN
                  LINKNO = LINKNO - 1
                  GOTO 10
               END IF
            END IF
            IF ((PIPE).AND.(.NOT.LINKS)) THEN
               IF ((NL1(LINKNO).EQ.1).OR.(NL1(LINKNO).EQ.2)) THEN
                  LINKNO = LINKNO - 1
                  GOTO 10
               END IF
            END IF

C           SET WRITESTYLE TO XOR TO ENABLE LINES TO BE
C           WRITTEN OVER

            CALL WRITESTYLE($GXOR)

C           PUT INITIAL COORDINATES IN IXPOS AND IYPOS

            IXPOS = XPOINT
            IYPOS = YPOINT

C           HIDE MOUSE CURSOR,DRAWLINE,SHOWMOUSECURSOR

             CALL HIDEMOUSECURSOR
             CALL DRAWLINE(XPOINT,YPOINT,
     1            IXPOS,IYPOS)
             CALL SHOWMOUSECURSOR

C         WHILE BUTTON IS PRESSED DOWN

          DO WHILE (BPOS.EQ.1)

C            STORE PREVIOUS MOUSE COORDS IN OLDXPOS&OLDYPOS

             OLDXPOS = IXPOS
             OLDYPOS = IYPOS

C            GET POSTION OF MOUSE

             CALL GETMOUSECURSORPOSITION(XPOS,YPOS,BPOS)

C            CONVERT POSITION INTO SCREEN COORDS

             IXPOS = CONVERTX(INT2(XPOS))
             IYPOS = CONVERTY(INT2(YPOS))

C            CHECK IF MOUSE HAS MOVED AND IF IT HAS
C            THEN DRAW OVER OLD LINE AND DRAW NEW ONE

             IF (((IXPOS - OLDXPOS) .NE.0).OR.
     1            ((IYPOS - OLDYPOS).NE.0)) THEN
                  CALL HIDEMOUSECURSOR
                  CALL DRAWLINE(XPOINT,YPOINT,
     1                          OLDXPOS,OLDYPOS)
                  CALL DRAWLINE(XPOINT,YPOINT,
     1                          IXPOS,IYPOS)
                  CALL SHOWMOUSECURSOR
             END IF
          END DO

          END IF

C         CHECK FOR MOUSE BEING RELEASED

          CALL GETMOUSEBUTTONRELEASEINFO(0,XPOS,YPOS,BPOS,BCOUNT)

C         IF IT HAS BEEN RELEASED

            IF (BCOUNT.EQ.1) THEN
               CHANGELINKNO = .TRUE.

C              CHECK IF MOUSE HAS BEEN CALLED APART FROM
C              IN THIS SUBROUTINE

               IF (CALLMOUSETRUE) THEN
                  GOTO 10
               END IF

C              SAME CHECKS AS FOR WHEN BUTTON WAS PRESSED
```

```
C               DOWN

                IF (.NOT.CHOOSELINK1) THEN
                   GOTO 10
                END IF
                IF ((CONNO.EQ.LINKNO).AND.(COUNTER.EQ.0)) THEN
                   GOTO 10
                END IF
               IF ((PIPE).AND.(LINKS)) THEN
                    IF ((NL1(LINKNO).EQ.0).OR.(NL1(LINKNO).EQ.3)) THEN
                        GOTO 10
                    END IF
               END IF
               IF ((PIPE).AND.(.NOT.LINKS)) THEN
                   IF ((NL1(LINKNO).EQ.1).OR.(NL1(LINKNO).EQ.2)) THEN
                      GOTO 10
                   END IF
               END IF

C              DRAW OVER PREVIOUS LINE

               CALL HIDEMOUSECURSOR
               CALL DRAWLINE(XPOINT,YPOINT,IXPOS,IYPOS)
               CALL SHOWMOUSECURSOR

C              FIND NEW POSITION OF MOUSE

               NIXPOS = CONVERTX(INT2(XPOS))
               NIYPOS = CONVERTY(INT2(YPOS))

C              CHECK TO SEE IF SECOND PROCESSOR WAS PICKED

               CALL FINDPROCNO2(NP2,NL2,NIXPOS,NIYPOS)

C              CHECK IF PROCESSOR WAS PICKED BUT NOT ON
C              A LINK

               IF (.NOT.CHOOSELINK2) THEN
                  IF (COUNTER.NE.0) THEN
                     CHANGELINKNO = .FALSE.
                  ELSE
                     LINKNO = LINKNO - 1
                     CHANGELINKNO = .TRUE.
                    END IF
                    GOTO 10
                  END IF

C              CHECK IF SAME PROCESSOR AND LINK NO
C              USED BEFORE FOR FIRST PROCESSOR PICKED

               IF (CHOOSELINK2) THEN

C                 CHECK IF SAME PROCESSOR AND LINK NO
C                 USED BEFORE

                  IF (NP1(LINKNO).NE.NP2(LINKNO)) THEN
                     DO 200 X = 1,LINKNO - 1
                        IF (((NP1(LINKNO).EQ.NP1(X)).AND.
     1                       (NL1(LINKNO).EQ.NL1(X))).OR.
     1                       ((NP1(LINKNO).EQ.NP2(X)).AND.
     1                       (NL1(LINKNO).EQ.NL2(X)))) THEN
                           IF (.NOT.CONNECT) THEN
                              CONNECT = .TRUE.
                           END IF
                           CHANGELINKNO = .TRUE.
                           LINKNO = LINKNO - 1
                           GOTO 10
                        END IF
200                  CONTINUE
                  END IF

C                 CHECK FOR CONNECTION BEING MADE

90                IF (CONNECT) THEN

C                 IF PIPELINE CHECK LINKS FOR PIELINE NOT
C                 USED BEFORE

                     IF ((PIPE).AND.(LINKS)) THEN
                      IF ((NL2(LINKNO).EQ.0).OR.(NL2(LINKNO).EQ.3)) THEN
                         IF (COUNTER.NE.0) THEN
                            CHANGELINKNO = .FALSE.
                         ELSE
                            LINKNO = LINKNO - 1
                            CHANGELINKNO = .TRUE.
                         END IF
                         GOTO 10
                       END IF
```

```
                END IF
               IF ((PIPE).AND.(.NOT.LINKS)) THEN
                  IF ((NL2(LINKNO).EQ.1).OR.(NL2(LINKNO).EQ.2)) THEN
                    IF (COUNTER.NE.0) THEN
                       CHANGELINKNO = .FALSE.
                    ELSE
                       LINKNO = LINKNO - 1
                       CHANGELINKNO = .TRUE.
                    END IF
                    GOTO 10
                  END IF
               END IF

C              CHECK IF TWO LINK NUMBERS THE SAME

               IF ((NL1(LINKNO).EQ.NL2(LINKNO)).AND.(PIPE).AND.
     1              (NP1(LINKNO).NE.NP2(LINKNO))) THEN
                    IF (COUNTER.NE.0) THEN
                       CHANGELINKNO = .FALSE.
                    ELSE
                       LINKNO = LINKNO - 1
                       CHANGELINKNO = .TRUE.
                    END IF
                    GOTO 10
                 END IF

C              CHECK IF LINK AND PROCESSOR USED BEFORE FOR
C              SECOND PROCESSOR AND LINK SELECTED

               IF (NP1(LINKNO).NE.NP2(LINKNO).OR.
     1            (NL1(LINKNO).NE.NL2(LINKNO).AND.NP1(LINKNO).EQ.
     1             NP2(LINKNO))) THEN
                  DO 300 X = 1,LINKNO - 1
                     IF (((NP2(LINKNO).EQ.NP1(X)).AND.
     1                    (NL2(LINKNO).EQ.NL1(X))).OR.
     1                    ((NP2(LINKNO).EQ.NP2(X)).AND.
     1                    (NL2(LINKNO).EQ.NL2(X)))) THEN
                       IF (COUNTER.EQ.0) THEN
                          CHANGELINKNO = .TRUE.
                          LINKNO = LINKNO - 1
                          GOTO 10
                       ELSE
                          CHANGELINKNO = .FALSE.
                          GOTO 10
                       END IF
                     END IF
300                CONTINUE
                END IF

                END IF



               END IF

C              DRAW NEW LINE

               CALL HIDEMOUSECURSOR
               CALL DRAWLINE(XPOINT,YPOINT,NIXPOS,NIYPOS)
               CALL SHOWMOUSECURSOR

C              INCREMENT COUNTER FOR COUNTING NO OF LINES

               COUNTER = COUNTER + 1

C              STORE NO OF LINES AND POINTS THAT MAKE UP
C              CONNECTIONS IN ARRAY(POINTS)

               POINTS(LINKNO,1) = COUNTER
               POINTS(LINKNO,COLUMN) = XPOINT
               COLUMN = COLUMN + 1
               POINTS(LINKNO,COLUMN) = YPOINT
               COLUMN = COLUMN + 1
               POINTS(LINKNO,COLUMN) = NIXPOS
               COLUMN = COLUMN + 1
               POINTS(LINKNO,COLUMN) = NIYPOS
               COLUMN = COLUMN + 1

C              IF CONNECTION HAS BEEN MADE CHECK IF DOUBLE CLICK
C              WAS MADE TO DELETE A LINE AND ALSO RESET COUNTER
C              AND COLUMN

               IF (CONNECT) THEN
                  COLUMN = 2
                  COUNTER = 0
                  CALL CHECKERASE(COUNTER,COLUMN,NIXPOS,NIYPOS,XPOINT,
     1                            YPOINT)
               END IF
```

```
        END IF
      GOTO 10
      END

C     ****OUTPUT TEXT TO SCREEN IN GRAPHICS MODE****

      SUBROUTINE TEXT(XCOORD,YCOORD,STRING)

      INCLUDE 'GRAPH.INC'
                  INCLUDE 'NEWXY.INC'
      INCLUDE 'FGRAPH.FD'

      RECORD/XYCOORD/XY
      INTEGER XCOORD,YCOORD
      CHARACTER*(*) STRING

      CALL MOVETO(NEWX(XCOORD),NEWY(YCOORD),XY)
      CALL OUTGTEXT(STRING)

      END

C     ****SET COLOUR****

      SUBROUTINE COLOUR(C)

      INCLUDE 'GRAPH.INC'
                  INCLUDE 'NEWXY.INC'
      INCLUDE 'FGRAPH.FD'

      INTEGER C
      INTEGER*2 PICK

      PICK = SETCOLOR(C)

      IF (PICK.EQ.-1) THEN
         STOP 'ERROR:CANNOT SET COLOUR'
      END IF

      END

C     ****SET THE WRITEMODE(I.E XOR.AND,ETC)****

      SUBROUTINE WRITESTYLE(STRING)

      INCLUDE 'GRAPH.INC'
                  INCLUDE 'NEWXY.INC'
      INCLUDE 'FGRAPH.FD'

      INTEGER*2 STRING,STYLE
      STYLE = SETWRITEMODE(STRING)

      IF (STYLE.EQ.-1) THEN
         STOP 'ERROR:CANNOT SET WRITEMODE'
      END IF

      END

C     ****CREATE BOX TO ENABLE YOU TO FINISH****

      SUBROUTINE ENDBOX(XCENT,YCENT)

      INCLUDE 'GRAPH.INC'
                  INCLUDE 'NEWXY.INC'
      INCLUDE 'FGRAPH.FD'

      INTEGER*2 STATUS
      INTEGER XCENT,YCENT

        STATUS = RECTANGLE($GBORDER,NEWX(XCENT - 48),NEWY(YCENT - 25),
     1                NEWX(XCENT + 60),NEWY(YCENT + 17))

        CALL COLOUR(13)
        CALL TEXT(XCENT - 45,YCENT - 15,'FINISH')

        END

C     ****FIND NO OF PROCESSOR AND LINKNO****

      SUBROUTINE FINDPROCNO1(ARRAY1,ARRAY2,IX,IY)

      INCLUDE 'GRAPH.INC'
      INTEGER M,X,N,Y,IXPOS,IYPOS
      INTEGER*2 ARRAY1(32),ARRAY2(32)

      PROCNO = 0
      Y = 150
```

```
C       MOVE DOWN ROW AT A TIME

        DO 20 M = 1,4
            X = 50

C        MOVE ACROSS COLUMN AT A TIME
         DO 10 N = 1,8

C        CHECK IF CLICKED IN A PROCESSOR ON A LINK THEN
C        STORE PROCESSOR AND LINK NUMBERS IN APPROPRIATE
C        ARRAYS(NP1,NP2,NL1,NL2)

           IF ((IY.GT.Y).AND.(IY.LT.Y + 100).AND.
     1        (IX.GT.X).AND.(IX.LT.X + 100)) THEN

C               INCREMENT  CONNECTION NUMBER BY ONE
                LINKNO = LINKNO + 1
                CHOOSELINK1 = .TRUE.
                IF ((IY.GT.Y).AND.(IY.LT.Y + 25).AND.
     1              (IX.GT.X + 35).AND.(IX.LT.X + 65)) THEN
                    IF (LINKS) THEN
                          ARRAY2(LINKNO) = 2
                    ELSE
                          ARRAY2(LINKNO) = 0
                    END IF
                    ARRAY1(LINKNO) = PROCNO
                    IX = X + 50
                    IY = Y
                ELSE IF ((IY.GT.Y + 35).AND.(IY.LT.Y + 65).AND.
     1              (IX.GT.X).AND.(IX.LT.X + 25)) THEN
                    IF (.NOT.LINKS) THEN
                       IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                          ARRAY2(LINKNO) = 2
                       ELSE
                          ARRAY2(LINKNO) = 1
                       END IF
                       ARRAY1(LINKNO) = PROCNO
                    ELSE
                       IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                          ARRAY2(LINKNO) = 3
                       ELSE
                          ARRAY2(LINKNO) = 0
                       END IF
                       ARRAY1(LINKNO) = PROCNO
                    END IF

                    IX = X
                    IY = Y + 50
                ELSE IF ((IY.GT.Y + 75).AND.(IY.LT.Y + 100).AND.
     1              (IX.GT.X + 35).AND.(IX.LT.X + 65)) THEN
                    IF (LINKS) THEN
                          ARRAY2(LINKNO) = 1
                    ELSE
                          ARRAY2(LINKNO) = 3
                    END IF
                    ARRAY1(LINKNO) = PROCNO
                    IX = X + 50
                    IY = Y + 100
                ELSE IF ((IY.GT.Y + 35).AND.(IY.LT.Y + 65).AND.
     1              (IX.GT.X + 75).AND.(IX.LT.X + 100)) THEN
                    IF (.NOT.LINKS) THEN
                       IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                          ARRAY2(LINKNO) = 1
                       ELSE
                          ARRAY2(LINKNO) = 2
                       END IF
                       ARRAY1(LINKNO) = PROCNO
                    ELSE
                       IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                          ARRAY2(LINKNO) = 0
                       ELSE
                          ARRAY2(LINKNO) = 3
                       END IF
                       ARRAY1(LINKNO) = PROCNO
                    END IF
                    IX = X + 100
                    IY = Y + 50
                ELSE

C                   IF LINK NOT CHOSEN THEN SET FLAG

                    LINKNO = LINKNO - 1
                    CHOOSELINK1 = .FALSE.
                    RETURN
                END IF
```

```
            RETURN
          ELSE
              CHOOSELINK1 = .TRUE.
          END IF

C         FOR SECOND AND FORTH ROWS HAVE PROCESSOR NUMBERS
C         RUNNING FROM LEFT TO RIGHT

          IF (((M.EQ.2).OR.(M.EQ.4)).AND.(N.LT.8)) THEN
             PROCNO = PROCNO - 1
          END IF

C         FOR FIRST AND THIRD ROWS HAVE PROCESSOR NUMBERS
C         RUNNING FROM RIGHT TO LEFT

          IF (((M.EQ.1).OR.(M.EQ.3)).AND.(N.LT.8)) THEN
             PROCNO = PROCNO + 1
          END IF
            X = X + 150
10        CONTINUE
          PROCNO = PROCNO + 8
          Y = Y + 200
20        CONTINUE

          END

C         ****FIND NO OF SECOND PROCESSOR AND LINK****

          SUBROUTINE FINDPROCNO2(ARRAY1,ARRAY2,IX,IY)

          INCLUDE 'GRAPH.INC'
          INTEGER M,X,N,Y,IXPOS,IYPOS
          INTEGER*2 ARRAY1(32),ARRAY2(32)

          PROCNO = 0
          Y = 150

C         LOOK AT ROWS OF PROCESSORS

          DO 20 M = 1,4
              X = 50

C         LOOK AT COLUMNS OF  PROCESSORS

             DO 10 N = 1,8

C            CHECK WHETHER MOUSE CLICKED WITHIN
C            PROCESSOR AND THEN WITHIN LINK

               IF ((IY.GT.Y).AND.(IY.LT.Y + 100).AND.
     1            (IX.GT.X).AND.(IX.LT.X + 100)) THEN
                CHOOSELINK2 = .TRUE.
                IF ((IY.GT.Y).AND.(IY.LT.Y + 25).AND.
     1              (IX.GT.X + 35).AND.(IX.LT.X + 65)) THEN
                    IF (LINKS) THEN
                          ARRAY2(LINKNO) = 2
                    ELSE
                          ARRAY2(LINKNO) = 0
                    END IF
                    ARRAY1(LINKNO) = PROCNO
                    IX = X + 50
                    IY = Y
                ELSE IF ((IY.GT.Y + 35).AND.(IY.LT.Y + 65).AND.
     1              (IX.GT.X).AND.(IX.LT.X + 25)) THEN
                    IF (.NOT.LINKS) THEN
                       IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                          ARRAY2(LINKNO) = 2
                       ELSE
                          ARRAY2(LINKNO) = 1
                       END IF
                       ARRAY1(LINKNO) = PROCNO
                    ELSE
                       IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                          ARRAY2(LINKNO) = 3
                       ELSE
                          ARRAY2(LINKNO) = 0
                       END IF
                       ARRAY1(LINKNO) = PROCNO
                    END IF
                    IX = X
                    IY = Y + 50
                ELSE IF ((IY.GT.Y + 75).AND.(IY.LT.Y + 100).AND.
     1              (IX.GT.X + 35).AND.(IX.LT.X + 65)) THEN
                    IF (LINKS) THEN
                          ARRAY2(LINKNO) = 1
                    ELSE
                          ARRAY2(LINKNO) = 3
                    END IF
```

```
            ARRAY1(LINKNO) = PROCNO
            IX = X + 50
            IY = Y + 100
         ELSE IF ((IY.GT.Y + 35).AND.(IY.LT.Y + 65).AND.
1           (IX.GT.X + 75).AND.(IX.LT.X + 100)) THEN
            IF (.NOT.LINKS) THEN
               IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                  ARRAY2(LINKNO) = 1
               ELSE
                  ARRAY2(LINKNO) = 2
               END IF
               ARRAY1(LINKNO) = PROCNO
            ELSE
               IF ((M.EQ.1).OR.(M.EQ.3)) THEN
                  ARRAY2(LINKNO) = 0
               ELSE
                  ARRAY2(LINKNO) = 3
               END IF
                  ARRAY1(LINKNO) = PROCNO
               IX = X + 100
               IY = Y + 50
            ELSE

C              SET FLAGS IF CLICKED ON PROCESSOR BUT
C              NOT ON LINK
                  CONNECT = .TRUE.
                  CHOOSELINK2 = .FALSE.
                  RETURN
               END IF

            CONNECT = .TRUE.
            RETURN
         ELSE
             CHOOSELINK2 = .TRUE.
             CONNECT = .FALSE.
         END IF

C        FOR SECOND AND THIRD ROWS HAVE PROCESSOR NUMBERS
C        RUNNING FROM RIGHT TO LEFT

         IF (((M.EQ.2).OR.(M.EQ.4)).AND.(N.LT.8)) THEN
            PROCNO = PROCNO - 1
            END IF

C           FOR SECOND AND THIRD ROWS HAVE PROCESSOR NUMBERS
C           RUNNING FROM LEFT TO RIGHT

            IF (((M.EQ.1).OR.(M.EQ.3)).AND.(N.LT.8)) THEN
               PROCNO = PROCNO + 1
            END IF
            X = X + 150
10       CONTINUE
         PROCNO = PROCNO + 8
         Y = Y + 200
20       CONTINUE

         END

C        ****DRAW BOXES WITH LINK NUMBERS IN THEM****

         SUBROUTINE LINKBOX(XCENT,YCENT,PROCNO)

                     INTEGER XCENT,YCENT,PROCNO
         CHARACTER*1 L,R,T,P

         INCLUDE 'GRAPH.INC'
                     INCLUDE 'NEWXY.INC'
         INCLUDE 'FGRAPH.FD'

         CALL COLOUR(9)

C        RIGHT HAND BOX

         CALL DRAWLINE(XCENT + 50,YCENT - 15,XCENT + 25,YCENT - 15)
         CALL DRAWLINE(XCENT + 25,YCENT - 15,XCENT + 25,YCENT + 15)
         CALL DRAWLINE(XCENT + 25,YCENT + 15,XCENT + 50,YCENT + 15)

C        TOP BOX

         CALL DRAWLINE(XCENT - 15,YCENT - 50,XCENT - 15,YCENT - 25)
         CALL DRAWLINE(XCENT - 15,YCENT - 25,XCENT + 15,YCENT - 25)
         CALL DRAWLINE(XCENT + 15,YCENT - 25,XCENT + 15,YCENT - 50)

C        BOTTOM BOX

         CALL DRAWLINE(XCENT + 15,YCENT + 50,XCENT + 15,YCENT + 25)
         CALL DRAWLINE(XCENT + 15,YCENT + 25,XCENT - 15,YCENT + 25)
```

```
      CALL DRAWLINE(XCENT - 15,YCENT + 25,XCENT - 15,YCENT + 50)

C     LEFT BOX

      CALL DRAWLINE(XCENT - 50,YCENT - 15,XCENT - 25,YCENT - 15)
      CALL DRAWLINE(XCENT - 25,YCENT - 15,XCENT - 25,YCENT + 15)
      CALL DRAWLINE(XCENT - 25,YCENT + 15,XCENT - 50,YCENT + 15)

C     CORNER BOXES

      CALL DRAWLINE(XCENT + 25,YCENT - 15,XCENT + 15,YCENT - 25)
      CALL DRAWLINE(XCENT - 15,YCENT - 25,XCENT - 25,YCENT - 15)
      CALL DRAWLINE(XCENT - 25,YCENT + 15,XCENT - 15,YCENT + 25)
      CALL DRAWLINE(XCENT + 15,YCENT + 25,XCENT + 25,YCENT + 15)

      CALL COLOUR(10)

C     PUT LINK NUMBERS IN BOXES

      IF (.NOT.LINKS) THEN
         IF ((PROCNO.EQ.1).OR.(PROCNO.EQ.3)) THEN
           CALL TEXT(XCENT - 46,YCENT - 11,'2')
           CALL TEXT(XCENT - 8,YCENT + 27,'3')
           CALL TEXT(XCENT + 30,YCENT - 11,'1')
           CALL TEXT(XCENT - 8,YCENT - 48,'0')
         ELSE
           CALL TEXT(XCENT - 46,YCENT - 11,'1')
           CALL TEXT(XCENT - 8,YCENT + 27,'3')
           CALL TEXT(XCENT + 30,YCENT - 11,'2')
           CALL TEXT(XCENT - 8,YCENT - 48,'0')
          END IF
       ELSE
         IF ((PROCNO.EQ.1).OR.(PROCNO.EQ.3)) THEN
            CALL TEXT(XCENT - 46,YCENT - 11,'3')
            CALL TEXT(XCENT - 8,YCENT + 27,'1')
            CALL TEXT(XCENT + 30,YCENT - 11,'0')
            CALL TEXT(XCENT - 8,YCENT - 48,'2')
         ELSE
            CALL TEXT(XCENT - 46,YCENT - 11,'0')
            CALL TEXT(XCENT - 8,YCENT + 27,'1')
            CALL TEXT(XCENT + 30,YCENT - 11,'3')
            CALL TEXT(XCENT - 8,YCENT - 48,'2')
         END IF
         END IF
         CALL COLOUR(2)

C     PUT NUMBERS IN SIDE BOXES

      CALL TEXT(XCENT - 40,YCENT - 40,'4')
      CALL TEXT(XCENT - 40,YCENT + 25,'7')
      CALL TEXT(XCENT + 25,YCENT - 40,'5')
      CALL TEXT(XCENT + 25,YCENT + 25,'6')
      END

C      ****DRAW PIPELINE****

      SUBROUTINE PIPELINE

      INCLUDE 'FGRAPH.FD'

      INTEGER X,Y,N,M

      CALL COLOUR(14)
      Y = 200

      DO 20 M = 1,4
         X = 0
      DO 10 N = 1,9
         IF ((N.EQ.1).AND.((M.EQ.2).OR.(M.EQ.3))) THEN
            CALL DRAWLINE(X + 10,Y,X + 50,Y)
         ELSE
          CALL DRAWLINE(X,Y,X + 50,Y)
         END IF
       IF ((N.EQ.9).AND.(M.EQ.1)) THEN
        CALL DRAWLINE(X + 50,Y,X + 50,Y + 200)
       END IF
       IF ((N.EQ.1).AND.(M.EQ.2)) THEN
         CALL DRAWLINE (X + 10,Y,X + 10,Y + 200)
       END IF
       IF ((N.EQ.9).AND.(M.EQ.3)) THEN
         CALL DRAWLINE (X + 50,Y,X + 50,Y + 200)
       END IF
       IF ((N.EQ.1).AND.(M.EQ.4)) THEN
         CALL DRAWLINE (X,Y,X,Y - 600)
       END IF
       X = X + 150
10     CONTINUE
       Y = Y + 200
```

```
20        CONTINUE

          END

C         DRAW ALL 32 PROCESSORS

          SUBROUTINE DRAWPROCESSOR()

                INTEGER N,X,Y,PROCNO,M,LINENO
          INCLUDE 'GRAPH.INC'

C         DRAW PROCESSORS ROW AT A TIME WITH FIRST
C         AND THIRD ROW NUMBERS FROM LEFT TO RIGHT
C         AND VICE-VERSA FOR RIGHT TO LEFT

          PROCNO = 0
          Y = 200
          DO 20 M = 1,4
            X = 100
           DO 10 N = 1,8
            LINENO = M
            CALL BOX(X,Y,PROCNO)
            CALL LINKBOX(X,Y,LINENO)
            IF (((M.EQ.2).OR.(M.EQ.4)).AND.(N.LT.8)) THEN
               PROCNO = PROCNO - 1
            END IF
            IF (((M.EQ.1).OR.(M.EQ.3)).AND.(N.LT.8)) THEN
               PROCNO = PROCNO + 1
            END IF
            X = X + 150
10         CONTINUE
           PROCNO = PROCNO + 8
          Y = Y + 200
20        CONTINUE

          END

C         **** ERASE CONNECTION****

          SUBROUTINE ERASE(CONNO)

          INCLUDE 'GRAPH.INC'
          INCLUDE 'FGRAPH.FD'
          INTEGER NOPOINTS,N,CONNO,J

C         FIND OUT HOW MANY LINES USED TO MAKE CONNECTION
C         FROM POINTS(LINKNO,1) AND THEN DELETE ONE AT A
C         TIME

          NOPOINTS = POINTS(CONNO,1)
          J = 2
          CALL HIDEMOUSECURSOR
          DO 10 N = 1,NOPOINTS
             CALL DRAWLINE(POINTS(CONNO,J),POINTS(CONNO,J + 1),
     1                     POINTS(CONNO,J + 2),POINTS(CONNO,J + 3))
             J = J + 4
10        CONTINUE
          CALL SHOWMOUSECURSOR

          END

C         ****CHECKS FOR CLICK ON PROCESSOR PREVIOUSLY USED
C             BEFORE****

          SUBROUTINE CHECKERASE(COUNTER,COLUMN,NIXPOS,NIYPOS,XPOINT,
     1                          YPOINT)

          INCLUDE 'GRAPH.INC'
          INTEGER K,P1,P2,L1,L2,IREM,IC,J,COUNTER,COLUMN,NIXPOS,
     1            NIYPOS,XPOINT,YPOINT
          LOGICAL DELETION

          DELETION = .FALSE.
          P1 = NP1(LINKNO)
          P2 = NP2(LINKNO)
          L1 = NL1(LINKNO)
          L2 = NL2(LINKNO)

C         CHECK MOUSE CLICKED AND RELEASED ON THE SAME PROCESSOR
C         THEN CHECK IF ONE LINE DRAWN TO START MAKING CONNECTION

          IF ((P1.EQ.P2).AND.(L1.EQ.L2)) THEN
           IF (POINTS(LINKNO,1).EQ.2) THEN
             CALL ERASEBOX(350,31)

C              IF YOU DO WANT ERASE BOX THEN LINKNO DECREASED
C              BY ONE. IF DON'T THEN REDRAWLINE.
```

```
      IF (TEST) THEN
         CHANGELINKNO = .TRUE.
         LINKNO = LINKNO - 1
         COUNTER = 0
         COLUMN = 2
         RETURN
      ELSE
         CALL HIDEMOUSECURSOR
         CALL DRAWLINE(XPOINT,YPOINT,NIXPOS,NIYPOS)
         CALL SHOWMOUSECURSOR
         POINTS(LINKNO,2) = NIXPOS
         POINTS(LINKNO,3) = NIYPOS
         POINTS(LINKNO,4) = XPOINT
         POINTS(LINKNO,5) = YPOINT
         COUNTER = 1
         COLUMN = 6
         CONNECT = .FALSE.
         NIXPOS = XPOINT
         NIYPOS = YPOINT
         CALLMOUSETRUE = .TRUE.
        END IF
      ELSE

C       CHECK IF PROCESSOR NUMBER AND LINK USED BEFORE THEN
C       CHECK IF CONNECTION IS REALLY MEANT TO DELETED
C       AND IF IT HAS TO BE DELETED ADJUST THE ARRAYS CONTAINING
C       LINK AND PROCESSOR NUMBERS ACCORDINGLY.

        DO 10 K = 1,LINKNO - 1
           IF (((P1.EQ.NP1(K)).AND.(L1.EQ.NL1(K))).OR.
     1          ((P1.EQ.NP2(K)).AND.(L1.EQ.NL2(K)))) THEN
              CALL ERASEBOX(350,31)
              IF (.NOT.TEST) THEN
                 LINKNO = LINKNO - 1
                 RETURN
              END IF
              DELETION = .TRUE.
              CALL ERASE(LINKNO)
              CALL ERASE(K)
              IREM = K
              IC = 0
              DO 20 J = 1,LINKNO - 1
                 IF (J.NE.IREM) THEN
                    IC = IC + 1
                    NP1(IC) = NP1(J)
                    NP2(IC) = NP2(J)
                    NL1(IC) = NL1(J)
                    NL2(IC) = NL2(J)
                    DO 30 N = 1,20
                       POINTS(IC,N) = POINTS(J,N)
30                  CONTINUE
                 END IF
20            CONTINUE
           END IF
10      CONTINUE
        END IF

C       CHECK IF CLICKED ON PROCESSOR AND LINK NOT USED
C       BEFORE AND IF YOU HAVE USED SEVERAL LINES TO MAKE
C       A CONNECTION THEN DELETE AUTOMATICALLY

        IF ((.NOT.DELETION.AND.POINTS(LINKNO,1).EQ.1)
     1       .OR.(POINTS(LINKNO,1).GT.2)) THEN
           CALL ERASE(LINKNO)
           CHANGELINKNO = .TRUE.
           LINKNO = LINKNO - 1
        END IF

C       IF A DELETION HAS BEEN MADE THEN SUBTRACT TWO
C       FROM THE LINK NUMBER AS POINTS FROM CLICKING
C       AND RELEASING ON SAME PROCESSOR ARE STORED

        IF (DELETION) THEN
           CHANGELINKNO = .TRUE.
           LINKNO = LINKNO - 2
        END IF
        END IF

        END

C       ****PRINT BOX TO ASK WHETHER YOU DO REALLY WANT TO DELETE A
C           CONNECTION AND DELETE THE BOX WHEN SELECTION
C           HAS BEEN MADE****

        SUBROUTINE ERASEBOX(XCENT,YCENT)

        INCLUDE 'MOUSE.FD'
        INCLUDE 'FGRAPH.FD'
```

```
      INCLUDE 'GRAPH.INC'
      INCLUDE 'NEWXY.INC'

      INTEGER*2 STATUS,BPOS,XPOS,YPOS
      INTEGER XCENT,YCENT,IXPOS,IYPOS

C     PRINT BOXES FOR QUESTION AND ANSWER(Y/N?)

      CALL COLOUR(12)
      STATUS = RECTANGLE($GBORDER,NEWX(XCENT - 200),NEWY(YCENT - 21),
     1             NEWX(XCENT + 336),NEWY(YCENT + 21))

      STATUS = RECTANGLE($GBORDER,NEWX(XCENT + 370),NEWY(YCENT - 21),
     1             NEWX(XCENT + 400),NEWY(YCENT + 21))

      STATUS = RECTANGLE($GBORDER,NEWX(XCENT + 401),NEWY(YCENT - 21),
     1             NEWX(XCENT + 431),NEWY(YCENT + 21))

      CALL COLOUR(14)
      CALL TEXT(XCENT - 190,YCENT - 12,'ARE YOU SURE YOU WANT TO')
      CALL TEXT(XCENT + 220,YCENT - 12,'DELETE?')
      CALL TEXT(XCENT + 377,YCENT - 12,'Y')
      CALL TEXT(XCENT + 408,YCENT - 12,'N')


10    CALL GETMOUSECURSORPOSITION(XPOS,YPOS,BPOS)
      CALLMOUSETRUE = .TRUE.
      IXPOS = CONVERTX(INT2(XPOS))
      IYPOS = CONVERTY(INT2(YPOS))

C     CHECK WHETHER CLICKED IN Y OR N BOX

      IF (BPOS.EQ.1) THEN
         IF ((IXPOS.GT.(XCENT + 370)).AND.(IXPOS.LT.(XCENT + 400))
     1        .AND.(IYPOS.GT.YCENT - 21).AND.(IYPOS.LT.YCENT + 21)) THEN
             TEST = .TRUE.
         ELSE IF ((IXPOS.GT.(XCENT + 401)).AND.(IXPOS.LT.(XCENT + 431))
     1        .AND.(IYPOS.GT.YCENT - 21).AND.(IYPOS.LT.YCENT + 21)) THEN
             TEST = .FALSE.
         ELSE
            GOTO 10
         END IF
      ELSE
         GOTO 10
      END IF
          CALL COLOUR(12)
          CALL HIDEMOUSECURSOR()

C         DRAW OVER RECTANGLES AND WORDS

          STATUS = RECTANGLE($GBORDER,NEWX(XCENT + 401),
     1                NEWY(YCENT - 21),NEWX(XCENT + 431),NEWY(YCENT + 21))

          CALL COLOUR($BLACK)
          CALL TEXT(XCENT + 408,YCENT - 12,'N')
          CALL COLOUR(12)
          STATUS = RECTANGLE($GBORDER,NEWX(XCENT + 370),
     1                NEWY(YCENT - 21),NEWX(XCENT + 400),NEWY(YCENT + 21))
          CALL COLOUR($BLACK)
          CALL TEXT(XCENT + 377,YCENT - 12,'Y')
          CALL COLOUR(12)
          STATUS = RECTANGLE($GBORDER,NEWX(XCENT - 200),
     1                NEWY(YCENT - 21),NEWX(XCENT + 336),NEWY(YCENT + 21))

          CALL COLOUR($BLACK)
          CALL TEXT(XCENT - 190,YCENT - 12,'ARE YOU SURE YOU WANT TO')
          CALL TEXT(XCENT + 220,YCENT - 12,'DELETE?')
          CALL SHOWMOUSECURSOR()
          CALL COLOUR(11)

          END

C     ****SET UP AN ARRAY CONTAINING CONNECTIONS MADE
C     TO COO4****

      SUBROUTINE COO4CONNECTION()

      INCLUDE 'GRAPH.INC'

      INTEGER I

C     SET UP ARRAYS CONTAINING CONNECTIONS
C     ON COO4-1 AND COO4-0

      DO 10 I = 1,32
          COO41(I,1) = 1
          COO41(I,2) = I - 1
          COO41(I,3) = I - 1
```

```
            C0041(I,4) = 2
            C0041(I,5) = I - 1
            C0041(I,6) = I - 1
            C0040(I,1) = 2
            C0040(I,2) = I - 1
            C0040(I,3) = I - 1
            C0040(I,4) = 1
            C0040(I,5) = I - 1
            C0040(I,6) = I - 1

10          CONTINUE
            END

C       ****FIND WHAT CONNECTIONS NEED TO BE MADE ON THE C004****

        SUBROUTINE CONNECTIONS(PNO,LNO)

        INCLUDE 'GRAPH.INC'

        INTEGER * 2 PNO(32),LNO(32)
        INTEGER I,J

C       FIND FROM LINK AND PROCESSOR NUMBERS USED WHAT THE
C       CORRESPONDING PIN ON THE C004 IS

          DO 10 I = 1,LINKNO
            DO 20 J = 1,32
              IF((LNO(I).EQ.C0041(J,1)).AND.(PNO(I).EQ.C0041(J,2)))
     1          THEN
                C41IN(I) = C0041(J,3)
              END IF
              IF ((LNO(I).EQ.C0041(J,4)).AND.(PNO(I).EQ.C0041(J,5)))
     1          THEN
                C41OUT(I) = C0041(J,6)
              END IF
              IF((LNO(I).EQ.C0040(J,1)).AND.(PNO(I).EQ.C0040(J,2)))
     1           THEN
                C40IN(I) = C0040(J,3)
              END IF
              IF((LNO(I).EQ.C0040(J,4)).AND.(PNO(I).EQ.C0040(J,5)))
     1           THEN
                C40OUT(I) = C0040(J,6)
              END IF
20          CONTINUE
10        CONTINUE
        END

C       ****OFFER OPTION TO INTERROGATE C004****

        SUBROUTINE INTERROGATE

        INCLUDE 'GRAPH.INC'

        CHARACTER*1 REPLY,IREPLY
        INTEGER C4,OUTPUT
        INTEGER*2 IN

10      WRITE(*,*) 'DO YOU WANT TO INTERROGATE C004S?(Y/N)'
        READ(*,'(A)') REPLY

        IF ((REPLY.EQ.'Y').OR.(REPLY.EQ.'y')) THEN
20         WRITE(*,*) 'WHICH C004 DO YOU WANT TO INTERROGATE?'
           READ(*,*) C4
           IF ((C4.NE.0).AND.(C4.NE.1)) THEN
              GOTO 20
           END IF
30         WRITE(*,*) 'WHICH OUTPUT TO YOU WANT TO INTERROGATE?'
           READ(*,*) OUTPUT
           IF (OUTPUT.GT.31) THEN
              WRITE(*,*) 'NUMBER IS TOO HIGH.'
              GOTO 30
           END IF

C          SEND A 2 THEN THE NUMBER OF THE C004 YOU
C          WANT TO INTERROGATE. TEST IF THE MSB OF THE BYTE
C          RETURNED IS SET INDICATING A CONNECTION AND IF
C          IT IS THEN SUBTRACT 128 FROM THE BYTE TO GET THE INPUT

           IF (C4.EQ.0) THEN
              CALL LINKOUT(2,LKAD(1))
              CALL LINKOUT(OUTPUT,LKAD(1))
              IN = LINKIN(LKAD(1))
              IF (((128 - IN).LT.0).OR.((128 - IN).EQ.0)) THEN
                 IN = IN - 128
                 WRITE(*,*) 'THIS OUTPUT IS CONNECTED TO INPUT',IN
              ELSE
                 WRITE(*,*) 'THIS OUTPUT IS NOT CONNECTED'
              END IF
```

```
      ELSE
         CALL LINKOUT(2,LKAD(2))
         CALL LINKOUT(OUTPUT,LKAD(2))
         IN = LINKIN(LKAD(2))
         IF (((128 - IN).LT.0).OR.((128 - IN).EQ.0)) THEN
            IN = IN - 128
            WRITE(*,*) 'THIS OUTPUT IS CONNECTED TO INPUT',IN
         ELSE
            WRITE(*,*) 'THIS OUTPUT IS NOT CONNECTED'
         END IF
      END IF


40    WRITE(*,*) 'DO YOU WANT TO INTERROGATE THE COO4S FURTHER?(Y/N)'
      READ(*,'(A)') IREPLY
      IF ((IREPLY.NE.'N').AND.(IREPLY.NE.'n').AND.
     1    (IREPLY.NE.'Y').AND.(IREPLY.NE.'y')) THEN
       GOTO 40
      END IF
      IF ((IREPLY.EQ.'Y').OR.(IREPLY.EQ.'y')) THEN
        GOTO 20
      END IF
      ELSE IF ((REPLY.EQ.'N').OR.(REPLY.EQ.'n')) THEN
           RETURN
      ELSE
          GOTO 10
      END IF

      END

C     ****DISPLAY ALL THE CONNECTIONS MADE ON THE COO4****

      SUBROUTINE DISPLAYCONNECTIONS()

      INCLUDE 'GRAPH.INC'

      CHARACTER*1 REPLY,IREPLY
      INTEGER M,P
      INTEGER * 2 IN0,IN1

C     DISPLAY CONNECTIONS MADE BETWEEN PROCESSORS

10    WRITE(*,5)
5     FORMAT(1X,'DO YOU WANT TO SEE ALL THE CONNECTIONS',1X,
     1       'BETWEEN THE PROCESSORS?(Y/N)')
      READ(*,'(A)') REPLY

C     FIND WHICH INPUT EACH OUTPUT IS CONNECTED TO
C     TEST MSB TO SEE IF SET

      IF ((REPLY.EQ.'Y').OR.(REPLY.EQ.'y')) THEN
40     DO 30 P = 1,32
         CALL LINKOUT(2,LKAD(1))
         CALL LINKOUT(COO40(P,6),LKAD(1))
         IN0 = LINKIN(LKAD(1))
         CALL LINKOUT(2,LKAD(2))
         CALL LINKOUT(IN0,LKAD(2))
         IN1 = LINKIN(LKAD(2))
         IF (((128 - IN0).LT.0).OR.((128 - IN0).EQ.0)) THEN
              IN0 = IN0 - 128
              IN1 = IN1 - 128
              WRITE(*,26) COO40(P,5),COO40(P,4),COO40(IN0 + 1,2),
     1                    COO40(IN0 + 1,1)
26            FORMAT(1X,'PROCESSOR',1X,I2,1X,'LINKIN',1X,I2,1X,
     1          'IS CONNECTED TO PROCESSOR',1X,I2,1X,
     1           'LINKOUT',1X,I2)
              WRITE(*,26) COO41(IN0 + 1,5),COO41(IN0 + 1,4),
     1                    COO41(IN1 + 1,2),COO41(IN1 + 1,1)
         END IF
30     CONTINUE
      ELSE IF ((REPLY.EQ.'N').OR.(REPLY.EQ.'n')) THEN
            RETURN
      ELSE
          GOTO 10
      END IF

      END

C     ENABLE TESTING OF COO4 USING FOUR LINK ADAPTERS

      SUBROUTINE TESTCOO4()

      INCLUDE 'GRAPH.INC'

      INTEGER OUTPUT1,OUTPUT2
      INTEGER * 2 RECEIVE1,RECEIVE2
      CHARACTER*1 REPLY
```

```
C       CODE FOR TESTING C004

10     WRITE(*,*) 'DO YOU WANT TO TEST THE C004''s?(Y/N)'
       READ(*,'(A1)') REPLY


       IF ((REPLY.EQ.'Y').OR.(REPLY.EQ.'y')) THEN
          WRITE(*,16)

C         OUTPUT NUMBER FROM ONE LINK ADAPTER AND RECEIVE
C         AT THE OTHER IF CONNECTIONS HAVE BEEN MADE PROPERLY
C         ON THE C004S


16        FORMAT(1X,'ENTER NUMBER YOU WANT TO SEND TO LINK',1X,
     1           'ADAPTOR 2')
          READ(*,*) OUTPUT1
          CALL LINKOUT(OUTPUT1,LKAD(3))
          WRITE(*,17)
17        FORMAT(1X,'ENTER NUMBER YOU WANT TO SEND TO LINK',1X,
     1         'ADAPTOR 3')
          READ(*,*) OUTPUT2
          CALL LINKOUT(OUTPUT2,LKAD(4))

          RECEIVE2 = LINKIN(LKAD(3))
          WRITE(*,18) RECEIVE2
18        FORMAT(1X,'THE NUMBER RECEIVED AT LINK ADAPTOR',1X,
     1       '2 WAS',I3)

          RECEIVE1 = LINKIN(LKAD(4))
          WRITE(*,19) RECEIVE1
19        FORMAT(1X,'THE NUMBER RECEIVED AT LINK ADAPTOR',1X,
     1         '3 WAS',I3)
          ELSE IF ((REPLY.EQ.'N').OR.(REPLY.EQ.'n')) THEN
             RETURN
          ELSE
             GOTO 10
          END IF

       END
```

Appendix B:

Source code for dynamic interconnection network.

```
{*************************************************************
* NAME: BOOT.dsp                                             *
* DESCRIPTION: ALLOWS ADSP-2105 TO BE BOOTED VIA A C012      *
*                                                            *
*                                                            *
*                                                            *
*************************************************************}

.MODULE/RAM/BOOT=0      parallel_boot_monitor;
.VAR/DM                 count;                      {counts bytes}
.VAR/DM                 ins_count;                  {counts instructions}
.INCLUDE                <E:\ADI_DSP\INCLUDE\DEF2105.h>;{control settings}

.PORT                   read_c012;
.PORT                   write_c012;
.PORT                   input_status;
.PORT                   output_status;
.PORT                   read_c004;
.PORT                   write_c004;
.PORT                   c004_input_status;
.PORT                   c004_output_status;
.GLOBAL                 code_start;

                        JUMP restarter;NOP;NOP;NOP;
                        RTI;NOP;NOP;NOP;
                        NOP;NOP;NOP;NOP;
                        NOP;NOP;NOP;NOP;
                        RTI;NOP;NOP;NOP;
                        RTI;NOP;NOP;NOP;
                        RTI;NOP;NOP;NOP;

restarter:              CALL initialisations;
data_ready:             AX0 = DM(input_status);
                        AY0 = 1;
                        AR = AX0 AND AY0;
                        IF EQ JUMP data_ready;

                        JUMP io_port;

initialisations:        I5=^code_start;             {pointer to start}
                        M5=1;                       {increment by 1}
                        L5=0;                       {length of code}
                        SR0=0;
                        SR1=0;                      {initialise results reg.}

                        AX1=1;
                        DM(count)=AX1;              {set no. of bytes=1}
                        AX0=0;
                        DM(Sys_Ctrl_Reg)=AX0;{disable sport1}
                        DM(Dm_Wait_Reg)=AX0;{no wait states}
                        DM(Tperiod_Reg)=AX0;{timer not used}
                        DM(Tcount_Reg)=AX0;
                        DM(Tscale_Reg)=AX0;

                        AX0=H#FFFF;
                        DM(ins_count)=AX0;          {set ins_count to -ve}
                        IMASK=0;                    {enable IRQ2 interrup}
                        AX0=0;
                        AY0=0;
                        RTS;

io_port:                AY1=DM(ins_count);
                        AR=PASS AY1;
                        IF GT JUMP next_instruction;
                        IF LT JUMP load_word_count;
                        IF EQ JUMP code_start;

load_word_count:        AY0=DM(count);
                        AR=PASS AY0;
                        IF NE JUMP first_byte;
                        IF EQ JUMP second_byte;

first_byte:             SI=DM(read_c012);
                        AR=AY0-1;
                        DM(count)=AR;
                        JUMP data_ready;

second_byte:            SR0=DM(read_c012);
                        SR=SR OR LSHIFT SI BY 8 (LO);
                        DM(ins_count) = SR0;
                        AX0=3;
                        DM(count)=AX0;
                        JUMP data_ready;
```

```
next_instruction:       AX0=2;(decide which byte is due)
                        AY0=DM(count);
                        AR=AX0-AY0;

                        IF LT JUMP most_sig_byte;
                        IF EQ JUMP middle_byte;
                        IF GT JUMP least_sig_byte;

most_sig_byte:          SI=DM(read_c012);           {load MS byte into SI}
                        AR=AY0 - 1;                 {decrement count}
                        DM(count) = AR;
                        JUMP data_ready;

middle_byte:            SR0=DM(read_c012);{load middle into SR}
                        SR=SR OR LSHIFT SI BY 8 (LO); {put MS and mid.}
                        AR=AY0-1;
                        DM(count)=AR;
                        JUMP data_ready;

least_sig_byte:         PX=DM(read_c012);{put LS byte into PX}
                        PM(I5,M5)=SR0;              {write SR0 into PM}
                                                    {PX provides 8 LS bits}
                        AX0=3;
                        DM(count)=AX0;              {reset byte count}
                        AR=AY1-1;                   {decrement ins count}
                        DM(ins_count)=AR;
                        JUMP data_ready;
code_start:             NOP
.ENDMOD;
```

;

```
(C*************************************************************
C NAME:FLASH.DSP                                                *
C                                                               *
C DESCRIPTION: DOWNLOADABLE PROGRAM THAT FLASHES LED            *
C                                                               *
C                                                               *
C                                                               *
C                                                               *
C*************************************************************)
.MODULE/ROM/SEG=int_pm/ABS=H#005Dflash_led;
.INCLUDE                        <E:\ADI_DSP\INCLUDE\DEF2105.h>;
.ENTRY                          flash;
.PORT                           read_c012;
.PORT                           input_status;

                                AX0=0;
                                DM(input_status) = AX0; (disable inputint)
                                AY0=DM(read_c012); (dummy read)
                                I6=H#0018; (address of timer int)
                                M6=0;
                                L6=0;
                                I7=^intinstr; (pointer to start of flash)
                                M7=0;
                                L7=0;
                                JUMP loadint;

intinstr:                       JUMP flash;
loadint:                        AX0 = PM(I7,M7);
                                PM(I6,M6) = AX0;
                                (AX0=H#0B00; loads jmp inst. at timer int
                                PX=H#C0;
                                PM(I6,M6)=AX0;)

                                AX0=H#FFFF; (sets timer period)
                                DM(Tperiod_Reg)=AX0;(Set counter)
                                DM(Tcount_Reg)=AX0;
                                AX0=H#1B;

                                DM(Tscale_Reg)=AX0;
                                IMASK=1;
                                ENA TIMER;
```

```
C*************************************************************
C NAME: DOWN1                                                 *
C DATE: 3/6/94                                                *
C DESCRIPTION: ROUTINE TO TAKE A PROM SPLITTER FILE           *
C              IN INTEL HEX FORMAT AND DOWNLOAD IT A BYTE     *
C              AT A TIME VIA C012                             *
C*************************************************************
      PROGRAMDOWN1
      INTEGER*1 RECORD,DATA(1000),BYTEOUT1,BYTEOUT2
      INTEGER*2 ADDR,NWORDS,NBYTES,START,
     1          TOTALNBYTES,RESETADDR,LINKADDR,
     1          BYTE1,BYTE2,TOPBYTE
       INTEGER*4 LOOPCOUNT
       CHARACTER*1 DELIM

       EQUIVALENCE (BYTE1,BYTEOUT1)
       EQUIVALENCE (BYTE2,BYTEOUT2)
       OPEN (UNIT=10,FILE='FLASH4.BNM',STATUS='OLD')
       OPEN (UNIT=20,FILE='RECEIVE.DAT',STATUS='OLD')
       OPEN (UNIT=30,FILE='CONVERT.DAT',STATUS='OLD')

       RESETADDR = #160
       LINKADDR = #150
       NWORDS = 0

       START = 1

       TOTALNBYTES = 0
       RECORD = 0

       WRITE(*,*) ' ENTER LOOP COUNT'
       READ(*,*) LOOPCOUNT

        DO WHILE (RECORD.EQ.0)
10        READ(10,5) DELIM,NBYTES,ADDR,RECORD,
     1           (DATA(I), I=START,START + NBYTES -1)
5         FORMAT (A1,Z2,Z4,Z2,50Z2)
               START = START + NBYTES
                    TOTALNBYTES = TOTALNBYTES + NBYTES

        END DO

       NWORDS = TOTALNBYTES/3
       BYTE2 = IAND(NWORDS,#FF)
       TOPBYTE = IAND(NWORDS,#FF00)
       BYTE1 = ISHFT(TOPBYTE,-8)

8        FORMAT (Z2)

        CALL PORTOUTBYTE(RESETADDR,1)
        CALL PORTOUTBYTE(RESETADDR,0)

        CALL LINKOUTBYTE(LINKADDR,BYTEOUT1)
       DO 100 K = 1,LOOPCOUNT
100    END DO

            CALL LINKOUTBYTE(LINKADDR,BYTEOUT2)
          DO 200 K = 1,LOOPCOUNT
200       END DO

           DO 20 I=1,TOTALNBYTES
              CALL LINKOUTBYTE(LINKADDR,DATA(I))
            DO 300 K = 1,LOOPCOUNT
300         END DO

20     END DO

       CALL LINKOUTBYTE(LINKADDR,55)

            DO 400 K = 1,LOOPCOUNT
400         END DO

       END
```

```
{************************************************************
* NAME: SETUP.DSP                                           *
*                                                           *
* DESCRIPTION: SETS UP ARRAYS WHICH CONTAIN CONECTIONS      *
*              BETWEEN NODES AND THE CROSSBAR SWITCH.       *
*                 TABLES ARE LOADED VIA C012                *
************************************************************}

.MODULE/ROM/SEG=int_pm/ABS=H#005DSETUP;
.INCLUDE                                        <E:\ADI_DSP\INCLUDE\DEF2105.h>;
.VAR/DM/RAM             node_id[32];
.VAR/DM/RAM             link_no[32];
.VAR/DM/RAM             crossbar_link_no[32];
.VAR/DM/RAM             connection_used_unused[32];
{.INIT                  node_id : <nodeid.dat>;
.INIT                   link_no : <linkno.dat>;
.INIT                   crossbar_link_no: <crossbar.dat>;}
.GLOBAL                 node_id;
.GLOBAL                 link_no;
.GLOBAL                 crossbar_link_no;
.GLOBAL                 connection_used_unused;
.PORT                   read_c004;
.PORT                   write_c004;
.PORT                   c004_input_status;
.PORT                   c004_output_status;
.PORT                   read_c012;
.PORT                   write_c012;
.PORT                   input_status;
.PORT                   output_status;
.GLOBAL                 read_c004;
.GLOBAL                 write_c004;
.GLOBAL                 c004_input_status;
.GLOBAL                 c004_output_status;
.GLOBAL                 read_c012;
.GLOBAL                 write_c012;
.GLOBAL                 input_status;
.GLOBAL                 output_status;
.EXTERNAL           read_bytes;

                        IMASK=0;                        {disable interrupts}
                        AX0=0;                           {load AX0}
                        DM(input_status)=AX0; {disable InputInt}
                        DM(output_status)=AX0;{disable OutputInt}
                        AY0=DM(read_c012);              {dummy read}
                        L0 = %node_id;                   {initialize L0}
                        L1 = %link_no;                   {initialize L1}
                        L2 = %crossbar_link_no;          {initialize L2}
                        L3 = %connection_used_unused; {initialize L3}
                        M0 = 1;                         {set increment to 1}
                        M1 = 1;                         {set increment to 1}
                        M2 = 1;                         {set increment to 1}
                        M3 = 1;                         {set increment to 1}


                    I0 = ^node_id;                       {set pointer}
                    I1 = ^link_no;                       {set pointer}
                    I2 = ^crossbar_link_no;   {set pointer}
                    I3 = ^connection_used_unused; {set pointer}

                    CNTR = %node_id;                     {set to CNTR to length
                                                          of array}
                    DO load_buffer UNTIL CE;  {loop until CNTR=0}

wait:               AX0=DM(input_status);     {test input status of C012}

{load AY0}
                    AR=AX0 AND AY0;                      {test LSB of input
                                                         status}
                    IF EQ JUMP wait;                     {if 0 keep looping}

                    AY0=H#FF;                            {load AY0}

                    AX0=DM(read_c012);                   {read C012}
                    AR=AX0 AND AY0;                 {clear upper half ofword}

load_buffer:        DM(I0,M0)=AR;                        {load array containing
                                                          node ids}
                    CNTR = %link_no;                     {set counter to length
                                                         of link_no}

                    DO load_buffer1 UNTIL CE;  {loop until CNTR=0}
wait1:              AX0=DM(input_status);                {test input status of
                                                          C012}
                    AY0=1;                               {load AY0}
                    AR=AX0 AND AY0;                      {test LSB of input
                                                         status}
```

```
                    IF EQ JUMP wait1;                       {if 0 keep looping}
                    AY0=H#FF;                               {load AY0}
                    AX0=DM(read_c012);                      {read C012}
                    AR=AX0 AND AY0;                         {clear upper half of
                                                             word}

load_buffer1:       DM(I1,M1)=AR;                           {load array containing
                                                             node ids}

                    CNTR = %crossbar_link_no;  {set counter to length
                                                        of crossbar_link_no}
                    DO load_buffer2 UNTIL CE;  {loop until CNTR=0}
wait2:              AX0=DM(input_status);     {test input status of
                                                        C012}
                    AY0=1;                                  {load AY0}
                    AR=AX0 AND AY0;                         {test LSB of input
                                                            status}
                    IF EQ JUMP wait2;                       {if 0 keep looping}
                    AY0=H#FF;                               {load AY0}
                    AX0=DM(read_c012);                      {read C012}
                    AR=AX0 AND AY0;                         {clear upper half of
                                                            word}
load_buffer2:       DM(I2,M2)=AR;                           {load array containing
                                                             node ids}
                    CNTR = %connection_used_unused;{set CNTR to length
                                                            of array}
                  AX0 = 0;                                  {load AX0}
                  DO clear_buffer UNTIL CE;{loop until CNTR=0}
clear_buffer:  DM(I3,M3) = AX0;                             {load array with 0's}
                    call read_bytes;                        {call array which
                                                            reads bytes from
                                                            nodes}

.ENDMOD;
```

```
{*************************************************************
* NAME: READBYTE                                             *
*                                                            *
* DESCRIPTION:READS THE THREE BYTES SENT BY THE NODES AND    *
*                INSERTS THEM INTO DATA MEMORY ADDRESS       *
*                                                SPACE       *
*                                                            *
* DATE:Wed 24-08-1994                                        *
{*************************************************************

.MODULE/ROM/SEG=int_pm                    readbytes;
.VAR/DM                                   source_node;
.VAR/DM                                   link_num;
.VAR/DM                                   destination_node;
.VAR/DM                                   byte_count;
.EXTERNAL                                 c004_input_status;
.EXTERNAL                                 read_c004;
.EXTERNAL                                 decode_request;
.ENTRY                                    read_bytes;
.GLOBAL                                   source_node;
.GLOBAL                                   link_num;
.GLOBAL                                   destination_node;

read_bytes:       AX0=3;                  {set AX0}
                  DM(byte_count)=AX0;     {set byte_count}
not_received:     AX0=0;                  {reset AX0}
                  AX0=DM(c004_input_status);{load input status of C012}
                  AY0=1;                  {load AY1}
                  AR=AX0 AND AY0;         {look at LSB}
                  IF EQ JUMP not_received;{test if =0}
                                          {i.e C012 empty}
                  AX0=2;                  {load AX0}
                  AY1=DM(byte_count);     {load byte_count
                                          into AY1}
                  AR=AX0-AY1;             {find which byte
                                          is present}
                  IF LT JUMPfirst_byte;
                  IF EQ JUMP second_byte;
                  IF GT JUMP third_byte;

first_byte:       AY0=H#FF;               {load AY0}
                  AX0=DM(read_c004);        {read port}
                  AR=AX0 AND AY0;         {mask off top byte}
                  DM(source_node)=AR;       {store result
                                           in DM}
                  AR=AY1-1;               {-1 from byte_
                                          count}
                  DM(byte_count)=AR;        {store new value
                                          for byte_count}
                  JUMP not_received;

second_byte:      AY0=H#FF;               {load AY0}
                  AX0=DM(read_c004);        {read port}
                  AR=AX0 AND AY0;         {mask off top
                                          byte}
                  DM(link_num)=AR;              {store result
                                                  in DM}
                  AR=AY1-1;               {-1 from byte_
                                          count}
                  DM(byte_count)=AR;            {store new value
                                                  for byte_count}
                  JUMP not_received;

third_byte:       AY0=H#FF;               {load AY0}
                  AX0=DM(read_c004);        {read port}
                  AR=AX0 AND AY0;                 {mask off top
                                                  byte}
                 DM(destination_node)=AR;   {store result in DM}
                  JUMP decode_request;

.ENDMOD;
```

```
{*****************************************************************
* NAME: DECODE.DSP                                               *
*                                                                *
* DESCRIPTION: DECIDES WHETHER MESSAGE FROM NODE IS A CONNECTION*
* OR DISCONNECTION REQUEST                                       *
*                                                                *
* DATE:Wed 24-08-1994                                            *
*                                                                *
*****************************************************************}

.MODULE/ROM/SEG=int_pm          decoderequest;
.EXTERNAL                       link_num;
.EXTERNAL                       make_connection;
.EXTERNAL                       break_connection;
.ENTRY                          decode_request;

decode_request:                 AY0=128;          {set AY0}
                                AX0=DM(link_num); {load AX0}
                                AR=AX0 AND AY0;   {look at MSBit}
                                IF EQ JUMP make_connection;
                                IF NE JUMP break_connection;

.ENDMOD;
```

```
{*************************************************************
* NAME: FINDSOUR.DSP;                                        *
*                                                            *
* DESCRIPTION: SEARCHES ARRAYS TO FIND WHICH LINK ON THE     *
*CROSSBAR SWITCH THE SOURCE NODE IS CONNECTED                *
* TO.                                                        *
*                                                            *
* DATE:Thu 25-08-1994                                        *
*                                                            *
*************************************************************}

.MODULE/ROM/SEG=int_pm          makeconnection;
.VAR                            source_node_crossbar_link;
.VAR                            pointer_to_source_node_link;
.GLOBAL                         pointer_to_source_node_link;
.GLOBAL                         source_node_crossbar_link;
.EXTERNAL                       source_node;
.EXTERNAL                       source_node_link_num;
.EXTERNAL                       connection_failed;
.EXTERNAL                       write_c012;
.EXTERNAL                       output_status;
.ENTRY                          find_crossbar_link_for_source;

find_crossbar_link_for_source:  AX0=DM(source_node);{load address of
                                                      source node}
                                AY1=0;{load AY1}
match_source:                   AY0=DM(I0,M0);    {load value from
                                                   table}
                                AR=AY1+1;         {count number of
                                                   times round loop}
                                AY1=AR;           {store new value of
                                                   AY1}
                                AR=AY0-AX0;       {test for match}

                                IF EQ JUMP load_pointer; {if match then
                                                          exit loop}
                                IF LT JUMP match_source; {keep searching
                                                          table}
                                IF GT JUMP match_source; {keep searching
                                                          table}

                                load_pointer:AX1=AY1; {store loop count
                                                       in AX1}
                                AY1=I1;               {put pointer to
                                                       link_no in AY1}
                                AY0=1;                {load AY1}
                                AR=AX1-AY0;           {calculate actual
                                                       amount to be added
                                                       to I1}
                                MR0=AR;               {store result in
                                                       MR0}
                                AR=MR0+AY1;       {calculate new
                                                   pointer address
                                                   for link_no}

                                I1=AR;        {load I1 with new
                                               value}
                                AX0=DM(source_node_link_num);{load linknum}
                                AY1=0;                {load AY1}
                                match_link:AY0=DM(I1,M1); {load value
                                                           from table}
                                AR=AY1+1;         {count number of
                                                   times round loop}
                                AY1=AR;           {store new value
                                                   of AY1}
                                AR=AX0-AY0;   {check for match}

                                IF EQ JUMP find_link_on_crossbar; {match}
                                IF LT JUMP match_link;{not-match}
                                IF GT JUMP match_link;{not-match}

find_link_on_crossbar:          TOGGLE FLAG_OUT; {used as test}
                                AY0=AY1;          {load AY0 with no
                                                   of times round
                                                   match_link}
                                AR=AX1+AY0;       {calculate total
                                                   distance from start}
                                AX1=AR;           {load AX1 with
                                                   result}
                                AX0=I2;           {load AX0 with
                                                   start address of
                                                   array}
                                AY1=2;
                                AR=AX1-AY1;       {find value
                                                   to be added
```

```
                                    to I2)
AY1=AR;                             (load AY1)
AR=AX0+AY1;                         (calculate new
                                    value of pointer)
I2=AR;                              (load start address
                                    into I2)
AX0=I3;                              (repeat for I3)
AR=AX0+AY1;                         (calculate new
                                    value of pointer)
I3=AR;                              (load new value
                                    into I3)
DM(pointer_to_source_node_link)=I3;
AX0=DM(I2,M2);
DM(source_node_crossbar_link)=AX0;
                                    (store pointers
                                    for later use)
RTS;
ENDMOD;
```

```
{*****************************************************************
* NAME:MAKE.DSP                                                  *
*                                                                *
* DECRIPTION:CALLS ROUTINES REQUIRED TO MAKE A CONNECTION;       *
*                                                                *
* DATE:Wed 31-08-1994                                            *
*                                                                *
*****************************************************************}

.MODULE/ROM/SEG=int_pm          makesconnection;
.VAR                            source_node_link_num;
.GLOBAL                         source_node_link_num;
.EXTERNAL                       link_num;
.EXTERNAL                       connection_failed;
.EXTERNAL                       find_destination_node;
.EXTERNAL                       connection_failed;
.EXTERNAL                       test_link_inuse;
.EXTERNAL                       program_crossbar;
.EXTERNAL                       find_crossbar_link_for_source;
.ENTRY                          make_connection;

make_connection:                AX0=DM(link_num); {load AX0 with
                                                   link_num}
                                DM(source_node_link_num)=AX0;{load value
                                                              into different
                                                               variable}
                                AY0=3; {load AY0}
                                AR=AY0-AX0;             {test link
                                                        number is not
                                                        greater than 4}
                                IF LT JUMP connection_failed;{if link number
                                                              not valid then
                                                              fail attempt}

{Call routine which searches tables to find which link on the crossbar
switch source node is connected to}

                                CALL find_crossbar_link_for_source;
test_connection:                AX0=DM(I3,M3);          {load AX0 from array}
                                AY0=1;                  {load AY1}
                                AR=AY0-AX0;             {test that link is
                                                        free - i.e '0'}
                                IF EQ JUMP connection_failed;

{call routine which locates a link on the crossbar switch the destination node could be
connected to}

                                IF NE CALL find_destination_node;
                                CALL test_link_inuse;  {searches for free
                                                        links on destination
                                                        node}
                                CALL program_crossbar; {program crossbar
                                                        switch}

.ENDMOD;
```

```
{*************************************************************
* NAME: FINDDEST.DSP                                         *
*                                                            *
* DESCRIPTION: FINDS LINK NUMBER OF DESTINATION NODE AND LINK*
*    ON CROSSBAR IT IS CONNECTED TO.                         *
*                                                            *
* DATE:Tue 30-08-1994                                        *
*************************************************************}
.MODULE/ROM/SEG=int_pm          finddestinationnode;
.VAR                            destination_node_link_num;
.VAR                            destination_node_crossbar_link;
.VAR                            pointer_to_destination_node_link;
.GLOBAL                         pointer_to_destination_node_link;
.GLOBAL                         destination_node_link_num;
.GLOBAL                         destination_node_crossbar_link;
.EXTERNAL                       destination_node;
.EXTERNAL                       node_id;
.EXTERNAL                       link_no;
.EXTERNAL                       crossbar_link_no;
.EXTERNAL                       connection_used_unused;
.EXTERNAL                       connection_failed;
.EXTERNAL                       program_crossbar;
.ENTRY                          find_destination_node;

find_destination_node:          I0=^node_id;    {reset pointers}
                                I1=^link_no;
                                I2=^crossbar_link_no;
                                I3=^connection_used_unused;

                                AX0=DM(destination_node);  {load destination
                                                            node}
                                AY1=0;                      {reset ay1}

match_destination:              AY0=DM(I0,M0);              {load node id}
                                AR=AY1+1;                  {increment counter}
                                AY1=AR;                     {store counter
                                                             value}
                                AR=AX0-AY0;                 {test for match
                                                            for dest.node}
                                IF EQ JUMP load_pointer1; {exit loop}
                                IF LT JUMP match_destination;{keep looping}
                                IF GT JUMP match_destination;{keep looping}

load_pointer1:                  AR=AY1-1;                   {-1 from loop
                                                            count}
                                MR0=AR;                     {store result}
                                AY1=I1;                     {put pointer to
                                                            link_no in AY1}
                                AR=MR0+AY1;                 {add offset to
                                                            I1}
                                I1=AR;                      {load new value
                                                            of I1}
                                AY1=I2;                     {find new value
                                                            for pointer to
                                                            crossbarlinkno}
                                AR=MR0+AY1;                 {add offset to
                                                            I2}

                                I2=AR;                      {load new value
                                                            of I2}
                                AY1=I3;                     {find new value
                                                            for pointer to
                                                            connectionused}
                                AR=MR0+AY1;                 {add offset to
                                                            I3}
                                I3=AR;                      {load new value
                                                            of I2}

                                RTS;
                                .ENDMOD;
```

```
{*************************************************************
* NAME: TESTLINK.DSP                                         *
*                                                            *
* DESCRIPTION:SERCHES LINKS FROM DESTINATION NODE TO CROSSBAR *
*  LOOKING FOR A FREE LINK TO CONNECT THE                    *
*  SOURCE NODE TO.                                           *
*                                                            *
* DATE:Tue 30-08-1994                                        *
*************************************************************}

.MODULE/ROM/SEG=int_pm          testforfreelink;
.EXTERNAL                       connection_failed;
.EXTERNAL                       destination_node_link_num;
.EXTERNAL                       destination_node_crossbar_link;
.EXTERNAL                       pointer_to_destination_node_link;
.ENTRY                          test_link_inuse;

test_link_inuse:                AY1=0;                  {reset ay1}
load_flag:                      AY0=DM(I3,M3);          {load flag}
                                AR=AY1+1;               {incement counter}
                                AY1=AR;
                                AX0=5;
                                AR=AY1-AX0;             {test if looped
                                                        4 times}
                                IF EQ JUMP connection_failed; {if checked
                                                              all
                                                              4 links then
                                                              failed}
                                AR=AY0-1;
                                IF EQ JUMP load_flag;{if connection used
                                                     then test further}
                                IF LT JUMP find_dest_link;{connection free
                                                          then proceed}

find_dest_link:                 AY0=I3;                 {load AY0 with
                                                        pointer}
                                AR=AY0-1;               {-1 from pointer}
                                DM(pointer_to_destination_node_link)=AR;
                                AR=AY1-1;               {-1 from AY1}
                                AY1=AR;                 {store loop count}
                                AX1=I1;                 {set pointer}
                                AR=AX1+AY1;             {find new value
                                                        of pointer}
                                I1=AR;                  {load pointer}
                                AX0=DM(I1,M1);          {load AX0}
                                DM(destination_node_link_num)=AX0;{store in
                                                                  DM}
crossbar_link:                  AX1=I2;                 {set pointer}
                                AR=AX1+AY1;             {find new value
                                                        of I2}
                                I2=AR;                  {load new value}
                                AX0=DM(I2,M2);          {store link
                                                        number}
                                DM(destination_node_crossbar_link)=AX0;

RTS;

.ENDMOD;
```

```
{*************************************************************
* NAME: CROSS.DSP                                            *
*                                                            *
* DESCRIPTION: PROGRAMS REQUIRED CONNECTION ON CROSSBAR SWITCH.*
*   UPDATES FLAGS INDICATING WHETHER LINKS ARE IN USE.       *
*   SENDS ACKNOWLEDGE BYTE.                                  *
*   RESETS POINTERS.                                         *
*                                                            *
* DATE:Tue 30-08-1994                                        *
*************************************************************}

.MODULE/ROM/SEG=int_pm          programcrossbar;
.EXTERNAL                       source_node_crossbar_link;
.EXTERNAL                       destination_node_crossbar_link;
.EXTERNAL                       pointer_to_source_node_link;
.EXTERNAL                       pointer_to_destination_node_link;
.EXTERNAL                       write_c012;
.EXTERNAL                       write_c004;
.EXTERNAL                       output_status;
.EXTERNAL                       destination_node_link_num;
.EXTERNAL                       node_id;
.EXTERNAL                       link_no;
.EXTERNAL                       crossbar_link_no;
.EXTERNAL                       connection_used_unused;
.EXTERNAL                       read_bytes;
.ENTRY                          program_crossbar;

program_crossbar:               AX0=DM(source_node_crossbar_link);
                                AY0=DM(destination_node_crossbar_link);
not_ready0:                     AX1=0;{load AX1}
                                AX1=DM(output_status);{check output
                                                       status}
                                AY1=1;                 {look AY1}
                                AR=AX1 AND AY1;        {look at LSBit}
                                IF EQ JUMP not_ready0;{keep looping if
                                                       not ready}

                                AX1=1;                 {load AX1}
                                DM(write_c012)=AX1;    {initiates discon
                                                       on C004}

not_ready1:                     AX1=0;                 {load AX1}
                                AX1=DM(output_status);{check output
                                                       status}
                                AY1=1;                 {look AY1}
                                AR=AX1 AND AY1;        {look at LSBit}
                                IF EQ JUMP not_ready1;{keep looping if
                                                       not ready}

                                DM(write_c012)=AX0;{send value of link
                                                       to be connected to
                                                       C004}

not_ready2:                     AX1=0;                 {load AX1}
                                AX1=DM(output_status);{check output
                                                       status}
                                AY1=1;                 {look AY1}
                                AR=AX1 AND AY1;        {look at LSBit}
                        IF EQ JUMP not_ready2;{keep looping if
                                                       not ready}
                                DM(write_c012)=AY0;{send value of link
                                                       to be connected to
                                                       C004}

update_connection_table:        I3=DM(pointer_to_source_node_link);{load
                                                                    pointer}
                                AX0=1;                 {load AX0}
                                DM(I3,M3)=1;           {set to 1 - connected}
                                I3=DM(pointer_to_destination_node_link);
                                AX0=1;                 {load AX1}
                                DM(I3,M3)=1;           {set to 1 - connected}

send_acknowledge_byte:          AX0=DM(destination_node_link_num);{load AX0}
                                DM(write_c004)=AX0;    {send num of dest node
                                                        as ack. byte}
                                I0=^node_id;    {reset pointers}
                                I1=^link_no;
                                I2=^crossbar_link_no;
                                I3=^connection_used_unused;

                                JUMP read_bytes;

.ENDMOD;
```

```
{*************************************************************F
* NAME: BREAK.DSP                                                *
*                                                                *
* DESCRIPTION: SEPARATES OUT SOURCE AND DESTINATION LINK         *
*   NUMBERS                                                      *
*   CALLS ROUTINES TO BREAK A CONNECTION                         *
*                                                                *
* DATE:Wed 31-08-1994                                            *
*                                                                *
*************************************************************)

.MODULE/ROM/SEG=int_pm          breakconnection;
.EXTERNAL                       destination_node_link_num;
.EXTERNAL                       source_node_link_num;
.EXTERNAL                       link_num;
.EXTERNAL                       find_crossbar_link_for_source;
.EXTERNAL                       find_destination_node;
.EXTERNAL                       find_destination_link;
.EXTERNAL                       disconnect_link;
.EXTERNAL                       connection_failed;
.ENTRY                          break_connection;

break_connection:               AX0=DM(link_num);          {load link_num}
                                AY0=7;                     {load AY0}
                                AR=AX0 AND AY0;            {extract first
                                                           four bits}
                                DM(source_node_link_num)=AR; {load variable}
                                AX0=DM(source_node_link_num);{load AX0}
                                AY0=3;                     {load AY0}
                                AR=AY0-AX0;                {check value of
                                                           link num is
                                                           sensible}
                                IF LT JUMP connection_failed; {fail}

                                AY0=127;                   {load AY0}
                                AR=AX0 AND AY0;            {remove flag bit}
                                SR0=AR;                    {load SR0}
                                SR=LSHIFT SR0 BY -3(LO);{remove source node
                                                           link number}
                                AX0=SR0;                   {load AX0}

                                DM(destination_node_link_num)=AX0;{load VAR}
                                AY0=3;{load AY0}
                                AR=AY0-AX0;{check linkno is
                                reasonable}
                                IF LT JUMP connection_failed; {fail}

                                CALL find_crossbar_link_for_source;
                                CALL find_destination_node;
                                CALL find_destination_link;
                                CALL disconnect_link;

.ENDMOD;
```

```
{*************************************************************
* NAME: FINDLINK.DSP                                         *
*                                                            *
* DESCRIPTION: FIND LINK ON CROSSBAR DESTINATION NODE IS *
*   CONNECTED TO IN ORDER TO BREAK A                         *
*   CONNECTION.                                              *
* *
* DATE:Wed 31-08-1994                                        *
*                                                            *
*************************************************************}

.MODULE/ROM/SEG=int_pm          findlinkoncrossbar;
.EXTERNAL                       destination_node_link_num;
.EXTERNAL                       destination_node_crossbar_link;
.EXTERNAL                       pointer_to_destination_node_link;
.ENTRY                          find_destination_link;

find_destination_link:          AX0=DM(destination_node_link_num);

loop1:                          AY1=0;                  {load AY1}
                                AY0=DM(I1,M1);          {load from table}
                                AR=AY1+1;               {increment counter}
                                AY1=AR;                 {load new value of
                                                        counter}
                                AX0=DM(destination_node_link_num);
                                AR=AY0-AX0;{test for match}

                                IF EQ JUMP find_link_on_crossbar;
                                IF GT JUMP loop1;
                                IF LT JUMP loop1;

find_link_on_crossbar:          AR=AY1-1;               {-1 from counter}
                                AY1=AR;                 {load new value}
                                AX1=I2;                 {load value of pointer}
                                AR=AX1+AY1;             {find new value of
                                                        pointer}
                                I2=AR;                  {load new value of
                                                         pointer}
                                AX1=I3;{                load value of pointer}
                                AR=AX1+AY1;             {find new value of
                                                         pointer}
                                I3=AR;                  {load new value of
                                                         pointer}
                                DM(pointer_to_destination_node_link)=AR;
                                AX0=DM(I2,M2);          {load AX0}
                                DM(destination_node_crossbar_link)=AX0;
RTS;

.ENDMOD;
```

```
{*************************************************************************
*                                                                        *
* NAME: DISCON.DSP                                                       *
*                                                                        *
* DESCRIPTION: DISCONNECTS A CONNECTION ON THE CROSSBAR SWITCH AND       *
*   SENDS AN ACKNOWLEDGE BYTE TO NODE.                                   *
*                                                                        *
* DATE: Mon 19-09-1994                                                   *
*                                                                        *
*************************************************************************}

.MODULE/ROM/SEG=int_pm          break_a_connection;
.EXTERNAL                       source_node_crossbar_link;
.EXTERNAL                       destination_node_crossbar_link;
.EXTERNAL                       pointer_to_source_node_link;
.EXTERNAL                       pointer_to_destination_node_link;
.EXTERNAL                       write_c012;
.EXTERNAL                       write_c004;
.EXTERNAL                       output_status;
.EXTERNAL                       destination_node_link_num;
.EXTERNAL                       node_id;
.EXTERNAL                       link_no;
.EXTERNAL                       crossbar_link_no;
.EXTERNAL                       connection_used_unused;
.EXTERNAL                       read_bytes;
.ENTRY                          disconnect_link;

disconnect_link:                AX0=DM(source_node_crossbar_link);
                                AY0=DM(destination_node_crossbar_link);

not_ready0:                     AX1=0;{load AX1}
                                AX1=DM(output_status);{check output
                                                       status}
                                AY1=1;{load AY1}
                                AR=AX1 AND AY1;{look at LSBit}
                                IF EQ JUMP not_ready0;{keep looping
                                                       if not ready}

                                AX1=6;                 {load AX1}
                                DM(write_c012)=AX1;{initiates discon
                                                       on C004}

not_ready1:                     AX1=0;{load AX1}
                                AX1=DM(output_status);{check output
                                                       status}
                                AY1=1;{load AY1}
                                AR=AX1 AND AY1;{look at LSBit}
                                IF EQ JUMP not_ready1;{keep looping
                                                       if not ready}

                                DM(write_c012)=AX0;{send value of link
                                                       to be disconnected
                                                       to C004}

not_ready2:                     AX1=0;                 {load AX1}
                                AX1=DM(output_status);{check output
                                                       status}
                                AY1=1;                 {load AY1}
                                AR=AX1 AND AY1;        {look at LSBit}
                                IF EQ JUMP not_ready2;{keep looping
                                                       if not ready}
                                DM(write_c012)=AY0;{send value of link
                                                       to be disconnected
                                                       to C004}

update_connection_table:        I3=DM(pointer_to_source_node_link);{load
                                                       pointer}
                                DM(I3,M3)=0;  {reset to 0 - disconnected}
                                I3=DM(pointer_to_destination_node_link);
                                                       {load pointer}
                                DM(I3,M3)=0;   {reset to 0 - disconnected}

send_acknowledge_byte:          AX0=DM(destination_node_link_num);{load AX0}
                                DM(write_c004)=AX0; {send no. of dest node
                                                       as ack. byte}

                                I0=^node_id;   {reset pointers}
                                I1=^link_no;
                                I2=^crossbar_link_no;
                                I3=^connection_used_unused;

                                JUMP read_bytes;

.ENDMOD;
```

```
NAME                    STATE2.PLD TOKEN PASSING AND FIFO CLOCKING;
PARTNO                  STATE MACHINE;
REVISION                01;
DATE                    13/02/93;
DESIGNER                LESLEY BISSLAND;
COMPANY                 GLAGOW UNIVERSITY;
LOCATION                STATE MACHINE;
ASSEMBLY                ;
DEVICE                  P22V10;
FORMAT                  -j;

/*************************************************************************/
/*CLOCKS FIFO AND PASSES TOKEN USING TWO SEPARATE STATE MACHINES.*/
/*ALSO SETS RESET SIGNALS.                                       */
/*                                                               */
/*************************************************************************/

/**INPUTS**/

PIN 1                   =       CLK;
PIN 2                   =       TOKENIN;
PIN 3                   =       !TOKENACCEPTED;
PIN 4                   =       RESET;
PIN 5                   =       KEEPTOKEN;
PIN 6                   =       IACK;
PIN 7                   =       !EF;
PIN 8                   =       SYSCONTROL;
PIN 9                   =       D0;
PIN 10                  =       RESETDRV;
/**PIN 11   =    TOKENPRESENT;**/

/**OUTPUTS**/

PIN 14                  =       IVALID;
PIN 15                  =       !CLOCKFIFO;
PIN 16                  =       !TOKENRECEIVED;
PIN 17                  =       !ENABLEBUFFER;
PIN 18                  =       TOKENOUT;
PIN 19                  =       LATCHRESET;
PIN 20                  =       SYSRESET;
PIN 21                  =       !FIFORESET;
PIN 22                  =       TOKENARRIVED;

FIELD TOKENSTATEBIT=            [TOKENARRIVED,TOKENOUT,TOKENRECEIVED,
                                ENABLEBUFFER];
FIELD FIFOSTATEBIT =            [IVALID,CLOCKFIFO];

$DEFINE TOKEN0 'b'0000
$DEFINE TOKEN1 'b'1011
$DEFINE TOKEN2 'b'0100

$DEFINE FIFO0 'b' 00
$DEFINE FIFO1 'b' 01
$DEFINE FIFO2 'b' 11
$DEFINE FIFO3 'b' 10

/**RESETS AND PRESETS**/

IVALID.SP               =       'b'0;
IVALID.AR               =       'b'0;
ENABLEBUFFER.SP         =       'b'0;
ENABLEBUFFER.AR         =       'b'0;
CLOCKFIFO.SP            =       'b'0;
CLOCKFIFO.AR            =       'b'0;
TOKENOUT.SP             =       'b'0;
TOKENOUT.AR             =       'b'0;
TOKENRECEIVED.SP        =       'b'0;
TOKENRECEIVED.AR        =       'b'0;
TOKENARRIVED.SP         =       'b'0;
TOKENARRIVED.AR         =       'b'0;
LATCHRESET.AR           =       'b'0;
LATCHRESET.SP           =       'b'0;

/**DEFINITIONS**/

NOTOKEN                 =       !TOKENIN & !RESET & !TOKENACCEPTED;
TOKEN                   =       TOKENIN & !RESET & !TOKENACCEPTED;
TOKENPASSED             =       TOKENACCEPTED & !RESET & !TOKENIN;
TOKENNOTPASSED          =       !TOKENACCEPTED & !RESET & !TOKENIN;
HNOTOKEN                =       !TOKENIN & !RESET & !TOKENACCEPTED &
                                 !KEEPTOKEN;
HTOKEN                  =       TOKENIN & !RESET & !TOKENACCEPTED &
                                 !KEEPTOKEN;
HOLD                    =       KEEPTOKEN & !RESET;
CLEAR                   =       RESET;

FIFOEMPTY               =       EF & !RESET & !IACK;
FIFONOTEMPTY            =       !EF & !RESET & !IACK & TOKENARRIVED;
```

```
TOKENNOTTHERE       =       !EF & !RESET & !IACK & !TOKENARRIVED;
DATASENT            =       IACK & !RESET;
DATANOTSENT         =       !IACK & !RESET;
TOKENISTHERE        =       TOKENARRIVED;

SEQUENCE TOKENSTATEBIT{

PRESENT TOKEN0              IF NOTOKEN    NEXT TOKEN0;   /*TOKEN NOT ARRIVED*/
                            IF TOKEN      NEXT TOKEN1;   /*TOKEN ARRIVED*/
                            IF CLEAR      NEXT TOKEN0;   /*RESET*/

PRESENT TOKEN1              IF HTOKEN     NEXT TOKEN2;   /*TOKEN STILL THERE*/
                            IF HNOTOKEN   NEXT TOKEN2;   /*TOKEN REMOVED*/
                            IF HOLD       NEXT TOKEN1;
                            IF CLEAR      NEXT TOKEN0;   /*RESET*/

PRESENT TOKEN2              IF TOKENPASSEDNEXT TOKEN0;   /*TOKEN PASSED*/
                            IF TOKENNOTPASSEDNEXT TOKEN2;/*TOKEN NOT
                                                           PASSED*/
                        IF TOKEN      NEXT TOKEN0;
                        IF CLEAR      NEXT TOKEN0;   /*RESET*/
}

SEQUENCE FIFOSTATEBIT{

PRESENT FIFO0       IF FIFOEMPT             NEXT FIFO0;
                    IF FIFONOTEMPTY         NEXT FIFO1;
                    IF TOKENNOTTHERE        NEXT FIFO0;
                    IF DATASENT             NEXT FIFO0;
                    IF RESET                NEXT FIFO0;

PRESENT FIFO1       IF RESET                NEXT FIFO0;
                    IF FIFOEMPTY            NEXT FIFO2;
                    IF FIFONOTEMPTY         NEXT FIFO2;

PRESENT FIFO2       IF DATANOTSENT          NEXT FIFO2;
                    IF DATASENT             NEXT FIFO0;
                    IF RESET                NEXT FIFO0;

PRESENT FIFO3       NEXT FIFO0;
}

LATCHRESET.d        =       D0 & SYSCONTROL # LATCHRESET & !SYSCONTROL;
SYSRESET            =       RESETDRV # LATCHRESET;
FIFORESET           =       SYSRESET;
```

Appendix C:

Source code for parallel energy minimisation.

```
C
C
C               ******************************************************
C               *                                                    *
C               *       H  H  OO   SSSS TTTTTT  M   M  II  N   N     *
C               *       H  H O  O  S      TT    MM MM  II  NN  N     *
C               *       HHHH O  O  SSSS   TT    M M M  II  N N N     *
C               *       H  H O  O     S   TT    M   M  II  N  NN     *
C               *       H  H  OO   SSSS   TT    M   M  II  N   N     *
C               *                                                    *
C               *                                                    *
C               *                                                    *
C               *      NAME:    HOSTMIN.FOR                          *
C               *                                                    *
C               *      DESCR:   BLOCK-DIAGONAL NEWTON-RAPHSON ENERGY *
C               *               MINIMISATION, WITH CONSTRAINTS IF    *
C               *               DESIRED.                             *
C               *                                                    *
C               *      ENTRY POINTS:  SELF CONTAINED                 *
C               *                                                    *
C               *                                                    *
C               *                     DAVID WHITE                    *
C               *                                                    *
C               *                COPYRIGHT (C), 1986                 *
C               *                ALL RIGHTS RESERVED                 *
C               *                                                    *
C               ******************************************************
C
C  BLOCK DIAGONAL ENERGY MINIMISATION PROGRAM
C
C  THIS PROGRAM IS BASED ON THE GENERAL NEWTON-RAPHSON MINIMISATION
C  ALGORITHM.  THE MATRIX OF THE SECOND DERIVATIVES OF THE STERIC
C  ENERGY WITH RESPECT TO THE ATOMIC COORDINATES IS REDUCED TO A BLOCK
C  DIAGONAL FORM, I.E. ALL VALUES OTHER THAN D2VS/DXI DXJ, I,J = 1,2
C  OR 3 FOR EACH ATOM, ARE SET TO ZERO.
C
C
C   -- MASTER SEGMENT & FILE READER FOR MOLECULAR MECHANICS ROUTINES --
C
c       INCLUDE 'HOST.INC'

      PROGRAM MOLMEC

      IMPLICIT NONE


      INCLUDE 'CHMCM3.INC'
C     INCLUDE 'HNCOM.INC'
      INTEGER TRNTAB,TRNTB2,NUMCON,INA,I,J
      REAL CELDIM
      DIMENSION CELDIM(6)
      DIMENSION TRNTAB(MXCHEM),TRNTB2(30)

      LOGICAL VALID
      CHARACTER ATNM*2,CHRANM*3,QQQ*1

C
      DATA TRNTAB/4,5,6,22,9,7,10,10,8,11,12,17,18,16,1,14,15,20,21/
      DATA TRNTB2/15,15,15,1,2,3,6,9,5,7,10,11,10,16,17,14,12,13
     1,13,18,19,22,23,24,25,26,27,28,29,30/

15    FORMAT(A40)
16    FORMAT(6F9.3)
17    FORMAT(I4)
18    FORMAT(A2,2X,I2,4X,3F9.4,2X,F10.5,2X,I2)
19    FORMAT(I1)
20    FORMAT(I2)
21    FORMAT(I3)
22    FORMAT(7I4)
C
                INQUIRE(FILE='chemmin.cwo',EXIST=VALID)
                IF(.NOT.(VALID)) STOP
C
                CALL MNINIT1
C
        OPEN(DLUNIN,FILE='chemmin.cwo')
C        OPEN(DLNOUT,FILE='MOLMEC.CWO',STATUS='NEW')
        OPEN(DLNOUT,FILE='MOLMEC.CWO')


C
C  -- ALL FILES OPEN, NOW READ CONTENTS OF INFILE --
C
      READ(DLUNIN,15) TITLE
      READ(DLUNIN,16) (CELDIM(I),I=1,6)
      READ(DLUNIN,17) NUMATS
        IF(NUMATS.GT.MXAT) THEN
```

```
          WRITE(*,*)' ***** ERROR  TOO MANY ATOMS *****'
        CLOSE(DLUNIN)
        STOP
        ENDIF
       DO 100 I=1,NUMATS
        READ(DLUNIN,18) ATNM,ATYNUM(I),(XO(I,J),J=1,3),CHARGE(I)
     1  ,MOLNUM(I)
        IF(I.LT.10)                    WRITE(CHRANM,19) I
        IF(I.GE.10.AND.I.LT.100)       WRITE(CHRANM,20) I
        IF(I.GE.100.AND.I.LT.1000)     WRITE(CHRANM,21) I
        ATMNAM(I)(1:3)=ATNM(1:2)
        ATMNAM(I)(4:6)=CHRANM(1:3)
        IF(ATYNUM(I).LT.20) ATYNUM(I)=TRNTAB(ATYNUM(I))
        IF(ATYNUM(I).EQ.Osp3.AND.(CHARGE(I).EQ.-1.0.OR.CHARGE(I)
     1  .EQ.-2.0)) ATYNUM(I)=Oanion
100    CONTINUE
       READ(DLUNIN,17) NUMCON
       DO 105 I=1,NUMCON
        READ(DLUNIN,22) INA,(ATMCON(I,J),J=1,MXCN)
105    CONTINUE
C
       CALL BDMIN
       WRITE(DLNOUT,15) TITLE
       WRITE(DLNOUT,16) (CELDIM(I),I=1,6)
       WRITE(DLNOUT,17) NUMATS
       DO 120 I=1,NUMATS
        ATYNUM(I)=TRNTB2(ATYNUM(I))
        IF(ATYNUM(I).EQ.22.AND.ATMNAM(I)(1:2).EQ.'C ') ATYNUM(I)=4
115     WRITE(DLNOUT,18) ATMNAM(I)(1:2),ATYNUM(I),(XO(I,J),J=1,3)
     1                  ,CHARGE(I),MOLNUM(I)
120    CONTINUE
       WRITE(DLNOUT,17) NUMCON
       DO 125 I=1,NUMCON
        WRITE(DLNOUT,22) I,(ATMCON(I,J),J=1,MXCN)
125    CONTINUE
C       CLOSE(DLUNIN,STATUS='DELETE')
       CLOSE(DLUNIN)
       CLOSE(DLNOUT)
C
       WRITE(*,*) '                Press RETURN to Continue.'
       READ(*,'(A)') QQQ
C
       END

       SUBROUTINE  BDMIN

       IMPLICIT NONE

       INCLUDE 'CHMCM3.INC'
                    include 'equiv.inc'
C       INCLUDE 'HNCOM.INC'
C       INCLUDE '\BOARD\F77\HSTLNKIF.INC'

       LOGICAL BFCERR,AFCERR,NFCERR,TFCERR
       INTEGER ITRCMP,NFIRST,LAST,BFLENG,I,NDIV,NMOD,L,
     1         NUMPROC
       REAL SHIFT2,ETOT,temp,sgdlsq,rmsdl
                    CHARACTER*64 FILE

                    INTEGER * 1 TEMP1

                    integer*2 error,ProcConn(4,4),netcast

                    EQUIVALENCE (TEMP1,TEMP)

10     FORMAT(/,'  ATOM   TYPE          X           Y           Z
     1  CHARGE  MOLECULE',/)
11     FORMAT(2X,A6,4X,I2,3X,4F12.5,6X,I2)
12     FORMAT(' MINIMISATION ABORTED DUE TO K(BOND STRETCH) OMISSIONS')
13     FORMAT(' MINIMISATION ABORTED DUE TO K( ANGLE  BEND) OMISSIONS')
14     FORMAT(' MINIMISATION ABORTED DUE TO K(NON - BONDED) OMISSIONS')
15     FORMAT(' MINIMISATION ABORTED DUE TO K(BOND TORSION) OMISSIONS')
16     FORMAT(/,' INITIAL POTENTIAL ENERGY = ',F12.4,' K.CAL PER MOLE',/)
17     FORMAT(///,' MINIMISATION ABANDONED DUE TO SINGULAR MATRIX',///)
18     FORMAT(' RMS VALUE OF dE/dX,dY,dZ = ',E12.4,' KCAL MOL-1 A-1')
19     FORMAT(/,' FINAL POTENTIAL ENERGY   = ',F12.4,' K.CAL PER MOLE',/)
20     format (14I4)
21     format (14F5.1)
C
C    -- SHIFTX IS THE INCREMENT IN ATOMIC COORDINATES USED FOR CALCULATING --
C    -- THE DERIVATIVES --
C

                    open (20,file = 'lpt1')
       CALL ASBOML
       CALL GETCOP
       CALL GETOPB
       IF(NPRINT.GT.1) THEN
         WRITE(*,10)
```

```
        WRITE(*,11) (ATMNAM(I),ATYNUM(I),XO(I,1),XO(I,2),XO(I,3)
     1                      ,CHARGE(I),MOLNUM(I),I=1,NUMATS)
       ENDIF
C
C    -- START CALCULATION PROPER --
C
       BFCERR=.FALSE.
       AFCERR=.FALSE.
       NFCERR=.FALSE.
       TFCERR=.FALSE.
       CALL POTE(ETOT,BFCERR,AFCERR,NFCERR,TFCERR)
       IF(BFCERR) THEN
         WRITE(*,12)
         RETURN
       ELSE
         IF(AFCERR) THEN
           WRITE(*,13)
           RETURN
         ELSE
          IF(NFCERR) THEN
            WRITE(*,14)
            RETURN
          ELSE
           IF(TFCERR) THEN
             WRITE(*,15)
             RETURN
           ENDIF
          ENDIF
         ENDIF
       ENDIF
C
C    -- NO MISSING FORCE CONSTANTS SO CONTINUE --
C
       IF(NPRINT.EQ.0) WRITE(*,16) ETOT
       IF(NUMITR.EQ.0) RETURN

       SHIFT2=SHIFTX*SHIFTX
       ITRCMP=0

C      CONFIGURES AND LOADS NODES

                     write(*,*) 'Enter no. of nodes in use'
       READ(*,*) NUMPROC
                   file= 'c:\comfort\lesley\min\nodemin.app'//char(0)

call configure(#180, numproc, #976f)
call reset(-1)
call load(-1, file, 100, error)

If (numproc.eq.4) then
                  do i=1,4
                    ProcConn(1,i)=4
                    ProcConn(2,i)=-1
                    ProcConn(3,i)=-1
                    ProcConn(4,i)=-1
                  end do
end if

if (numproc.eq.1) then
 do i = 1,4
    ProcConn(1,i) = 1
    ProcConn(2,i) = -1
    ProcConn(3,i) = -1
    ProcConn(4,i) = -1
 end do
end if

                netcast = -1

       call initialize(ProcConn, 100, error)


C      SENDS BUFFERS TO NODES

                     call send (NETCAST,buffer_atmdat0,1,total_atmdat0,100,error)
                     call send (NETCAST,buffer_atmdat1,2,total_atmdat1,100,error)
                     call send (NETCAST,buffer_moldat,3,total_moldat,100,error)
                     call send (NETCAST,buffer_ffp,4,total_ffp,100,error)
                     call send (NETCAST,buffer_cffp,5,total_cffp,100,error)
                     call send (NETCAST,buffer_contrl,6,total_contrl,100,error)
                     call send (NETCAST,buffer_constn,7,total_constn,100,error)

C      SEND BYTE ARRAYS SEPARATELY

                     CALL SEND (NETCAST,ATYNUM,8,LENGTH9,100,ERROR)
                     CALL SEND (NETCAST,BONDML,9,LENGTH10,100,ERROR)
                     CALL SEND (NETCAST,MOLNUM,10,LENGTH9,100,ERROR)
```

```
 999  write (5,*)'No of iterations =',itrcmp +1
C     SENDS COORDINATES TO NODES
      call send(NETCAST,XO1,42,INT2(length7),100,error)
C                    NUMPROC = 4
      NDIV = NUMATS / NUMPROC
      NMOD = MOD (NUMATS,NUMPROC)
                    sgdlsq = 0.0
      DO 321 L=0,NUMPROC-1
        IF(L.lt.NMOD)THEN
          NFIRST = (L*NDIV)+L+1
          LAST = ((L+1)*NDIV)+L+1
        ELSE IF(L.eq.NMOD)THEN
           NFIRST = (L*NDIV)+L+1
           LAST = ((L+1)*NDIV)+L
        ELSE IF(L.gt.NMOD)THEN
           NFIRST = (L*NDIV)+NMOD+1
           LAST = ((L+1)*NDIV)+NMOD
        ENDIF
        BFLENG=((LAST+1)-NFIRST)
C       RECALCULATE NFIRST FOR XO1(INTEGER*1 SIZE ARRAY)
                        nfirst = nfirst*4 - 3
             call receive(L,xo1(nfirst),43,INT2(bfleng*4),200,error)
c            write (*,*) 'xo1 receive',error
             call receive(L,xo2(nfirst),44,INT2(bfleng*4),
     1                    200,error)
c            write (*,*) 'xo2 receive',error
             call receive(L,xo3(nfirst),45,INT2(bfleng*4),
     1                    200,error)
c            write (*,*) 'xo3 receive',error
             call receive(L,temp1,46,4,200,error)
c            write (*,*) 'rms receive',error
             sgdlsq = sgdlsq + temp
 321  CONTINUE
      rmsd1=sqrt(sgdlsq/float(numats*3))
      write(*,18)rmsd1
      ITRCMP=ITRCMP+1
      IF(ITRCMP.LT.NUMITR) GO TO 999
C
C    -- IF NPRINT IS NOT EQUAL TO ZERO  --
C
      CALL POTE(ETOT,BFCERR,AFCERR,NFCERR,TFCERR)
      IF(NPRINT.EQ.0) WRITE(*,19) ETOT
      RETURN
      END
```

```
C
C    -- FORCE FIELD SETUP --
C
      BLOCK DATA FFSET

                  IMPLICIT NONE

      INTEGER I,J

                  INCLUDE 'CHMCM3.INC'
                  DATA H,Har,Hh,Csp3,Csp2,Car,Nsp,Namide,Ncation,Nar
     1,Osp3,Osp2,Oanion,F,Cl,Piii,Sii,Siii,Svi,Br,Iod,MET,MET1
     2,Mg2,Ca2,Ba2,Fe2,Fe3,Cu1,Cu2
     3 /1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22
     4 ,23,24,25,26,27,28,29,30/
                  DATA REFANG/3*10.,112.,122.,120.,108.,121.,108.
     1       ,120.,109.,4*10.,103.,96.,105.,109.47,2*10.,180.,8*10./
                  DATA EN/3*2.1,3*2.5,4*3.,3*3.5,4.,3.,2.1,3*2.5,2.8,2.5,2*0.
     1       ,1.2,1.0,0.9,1.8,2*1.9,2.0/
                  DATA ARTYPS/6,10/
                  DATA DBTYPS/5,8,12,18,19/

       DATA (REFLEN(I,1),I=1,MAXTYP)/3*0.746,1.091,1.07,1.089,
     1       1.01,0.99,1.03,0.99,0.97,2*10.0,0.92,1.27,1.43,
     2       3*1.32,1.41,1.61,1.056,8*10.0/

       DATA (REFLEN(I,2),I=1,MAXTYP)/0.0,2*0.746,1.091,1.07,
     1       1.089,1.01,0.99,1.03,0.99,0.97,2*10.0,0.92,1.27,1.43,
     2       3*1.32,1.41,1.61,1.056,8*10.0/

                  DATA (REFLEN(I,3),I=1,MAXTYP)/2*0.0,0.746,1.09,1.07,1.089,
     1       1.01,0.99,1.03,0.99,0.97,2*10.0,0.92,1.27,1.43,3*1.32,
     2       1.41,1.61,1.056,8*10.0/

                  DATA (REFLEN(I,4),I=1,MAXTYP)/3*0.0,1.541,2*1.52,1.47,
     1       3*1.48,1.43,2*10.0,1.381,1.767,1.815,1.81,1.84,1.76,
     2       1.937,2.14,1.46,8*10.0/

       DATA (REFLEN(I,5),I=1,MAXTYP)/4*0.0,1.335,1.47,2*1.32,
     1       2*1.48,1.36,1.21,1.26,1.33,1.72,1.77,1.75,1.71,1.61,
     2       1.89,2.09,1.43,8*10.0/

       DATA (REFLEN(I,6),I=1,MAXTYP)/5*0.0,1.395,1.43,1.43,1.45,
     1       1.35,1.38,2*10.0,1.3,1.7,1.76,1.74,1.76,1.74,1.85,2.05,
     2       1.43,8*10.0/

       DATA (REFLEN(I,7),I=1,MAXTYP)/6*0.0,1.45,1.35,2*10.0,1.36,
     1       2*10.0,1.36,1.75,1.67,3*1.62,2.14,2.34,1.158,8*10.0/

       DATA (REFLEN(I,8),I=1,MAXTYP)/7*0.0,1.27,2*10.0,1.39,1.23,
     1       1.25,1.36,1.79,1.62,1.61,2*1.53,2.14,2.34,1.2,
     2       8*10.0/

       DATA (REFLEN(I,9),I=1,MAXTYP)/8*0.0,2*10.0,1.2,1.06,10.0,
     1       1.36,1.75,1.67,3*1.62,2.14,2.34,9*10.0/

       DATA (REFLEN(I,10),I=1,MAXTYP)/9*0.0,10.0,1.34,4*10.0,1.57,
     1       5*10.0,1.5,8*10.0/

       DATA (REFLEN(I,11),I=1,MAXTYP)/10*0.0,1.48,10.0,1.26,1.418,
     1       1.7,1.5,1.43,2*1.62,1.85,2.05,1.3,8*10.0/

       DATA (REFLEN(I,12),I=1,MAXTYP)/11*0.0,4*10.0,1.47,10.0,
     1       1.49,1.43,2*10.0,1.2,8*10.0/

       DATA (REFLEN(I,13),I=1,MAXTYP)/12*0.0,18*10.0/

       DATA (REFLEN(I,14),I=1,MAXTYP)/13*0.0,1.417,1.63,1.535,
     1    3*1.585,1.76,1.96,1.26,8*10.0/

       DATA (REFLEN(I,15),I=1,MAXTYP)/14*0.0,1.988,2.1,3*2.08,
     1       2.14,2.32,1.64,8*10.0/

       DATA (REFLEN(I,16),I=1,MAXTYP)/15*0.0,2.2,1.86,2*2.0,
     1    2.13,2.48,1.7,8*10.0/

       DATA (REFLEN(I,17),I=1,MAXTYP)/16*0.0,2.05,2*2.12,2.27,
     1    2.73,1.7,8*10/

       DATA (REFLEN(I,18),I=1,MAXTYP)/17*0.0,2*2.12,2.27,2.73,
     1    1.56,8*10.0/

       DATA (REFLEN(I,19),I=1,MAXTYP)/18*0.0,2.07,2.27,2.73,
     1    9*10.0/

       DATA (REFLEN(I,20),I=1,MAXTYP)/19*0.0,2.29,2.4,1.79,
     1    8*10.0/
```

```
      DATA (REFLEN(I,21),I=1,MAXTYP)/20*0.0,2.84,1.99,8*10.0/
      DATA (REFLEN(I,22),I=1,MAXTYP)/21*0.0,1.2,8*10.0/
      DATA (REFLEN(I,23),I=1,MAXTYP)/22*0.0,8*10.0/
      DATA (REFLEN(I,24),I=1,MAXTYP)/23*0.0,7*10.0/
      DATA (REFLEN(I,25),I=1,MAXTYP)/24*0.0,6*10.0/
      DATA (REFLEN(I,26),I=1,MAXTYP)/25*0.0,5*10.0/
      DATA (REFLEN(I,27),I=1,MAXTYP)/26*0.0,4*10.0/
      DATA (REFLEN(I,28),I=1,MAXTYP)/27*0.0,3*10.0/
      DATA (REFLEN(I,29),I=1,MAXTYP)/28*0.0,2*10.0/
      DATA (REFLEN(I,30),I=1,MAXTYP)/29*0.0,10.0/
C
C     -- REFERENCE LENGTHS FOR CONJUGATED SINGLE BONDS --
C
      DATA CREFLN/
     1      8*10.
     2      ,0.,7*10.
     3      ,2*0.,6*10.
     4      ,3*0.,5*10.
     5      ,4*0.,1.48,1.47,10.,1.43
     6      ,5*0.,1.50,10.,1.42
     7      ,6*0.,2*10.
     8      ,7*0.,1.39/

      DATA (PERIOD(I,1),I=1,MAXTYP)/30*0.0/

      DATA (PERIOD(I,2),I=1,MAXTYP)/1*0.0,29*0.0/

      DATA (PERIOD(I,3),I=1,MAXTYP)/2*0.0,28*0.0/

      DATA (PERIOD(I,4),I=1,MAXTYP)/3*0.0,3.0,-3.0,-6.0,3.0,
     1     -3.0,3.0,0.0,3.0,4*0.0,2*3.0,-3.0,3.0,2*0.0,3.0,
     2     8*0.0/

      DATA (PERIOD(I,5),I=1,MAXTYP)/4*0.0,2*-2.0,-3.0,-2.0,-3.0,
     1      0.0,2*-2.0,3*0.0,-3.0,2*-2.0,-3.0,2*0.0,2.0,8*0.0/

      DATA (PERIOD(I,6),I=1,MAXTYP)/5*0.0,-2.0,-6.0,-2.0,-6.0,
     1      2*-2.0,4*0.0,2*-6.0,-2.0,-6.0,2*0.0,2.0,8*0.0/

      DATA (PERIOD(I,7),I=1,MAXTYP)/6*0.0,3.0,-3.0,3.0,0.0,3.0,
     1      4*0.0,2*3.0,-3.0,3.0,2*0.0,3.0,8*0.0/

      DATA (PERIOD(I,8),I=1,MAXTYP)/7*0.0,-2.0,-3.0,0.0,2*-2.0,
     1      3*0.0,-3.0,2*-2.0,-3.0,2*0.0,2.0,8*0.0/

      DATA (PERIOD(I,9),I=1,MAXTYP)/8*0.0,2*0.0,3.0,4*0.0,3.0,
     1     3.0,-3.0,3.0,11*0.0/

      DATA (PERIOD(I,10),I=1,MAXTYP)/9*0.0,6*0.0,-2.0,5*0.0,
     1     2.0,8*0.0/

      DATA (PERIOD(I,11),I=1,MAXTYP)/10*0.0,2.0,5*0.0,2.0,0.0,
     1     3.0,2*0.0,3.0,8*0.0/

      DATA (PERIOD(I,12),I=1,MAXTYP)/11*0.0,6*0.0,-2.0,3*0.0,
     1     2.0,8*0.0/

      DATA (PERIOD(I,13),I=1,MAXTYP)/12*0.0,18*0.0/

      DATA (PERIOD(I,14),I=1,MAXTYP)/13*0.0,17*0.0/

      DATA (PERIOD(I,15),I=1,MAXTYP)/14*0.0,16*0.0/

      DATA (PERIOD(I,16),I=1,MAXTYP)/15*0.0,2*3.0,-3.0,3.0,2*0.0,
     1     3.0,8*0.0/

      DATA (PERIOD(I,17),I=1,MAXTYP)/16*0.0,2.0,4*0.0,2.0,8*0.0/

      DATA (PERIOD(I,18),I=1,MAXTYP)/17*0.0,4*0.0,3.0,8*0.0/

      DATA (PERIOD(I,19),I=1,MAXTYP)/18*0.0,3.0,11*0.0/

      DATA (PERIOD(I,20),I=1,MAXTYP)/19*0.0,11*0.0/

      DATA (PERIOD(I,21),I=1,MAXTYP)/20*0.0,10*0.0/

      DATA (PERIOD(I,22),I=1,MAXTYP)/21*0.0,1.0,8*0.0/
```

```
      DATA (PERIOD(I,23),I=1,MAXTYP)/22*0.0,8*0.0/
      DATA (PERIOD(I,24),I=1,MAXTYP)/23*0.0,7*0.0/
      DATA (PERIOD(I,25),I=1,MAXTYP)/24*0.0,6*0.0/
      DATA (PERIOD(I,26),I=1,MAXTYP)/25*0.0,5*0.0/
      DATA (PERIOD(I,27),I=1,MAXTYP)/26*0.0,4*0.0/
      DATA (PERIOD(I,28),I=1,MAXTYP)/27*0.0,3*0.0/
      DATA (PERIOD(I,29),I=1,MAXTYP)/28*0.0,2*0.0/
      DATA (PERIOD(I,30),I=1,MAXTYP)/29*0.0,0.0/

C     ------BARRIER TO FREE ROTATION DATA----------

      DATA (BARIER(I,1),I=1,MAXTYP)/30*0.0/

      DATA (BARIER(I,2),I=1,MAXTYP)/1*0.0,29*0.0/

      DATA (BARIER(I,3),I=1,MAXTYP)/2*0.0,28*0.0/

      DATA (BARIER(I,4),I=1,MAXTYP)/3*0.0,0.133,0.182,0.008,0.114,
     1      0.083,0.0,0.008,0.1,4*0.0,0.163,0.195,0.245,12*0.0/

      DATA (BARIER(I,5),I=1,MAXTYP)/4*0.0,8.125,0.708,0.0,2.25,
     1      2*0.0,2.725,5*0.0,0.475,0.0,0.097,11*0.0/

      DATA (BARIER(I,6),I=1,MAXTYP)/5*0.0,5.0,0.825,3.65,0.0,5.0,
     1      0.821,5*0.0,0.19,13*0.0/

      DATA (BARIER(I,7),I=1,MAXTYP)/6*0.0,1.205,0.783,2*0.0,2.475,
     1      5*0.0,1.667,0.0,0.233,11*0.0/

      DATA (BARIER(I,8),I=1,MAXTYP)/7*0.0,40.0,2*0.0,5.0,5*0.0,4.3,
     1      0.0,0.467,11*0.0/

      DATA (BARIER(I,9),I=1,MAXTYP)/8*0.0,2*0.0,0.215,19*0.0/

      DATA (BARIER(I,10),I=1,MAXTYP)/9*0.0,21*0.0/

      DATA (BARIER(I,11),I=1,MAXTYP)/10*0.0,3.5,5*0.0,0.4,0.0,0.31,
     1      11*0.0/

      DATA (BARIER(I,12),I=1,MAXTYP)/11*0.0,19*0.0/

      DATA (BARIER(I,13),I=1,MAXTYP)/12*0.0,18*0.0/

      DATA (BARIER(I,14),I=1,MAXTYP)/13*0.0,17*0.0/

      DATA (BARIER(I,15),I=1,MAXTYP)/14*0.0,16*0.0/

      DATA (BARIER(I,16),I=1,MAXTYP)/15*0.0,0.513,14*0.0/

      DATA (BARIER(I,17),I=1,MAXTYP)/16*0.0,4.0,13*0.0/

      DATA (BARIER(I,18),I=1,MAXTYP)/17*0.0,13*0.0/

      DATA (BARIER(I,19),I=1,MAXTYP)/18*0.0,0.407,11*0.0/

      DATA (BARIER(I,20),I=1,MAXTYP)/19*0.0,11*0.0/

      DATA (BARIER(I,21),I=1,MAXTYP)/20*0.0,10*0.0/

      DATA (BARIER(I,22),I=1,MAXTYP)/21*0.0,9*0.0/

      DATA (BARIER(I,23),I=1,MAXTYP)/22*0.0,8*0.0/

      DATA (BARIER(I,24),I=1,MAXTYP)/23*0.0,7*0.0/

      DATA (BARIER(I,25),I=1,MAXTYP)/24*0.0,6*0.0/

      DATA (BARIER(I,26),I=1,MAXTYP)/25*0.0,5*0.0/

      DATA (BARIER(I,27),I=1,MAXTYP)/26*0.0,4*0.0/

      DATA (BARIER(I,28),I=1,MAXTYP)/27*0.0,3*0.0/

      DATA (BARIER(I,29),I=1,MAXTYP)/28*0.0,2*0.0/

      DATA (BARIER(I,30),I=1,MAXTYP)/29*0.0,0.0/

C-----------------------------------------------------
```

```
      DATA (A6(I,I),I=1,MAXTYP)/3*72.9,359.2,421.5,477.6,
     1      391.8,425.6,360.3,425.6,247.2,269.8,294.1,258.3,
     2      1638.4,2902.5,3*2524.9,3792.8,7579.4,492.8,0.0,
     3      2.2,27.8,178.8,5.3,1.4,23.3,7.9/

      DATA (B12(J,J),J=1,MAXTYP)/3*26572.0,460806.6,634601.7,
     1      814852.0,426454.3,503218.8,360564.3,503218.8,
     2      190913.6,227467.4,270337.6,166788.9,3355443.0,
     3      10530820.0,3*6929778.0,11987890.0,35904920.0,
     4      866709.0,0.0,17.7,2763.1,114230.3,142.1,9.6,2706.4,
     5      309.5/


C
C    -- BARRIER HEIGHTS AROUND CONJUGATED SINGLE BONDS --
C
      DATA CBARR /
     1  8*.0
     2 ,8*.0
     3 ,8*.0
     4 ,8*.0
     5 ,4*.0,.538,.568,.0,.463
     6 ,5*.0,.250,.0,.588
     7 ,8*.0
     8 ,7*.0,.625/

      END
C
C    -- SET UP AND SYMMETRIZE FORCE CONSTANT ARRAYS --
C
      SUBROUTINE MNINIT1

      IMPLICIT NONE

      INCLUDE 'CHMCM3.INC'
      INTEGER I,J

      REAL SKIJ,DEQ,A6IJ,B12IJ

C
C    -- I/O LOGICAL UNIT NUMBERS --
C
      DLUNIN=10
      DLNOUT=12
C
C    -- SET UP BOND STRETCH ARRAY --
C
      DO 1 I=H,MAXTYP
      DO 1 J=I,MAXTYP
        DEQ=REFLEN(J,I)
        IF(I.GT.Hh.AND.J.GT.Hh) THEN
          SKIJ=(1800./(DEQ*DEQ)) + (7.90/((DEQ-1.)*(DEQ-1.))) - 670./DEQ
        ELSE
          SKIJ = 395./(DEQ*DEQ)
        ENDIF
        STRCON(J,I)=SKIJ
        IF(I.EQ.J) GO TO 1
        STRCON(I,J)=SKIJ
1     CONTINUE
C
C    -- SET UP BOND STRETCH ARRAY FOR CONJUGATED SINGLE BONDS --
C
      DO 2 I=H,MXCNJ
      DO 2 J=I,MXCNJ
        DEQ=CREFLN(J,I)
        SKIJ=(1800./(DEQ*DEQ)) + (7.90/((DEQ-1.)*(DEQ-1.))) - 670./DEQ
        CSTCON(J,I)=SKIJ
        IF(I.EQ.J) GO TO 2
        CSTCON(I,J)=SKIJ
2     CONTINUE
C
C    -- FILL OFF DIAGONAL TERMS IN A6 AND B12 --
C
      DO 3 I=1,MAXTYP
      DO 3 J=I+1,MAXTYP
        A6IJ=SQRT(A6(I,I)*A6(J,J))
        A6(J,I)=A6IJ
        A6(I,J)=A6IJ
        B12IJ=SQRT(B12(I,I)*B12(J,J))
        B12(J,I)=B12IJ
        B12(I,J)=B12IJ
3     CONTINUE
C
C    -- MAKE REFLEN, PERIODICITY & BARRIER MATRICES SYMMETRIC --
C
      DO 4 I=2,MAXTYP
      DO 4 J=1,I-1
      IF(I.LE.MXCNJ.AND.J.LE.MXCNJ) THEN
```

```
            CREFLN(J,I)=CREFLN(I,J)
            CBARR(J,I)=CBARR(I,J)
          ENDIF
          REFLEN(J,I)=REFLEN(I,J)
          PERIOD(J,I)=PERIOD(I,J)
          BARIER(J,I)=BARIER(I,J)
4         CONTINUE
C
C     -- BARRIER PERIODICITY FOR CONJUGATED SINGLE BONDS --
C
          CPRIOD=-2.0
C
          NUMATS=0
          NMOLS=0
C
C     -- DELTA USED TO CALCULATE NUMERICAL DERIVATIVES
C
          SHIFTX=1.0E-03
C
C     -- BOND LENGTH TOLERANCE --
C
          DISTOL=0.1
C
C     -- CONVERSION FACTORS --
C
          PI=3.1415926
          RAD1=PI/180.0
          RAD2=RAD1*RAD1
          RADI=1.0/RAD1

          RETURN
          END
```

```
C
C    -- ASSSIGNS PSEUDO BOND ORDERS --
C
      SUBROUTINE ASBOML

                  IMPLICIT NONE

      INTEGER I,J,IKAC,L,K
      REAL DISFIL,VL
      BYTE IATN,IKACTN

      INCLUDE 'CHMCM3.INC'
      DISFIL=0.5*DISTOL
      DO 10 J=1,MXCN
      DO 10 I=1,NUMATS
        BONDML(I,J)=0
        IF(ATMCON(I,J).NE.0) BONDML(I,J)=10
10    CONTINUE
      DO 100 I=1,NUMATS
      IATN=ATYNUM(I)
      DO 50 J=1,NARTYP
      IF(IATN.EQ.ARTYPS(J)) THEN
        DO 40 K=1,MXCN
        IKAC=ATMCON(I,K)
        IKACTN=ATYNUM(IKAC)
        DO 20 L=1,NARTYP
        IF(IKACTN.EQ.ARTYPS(L)) THEN
          VL=SQRT((XO(I,1)-XO(IKAC,1))**2+(XO(I,2)-XO(IKAC,2))**2
     1           +(XO(I,3)-XO(IKAC,3))**2)
          BONDML(I,K)=15
          IF(VL.GE.(REFLEN(IATN,IKACTN)+DISFIL)) BONDML(I,K)=11
          GO TO 40
        ENDIF
20      CONTINUE
        DO 30 L=1,NDBTYP
        IF(IKACTN.EQ.DBTYPS(L)) THEN
          BONDML(I,K)=11
          GO TO 40
        ENDIF
30      CONTINUE
40      CONTINUE
        GO TO 100
      ENDIF
50    CONTINUE
      DO 90 J=1,NDBTYP
      IF(IATN.EQ.DBTYPS(J)) THEN
        DO 80 K=1,MXCN
        IKAC=ATMCON(I,K)
        IKACTN=ATYNUM(IKAC)
        DO 60 L=1,NDBTYP
        IF(IKACTN.EQ.DBTYPS(L)) THEN
          VL=SQRT((XO(I,1)-XO(IKAC,1))**2+(XO(I,2)-XO(IKAC,2))**2
     1           +(XO(I,3)-XO(IKAC,3))**2)
          IF(IATN.EQ.Csp2.AND.IKACTN.EQ.Namide.
     1         OR.IATN.EQ.Namide.AND.IKACTN.EQ.Csp2) THEN
            BONDML(I,K)=15
            IF(VL.LE.(REFLEN(IATN,IKACTN)-DISFIL)) BONDML(I,K)=20
            IF(VL.GE.(REFLEN(IATN,IKACTN)+DISFIL)) BONDML(I,K)=11
          ELSE
            BONDML(I,K)=20
            IF(VL.GE.(REFLEN(IATN,IKACTN)+DISFIL)) BONDML(I,K)=11
          ENDIF
          GO TO 80
        ENDIF
60      CONTINUE
        DO 70 L=1,NARTYP
        IF(IKACTN.EQ.ARTYPS(L)) THEN
          BONDML(I,K)=11
          GO TO 80
        ENDIF
70      CONTINUE
80      CONTINUE
        GO TO 100
      ENDIF
90    CONTINUE
100   CONTINUE
      RETURN
      END

C
C    -- DECODE OUT OF PLANE BENDING --
C
      SUBROUTINE GETOPB

                  IMPLICIT NONE

      INCLUDE 'CHMCM3.INC'
```

```
      INTEGER IOOPBA,IOPBS,L,MF,NCON,M,IACLN,N

      REAL SOPBKS

      DIMENSION IOOPBA(3),SOPBKS(3),IOPBS(MXCN)

      DATA IOOPBA/   5   ,   6   ,   8  /
      DATA SOPBKS/1.2E-3,1.2E-3,0.2E-3/

      NO=0
      DO 115 L=1,NUMATS
      IF(NUMMFX.NE.0) THEN
        DO 100 MF=1,NUMMFX
          IF(MOLNUM(L).EQ.KMOL(MF)) GO TO 115
100     CONTINUE
      ENDIF
      NCON=0
      DO 110 M=1,3
      IF(ATYNUM(L).NE.IOOPBA(M)) GO TO 110
      DO 105 N=1,MXCN
      IACLN=ATMCON(L,N)
      IF(IACLN.NE.0) THEN
        NCON=NCON+1
        IOPBS(NCON)=IACLN
      ENDIF
105   CONTINUE
      IF(NCON.NE.3) GO TO 115
      NO=NO+1
      IOPB3(NO)=L
      IOPB1(NO)=IOPBS(1)
      IOPB2(NO)=IOPBS(2)
      IOPB4(NO)=IOPBS(3)
      OPBK(NO)=SOPBKS(M)
      GO TO 115
110   CONTINUE
115   CONTINUE
      RETURN
      END
```

C

```
C     -- GET CONTROL PARAMETERS FROM CONSOLE--
C
        SUBROUTINE GETCOP
                        IMPLICIT NONE
        INCLUDE 'CHMCM3.INC'
        INTEGER NFXATM,IKON,JKON,KKON,LKON,I,J
        REAL ATSEV,FLEN,DSEV,FANG,ANSEV,FTOR,TSEV

1       FORMAT(I5)
2       FORMAT(E12.5)
3       FORMAT(L1)
        OPEN(2,FILE='MINDAT.CWO')
C
C     -- GET NUMBER OF ITERATIONS
C
          READ(2,1)NUMITR


C
C     -- GET VAN DER WAALS CUTOFF DISTANCE --
C
          READ(2,2)DXXM
          IF(DXXM.LT.2.0) DXXM=2.0
C
C     -- GET ENERGY THRESHOLD FOR PRINTING --
C
          READ(2,2) ETHRSH
          IF(ETHRSH.EQ.0.0) ETHRSH=-10.0
C
C     -- GET MAXIMUM ALLOWED SHIFT --
C
          READ(2,2) SHFTMX
          IF(SHFTMX.EQ.0.0) SHFTMX=0.5
C
C     -- SELECT LONG, ABBREVIATED OR SHORT PRINTED OUTPUT --
C
          READ(2,1)NPRINT
C
C     -- CHOOSE SECOND DERIVATIVES CALCULATED EVERY ITERATION OR NOT --
C
          NDERIV=1
C
C     -- GET CONSTRAINTS, IF ANY --
C
          NUMLFX=0
          NUMAFX=0
          NUMTFX=0
          NUMMFX=0
          READ(2,3)COMMIN
          IF(COMMIN) THEN
            DO 100 I=1,4
              DO 100 J=1,NUMATS
                ATCONS(J,I)=.FALSE.
100       CONTINUE
C
C     -- CODE TO FIX ATOMIC POSITIONS --
C
          READ(2,1)NFXATM
          IF(NFXATM.EQ.0) GOTO 400
          DO 300 I = 1,NFXATM
            READ(2,1)IKON
            READ(2,2)ATSEV
            ATCONS(IKON,1)=.TRUE.
            FATXYZ(IKON,1)=XO(IKON,1)
            FATXYZ(IKON,2)=XO(IKON,2)
            FATXYZ(IKON,3)=XO(IKON,3)
            FATSEV(IKON)=ATSEV*ATSFAC
300       CONTINUE
C
C     -- CODE TO FIX LENGTHS --
C
400       READ(2,1)NUMLFX
          IF(NUMLFX.EQ.0) GOTO 500
          DO 410 I = 1,NUMLFX
            READ(2,1)IKON
            READ(2,1)JKON
            READ(2,2)FLEN
            READ(2,2)DSEV
            ATCONS(IKON,2)=.TRUE.
            ATCONS(JKON,2)=.TRUE.
            KLATM1(I)=IKON
            KLATM2(I)=JKON
            FIXLEN(I)=FLEN
```

```
            FLNSEV(I)=DSEV
410       CONTINUE
C
C     -- CODE TO FIX ANGLES --
C
500       READ(2,1)NUMAFX
          IF(NUMAFX.EQ.0) GOTO 600
          DO 510 I = 1,NUMAFX
            READ(2,1)IKON
            READ(2,1)JKON
            READ(2,1)KKON
            READ(2,2)FANG
            READ(2,2)ANSEV
            ATCONS(IKON,3)=.TRUE.
            ATCONS(JKON,3)=.TRUE.
            ATCONS(KKON,3)=.TRUE.
            KAATM1(I)=IKON
            KAATM2(I)=JKON
            KAATM3(I)=KKON
            FIXANG(I)=FANG
            FANSEV(I)=ANSEV*ANSFAC
510       CONTINUE
C
C     -- CODE TO FIX TORSION ANGLES --
C
600       READ(2,1)NUMTFX
          IF(NUMTFX.EQ.0) GOTO 700
          DO 610 I = 1,NUMTFX
            READ(2,1)IKON
            READ(2,1)JKON
            READ(2,1)KKON
            READ(2,1)LKON
            READ(2,2)FTOR
            READ(2,2)TSEV
            ATCONS(IKON,4)=.TRUE.
            ATCONS(JKON,4)=.TRUE.
            ATCONS(KKON,4)=.TRUE.
            ATCONS(LKON,4)=.TRUE.
            KTATM1(I)=IKON
            KTATM2(I)=JKON
            KTATM3(I)=KKON
            KTATM4(I)=LKON
            FIXTOR(I)=FTOR
            FTOSEV(I)=TSEV*TOSFAC
610       CONTINUE
C
C     -- CODE TO FIX MOLECULES --
C
700       READ(2,1)NUMMFX
          IF(NUMMFX.EQ.0) GOTO 800
          DO 710 I = 1,NUMMFX
            READ(2,1)IKON
            KMOL(NUMMFX)=MOLNUM(IKON)
710       CONTINUE
C
C     -- EXIT --
C
800     ENDIF
        CLOSE(2)

        RETURN
        END
```

```
C     -- SUBROUTINE POTE --
C
C
C     -- THIS SUBROUTINE CALCULATES AND PRINTS OUT THE ENERGY OF THE MOLEC
C     -- AND SETS UP 'TABLES' OF BONDS ETC. FOR USE IN ENERGL IF NPRINT EQ
C     -- ZERO, ALL INTERACTIONS BETWEEN ALL ATOMS ARE PRINTED.  IF NPRINT
C     -- NOT ZERO THEN ONLY THE TOTAL ENERGIES FOR EACH TYPE OF INTERACTIO
C     -- ARE PRINTED. XFCERR IS SET TRUE IF ANY FORCE FIELD ERRORS ARE FOU
C
      SUBROUTINE POTE(ETOT,BFCERR,AFCERR,NFCERR,TFCERR)

                  IMPLICIT NONE

       INTEGER I,J,IBOND,NK,ITI,NVANG,JB,MF,JA,
     1         JC,IT1,IT2,IT3,JBCN,NI,
     2         INDT,MTII,MTJI,MTKI,
     3         MTLI,LL,IT4,ITJ,K,
     4         JAPLS1,INDA,
     5         NNT,ILI,L,IND,MT


       REAL DXXM2,SIGEB,SIGEV,SIGEA,SIGET,SIGEO,SIGEQ,DIR1,
     1      DIR2,DIR3,DOIST,DIST,RLITIJ,SCITIJ,EB,XJB1,XJB2,
     2      XJB3,DC11,DC12,DC21,DC22,DC31,DC32,RM1,RM2,R12,
     3      COSA,SUBST,RLIT12,RLIT23,BKT,THT,THSX,
     4      DELTH2,DELTH3,DELTH5,EA,DIST2,RDIST2,RDIST4,
     5      RDIST6,RDIST12,EV,EQ,XJC1,XJC2,XJC3,AA11,AA12,
     6      AA13,AA21,AA22,AA23,AA31,AA32,AA33,V11,V21,V12,
     7      V22,V13,V23,R1,R2,COSW,WASIGN,WA,XFD,TA,SGN,
     8      FOLD,ET,OPBSGN,WAOPB,EO,ETOT,A,DISTI,RDIS12


       INCLUDE 'CHMCM3.INC'
       LOGICAL PRNTOG,BFCERR,AFCERR,NFCERR,TFCERR

1      FORMAT(30(/),22X,'MOLECULAR POTENTIAL ENERGY (KCAL)')
2      FORMAT(////,24X,'INTRAMOLECULAR BONDED DISTANCES',//)
3      FORMAT(' ATOM A-ATOM B  DISTANCE  BOND ENERGY    ATOM A-ATOM B  DI
     1STANCE  BOND ENERGY',/)
4      FORMAT(4X,A6,1X,A6,2X,F8.3,2X,F11.4)
5      FORMAT(1X,A6,1X,A6,2X,F8.3,2X,F11.4,$)
6      FORMAT(////,33X,'BOND ANGLES',//)
7      FORMAT(' ATOM A-ATOM B-ATOM C   ANGLE  ENERGY    ATOM A-ATOM B-ATO
     1M C   ANGLE  ENERGY',/)
8      FORMAT(4X,A6,1X,A6,1X,A6,F8.2,F8.4)
9      FORMAT(1X,A6,1X,A6,1X,A6,F8.2,F8.4,$)
10     FORMAT(////,23X,'INTRAMOLECULAR NON-BONDED DISTANCES',//)
11     FORMAT(29X,'UP TO ',F6.2,' ANGSTROMS',//)
12     FORMAT(' ATOMA..ATOMB DISTANCE NON-BOND COULOMB  ATOMA..ATOMB  DIS
     1TANCE NON-BOND COULOMB',/)
13     FORMAT(2X,A6,1X,A6,3X,F6.3,1X,F8.4,1X,F7.3)
14     FORMAT(1X,2A6,3X,F6.3,1X,F8.4,1X,F7.3,$)
15     FORMAT(////,33X,'TORSION ANGLES',//)
16     FORMAT(5X,'ATOM A  ATOM B  ATOM C  ATOM D    TORSION ANGLE    TORS
     1ION ENERGY',/)
17     FORMAT(5X,4(A6,2X),2X,F13.2,4X,F14.4)
18     FORMAT(12X,'*****************OUT-OF-PLANE BENDING*************'
     1)
19     FORMAT(////,
     1' TOTAL E(BONDED) = ',                    F10.4,' K.CAL PER MOLE',//,
     2' TOTAL E(VAN DER WAALS) = ',             F10.4,' K.CAL PER MOLE',//,
     3' TOTAL E(ANGLES) = ',                    F10.4,' K.CAL PER MOLE',//,
     4' TOTAL E(TORSION) = ',                   F10.4,' K.CAL PER MOLE',//,
     5' TOTAL E(OUT-OF-PLANE BENDING) = ', F10.4,' K.CAL PER MOLE',//,
     6' TOTAL E(COULOMB) = ',                   F10.4,' K.CAL PER MOLE',//)
20     FORMAT(//,' TOTAL POTENTIAL ENERGY = ',F10.4,' K.CAL PER MOLE',//)
C
       IF(NPRINT.GT.1) THEN
         WRITE(*,1)
       ENDIF
       DXXM2=DXXM*DXXM
C
C     -- SIGEB ETC. ARE THE TOTAL ENERGIES FOR EACH TYPE OF INTERACTION --
C
       SIGEB=0.0
       SIGEV=0.0
       SIGEA=0.0
       SIGET=0.0
       SIGEO=0.0
       SIGEQ=0.0
C
C     -- INITIALISE MATRICES --
C
       DO 400 I=1,NUMATS
       DO 400 J=1,NUMATS
         NBMAT(J,I)=1
400    CONTINUE
```

```
      DO 401 I=1,NUMATS
      DO 401 J=1,18
        NAMAT(J,I)=0
401   CONTINUE
      DO 402 I=1,NUMATS
      DO 402 J=1,50
        NTMAT(J,I)=0
402   CONTINUE
C
      IF(NPRINT.GT.1) THEN
        WRITE(*,2)
        WRITE(*,3)
      ENDIF
C
C   -- PUT BONDS INTO NBMAT --
C
      DO 405 I=1,NUMATS
      DO 405 J=1,MXCN
        IBOND=ATMCON(I,J)
        IF(IBOND.EQ.0) GO TO 405
        NBMAT(IBOND,I)=2
        NBMAT(I,IBOND)=2
405   CONTINUE
C
      NK=NUMATS-1
      PRNTOG=.TRUE.
C
      DO 412 I=1,NK
      NI=I+1
      DO 411 J=NI,NUMATS
C
C   -- SPEED UP IF THERE ARE ANY FIXED MOLECULES --
C
      IF(NUMMFX.NE.0) THEN
        DO 406 MF=1,NUMMFX
          IF(MOLNUM(I).EQ.KMOL(MF).AND.MOLNUM(J).EQ.KMOL(MF)) GO TO 410
406     CONTINUE
      ENDIF
C
C   -- CALCULATE DISTANCE BETWEEN BONDED ATOMS --
C
      IF(NBMAT(J,I).NE.2) GO TO 409
      DIR1=XO(I,1)-XO(J,1)
      DIR2=XO(I,2)-XO(J,2)
      DIR3=XO(I,3)-XO(J,3)
      DIST=SQRT(DIR1*DIR1+DIR2*DIR2+DIR3*DIR3)
      ITI=ATYNUM(I)
      ITJ=ATYNUM(J)
      RLITIJ=REFLEN(ITI,ITJ)
      SCITIJ=STRCON(ITI,ITJ)
      IF(ITI.LE.Namide.AND.ITJ.LE.Namide) THEN
        IF(CREFLN(ITI,ITJ).NE.10.) THEN
          DO 407 K=1,MXCN
            IF(ATMCON(I,K).EQ.J) GO TO 408
407       CONTINUE
408       IF(BONDML(I,K).EQ.11) THEN
            RLITIJ=CREFLN(ITI,ITJ)
            SCITIJ=CSTCON(ITI,ITJ)
          ENDIF
        ENDIF
      ENDIF
      IF(RLITIJ.EQ.10.) THEN
        RLITIJ=DIST
        BFCERR=.TRUE.
      ENDIF
      DOIST=RLITIJ-DIST
C
C   -- CALCULATE BOND ENERGY (EB) --
C
      EB=SCITIJ*DOIST*DOIST
      SIGEB=SIGEB+EB
      IF(NPRINT.GT.1.AND.EB.GE.ETHRSH) THEN
        IF(.NOT.PRNTOG) WRITE(*,4) ATMNAM(I),ATMNAM(J),DIST,EB
        IF(PRNTOG)      WRITE(*,5) ATMNAM(I),ATMNAM(J),DIST,EB
        PRNTOG=.NOT.PRNTOG
      ENDIF
      GO TO 411
409   DIR1=XO(I,1)-XO(J,1)
      IF(ABS(DIR1).GT.DXXM) GO TO 410
      DIR2=XO(I,2)-XO(J,2)
      IF(ABS(DIR2).GT.DXXM) GO TO 410
      DIR3=XO(I,3)-XO(J,3)
      IF(ABS(DIR3).GT.DXXM) GO TO 410
      DIST2=DIR1*DIR1+DIR2*DIR2+DIR3*DIR3
      IF(DIST2.GT.DXXM2) GO TO 410
      NBMAT(J,I)=4
      NBMAT(I,J)=4
      GO TO 411
```

```
410     NBMAT(I,J)=5
        NBMAT(J,I)=5
411     CONTINUE
412     CONTINUE
        IF(NPRINT.GT.1) THEN
          WRITE(*,6)
          WRITE(*,7)
        ENDIF
C
C   -- CALCULATE ANGLE ENERGY AND SET UP MATRIX NAMAT --
C
        NVANG=0
        PRNTOG=.TRUE.
        DO 429 JB=1,NUMATS
        IF(NUMMFX.NE.0) THEN
          DO 413 MF=1,NUMMFX
            IF(MOLNUM(JB).EQ.KMOL(MF)) GO TO 429
413       CONTINUE
        ENDIF
        DO 428 JA=1,NK
        IF(NUMMFX.NE.0) THEN
          DO 414 MF=1,NUMMFX
            IF(MOLNUM(JA).EQ.KMOL(MF)) GO TO 428
414       CONTINUE
        ENDIF
C
C     -- SORT OUT WHICH SETS OF THREE ATOMS FORM ANGLES --
C
        IF(JA.EQ.JB) GO TO 428
        IF(NBMAT(JA,JB).NE.2) GO TO 428
        JAPLS1=JA+1
        DO 427 JC=JAPLS1,NUMATS
        IF(NUMMFX.NE.0) THEN
          DO 415 MF=1,NUMMFX
            IF(MOLNUM(JC).EQ.KMOL(MF)) GO TO 427
415       CONTINUE
        ENDIF
        IF(JB.EQ.JC) GO TO 427
        IF(NBMAT(JC,JB).NE.2) GO TO 427
        NVANG=NVANG+1
        MAI(NVANG)=JA
        MAJ(NVANG)=JB
        MAK(NVANG)=JC
C
C   -- SET NBMAT ENTRY TO 3 FOR ALL 1,3 PAIRS OF ATOMS --
C
          NBMAT(JA,JC)=3
          NBMAT(JC,JA)=3
C
C    -- CALCULATE ANGLE JA-JB-JC --
C
          XJB1=XO(JB,1)
          XJB2=XO(JB,2)
          XJB3=XO(JB,3)
          DC11=XO(JA,1)-XJB1
          DC12=XO(JC,1)-XJB1
          DC21=XO(JA,2)-XJB2
          DC22=XO(JC,2)-XJB2
          DC31=XO(JA,3)-XJB3
          DC32=XO(JC,3)-XJB3
          RM1=DC11*DC11+DC21*DC21+DC31*DC31
          RM2=DC12*DC12+DC22*DC22+DC32*DC32
          R12=DC11*DC12+DC21*DC22+DC31*DC32
          RM1=SQRT(RM1)+0.000001
          RM2=SQRT(RM2)+0.000001
          COSA=R12/(RM1*RM2)
          COSA=SIGN(AMIN1(ABS(COSA),1.0E+00),COSA)
          A=ACOS(COSA)*RADI
C
C    -- SELECT CORRECT FORCE FIELD RECORD --
C
          IT1=ATYNUM(JA)
          IT2=ATYNUM(JB)
          IT3=ATYNUM(JC)
          IF(REFANG(IT2).EQ.10.) THEN
            BKS(NVANG)=0.0
            BKAS(NVANG)=0.0
            THS(NVANG)=A
            AFCERR=.TRUE.
          ELSE
            SUBST=0.
            DO 416 K=1,MXCN
            JBCN=ATMCON(JB,K)
            IF(JBCN.NE.0) THEN
              IF(ATYNUM(JBCN).GT.Hh) SUBST=SUBST+1.
            ENDIF
416         CONTINUE
            RLIT12=REFLEN(IT1,IT2)
```

```
      RLIT23=REFLEN(IT2,IT3)
      IF(IT1.LE.Namide.AND.IT2.LE.Namide) THEN
        IF(CREFLN(IT1,IT2).NE.10.) THEN
          DO 417 K=1,MXCN
            IF(ATMCON(JA,K).EQ.JB) GO TO 418
417       CONTINUE
418       IF(BONDML(JA,K).EQ.11) RLIT12=CREFLN(IT1,IT2)
        ENDIF
      ENDIF
      IF(IT2.LE.Namide.AND.IT3.LE.Namide) THEN
        IF(CREFLN(IT2,IT3).NE.10.) THEN
          DO 419 K=1,MXCN
            IF(ATMCON(JB,K).EQ.JC) GO TO 420
419       CONTINUE
420       IF(BONDML(JB,K).EQ.11) RLIT23=CREFLN(IT2,IT3)
        ENDIF
      ENDIF
      BKT=0.001388
     1      *(15.+2.33*(ABS(EN(IT1)-EN(IT2))+ABS(EN(IT2)-EN(IT3))))
     2      /(RLIT12*RLIT23)
        THT=REFANG(IT2)
        IF((IT1.LE.Hh.AND.IT3.GT.Hh).OR.(IT1.GT.Hh.AND.IT3.LE.Hh)) THEN
          BKT=0.45*BKT
          THT=0.98*THT
        ENDIF
        IF(IT1.LE.Hh.AND.IT3.LE.Hh) THEN
          BKT=0.20*BKT
          THT=0.95*THT
        ENDIF
        BKS(NVANG)=BKT
        BKAS(NVANG)=0.0096
        THS(NVANG)=THT
      ENDIF
      THSX=THS(NVANG)-A
      DELTH2=THSX*THSX
      DELTH3=ABS(DELTH2*THSX)
      DELTH5=DELTH3*DELTH2
C
C     -- CALCULATE ANGLE ENERGY --
C
      EA=BKS(NVANG)*(DELTH2-BKAS(NVANG)*(DELTH3-(0.0004*DELTH5)))
      SIGEA=SIGEA+EA
C
C     -- SET UP NAMAT, WHICH IS USED AS FOLLOWS --
C     -- TO SEE WHICH ANGLES THE JTH ATOM IS INVOLVED IN, --
C     -- READ NAMAT(J,1), (J,2) ETC TILL A ZERO ENTRY IS FOUND. --
C     -- IF NAMAT(J,1) = 5, THEN ATOM J IS PART OF THE ANGLE NVANG=5 --
C     --(IE THE ANGLE WITH ATOMS MAI(5), MAJ(5), MAK(5)) --
C
      DO 421 INDA=1,18
        IF(NAMAT(INDA,JA).NE.0) GO TO 421
        NAMAT(INDA,JA)=NVANG
        GO TO 422
421   CONTINUE
422   DO 423 INDA=1,18
        IF(NAMAT(INDA,JB).NE.0) GO TO 423
        NAMAT(INDA,JB)=NVANG
        GO TO 424
423   CONTINUE
424   DO 425 INDA=1,18
        IF(NAMAT(INDA,JC).NE.0) GO TO 425
        NAMAT(INDA,JC)=NVANG
        GO TO 426
425   CONTINUE
C
426   IF(NPRINT.GT.1.AND.EA.GE.ETHRSH) THEN
        IF(.NOT.PRNTOG) WRITE(*,8) ATMNAM(JA),ATMNAM(JB),
     1    ATMNAM(JC),A,EA
        IF(PRNTOG)      WRITE(*,9) ATMNAM(JA),ATMNAM(JB),
     1    ATMNAM(JC),A,EA
        PRNTOG=.NOT.PRNTOG
      ENDIF
427   CONTINUE
428   CONTINUE
429   CONTINUE
      IF(NPRINT.GT.1) THEN
        WRITE(*,10)
        IF(DXXM.NE.25.0) WRITE(*,11) DXXM
        WRITE(*,12)
      ENDIF
C
C     -- CALCULATE NON-BONDED INTERACTIONS USING NBMAT --
C     -- LOOK AT ENTRY NBMAT(I,J) - IF NOT = 4 (IE I AND J ARE BONDED, 1,3
C     -- , OR SEPARATED BY MORE THAN DXXM) GO ON TO NEXT PAIR, OTHERWISE -
C     -- CALCULATE VAN DER WAALS ENERGY --
C
      PRNTOG=.TRUE.
      DO 430 I=1,NK
```

```
      NI=I+1
      DO 430 J=NI,NUMATS
      IF(NBMAT(J,I).NE.4) GO TO 430
      DIR1=XO(I,1)-XO(J,1)
      DIR2=XO(I,2)-XO(J,2)
      DIR3=XO(I,3)-XO(J,3)
      DIST2=DIR1*DIR1+DIR2*DIR2+DIR3*DIR3
      DISTI=SQRT(DIST2)
      ITI=ATYNUM(I)
      ITJ=ATYNUM(J)
      IF(A6(ITI,ITJ).EQ.0.) NFCERR=.TRUE.
      RDIST2=1.0/DIST2
      RDIST4=RDIST2*RDIST2
      RDIST6=RDIST2*RDIST4
      RDIS12=RDIST6*RDIST6
C
C     -- CALCULATE VAN DER WAALS ENERGY (EV) --
C
      EV=B12(ITI,ITJ)*RDIS12 - A6(ITI,ITJ)*RDIST6
      SIGEV=SIGEV+EV
C
C     -- CALCULATE COULOMBIC ENERGY (EQ) --
C
      EQ=332.17*CHARGE(I)*CHARGE(J)*RDIST2
      SIGEQ=SIGEQ+EQ
      IF(NPRINT.GT.1.AND.(EV.GE.ETHRSH
     1                .OR.EQ.GE.ETHRSH)) THEN
        IF(.NOT.PRNTOG) WRITE(*,13) ATMNAM(I),ATMNAM(J),DISTI,EV,
     1    EQ
        IF(PRNTOG)      WRITE(*,14) ATMNAM(I),ATMNAM(J),DISTI,EV,
     1    EQ
        PRNTOG=.NOT.PRNTOG
      ENDIF
430   CONTINUE
      IF(NPRINT.GT.1 ) THEN
        WRITE(*,15)
        WRITE(*,16)
      ENDIF
C
C     -- CALCULATE TORSIONAL ENERGY AND SET UP NTMAT IN SAME WAY AS NAMAT
C
      NNT=0
C
C     -- SORT OUT WHICH SETS OF ATOMS FORM A TORSION ANGLE --
C
      DO 445 I=1,NK
      IF(NUMMFX.NE.0) THEN
        DO 431 MF=1,NUMMFX
          IF(MOLNUM(I).EQ.KMOL(MF)) GO TO 445
431     CONTINUE
      ENDIF
      ILI=I+1
      DO 444 J=1,NUMATS
      IF(NUMMFX.NE.0) THEN
        DO 432 MF=1,NUMMFX
          IF(MOLNUM(J).EQ.KMOL(MF)) GO TO 444
432     CONTINUE
      ENDIF
      IF(NBMAT(J,I).NE.2) GO TO 444
      DO 443 K=1,NUMATS
      IF(NUMMFX.NE.0) THEN
        DO 433 MF=1,NUMMFX
          IF(MOLNUM(K).EQ.KMOL(MF)) GO TO 443
433     CONTINUE
      ENDIF
      IF(K.EQ.I) GO TO 443
      IF(NBMAT(J,K).NE.2) GO TO 443
      DO 442 L=ILI,NUMATS
      IF(NUMMFX.NE.0) THEN
        DO 434 MF=1,NUMMFX
          IF(MOLNUM(L).EQ.KMOL(MF)) GO TO 442
434     CONTINUE
      ENDIF
      IF(L.EQ.J) GO TO 442
      IF(NBMAT(L,K).NE.2) GO TO 442
      NNT=NNT+1
      MTI(NNT)=I
      MTJ(NNT)=J
      MTK(NNT)=K
      MTL(NNT)=L
C
C     -- SET UP NTMAT --
C
      DO 435 INDT=1,50
        IF(NTMAT(INDT,I).NE.0) GO TO 435
        NTMAT(INDT,I)=NNT
        GO TO 436
435   CONTINUE
```

```
436     DO 437 INDT=1,50
          IF(NTMAT(INDT,J).NE.0) GO TO 437
          NTMAT(INDT,J)=NNT
          GO TO 438
437     CONTINUE
438     DO 439 INDT=1,50
          IF(NTMAT(INDT,K).NE.0) GO TO 439
          NTMAT(INDT,K)=NNT
          GO TO 440
439     CONTINUE
440     DO 441 INDT=1,50
          IF(NTMAT(INDT,L).NE.0) GO TO 441
          NTMAT(INDT,L)=NNT
          GO TO 442
441     CONTINUE
442     CONTINUE
443     CONTINUE
444     CONTINUE
445     CONTINUE
C
        DO 448 I=1,NNT
        MTII=MTI(I)
        MTJI=MTJ(I)
        MTKI=MTK(I)
        MTLI=MTL(I)
C
C     -- CALCULATE TORSION ANGLE --
C
        XJB1=XO(MTJI,1)
        XJB2=XO(MTJI,2)
        XJB3=XO(MTJI,3)
        XJC1=XO(MTKI,1)
        XJC2=XO(MTKI,2)
        XJC3=XO(MTKI,3)
        AA11=XO(MTII,1)-XJB1
        AA12=XJC1-XJB1
        AA13=XJC1-XO(MTLI,1)
        AA21=XO(MTII,2)-XJB2
        AA22=XJC2-XJB2
        AA23=XJC2-XO(MTLI,2)
        AA31=XO(MTII,3)-XJB3
        AA32=XJC3-XJB3
        AA33=XJC3-XO(MTLI,3)
        V11=AA21*AA32-AA31*AA22
        V21=AA22*AA33-AA32*AA23
        V12=AA31*AA12-AA11*AA32
        V22=AA32*AA13-AA12*AA33
        V13=AA11*AA22-AA21*AA12
        V23=AA12*AA23-AA22*AA13
        R1=SQRT(V11*V11+V12*V12+V13*V13)+0.000001
        R2=SQRT(V21*V21+V22*V22+V23*V23)+0.000001
        COSW=(V11/R1)*(V21/R2)+(V12/R1)*(V22/R2)+(V13/R1)*(V23/R2)
        COSW=SIGN(AMIN1(ABS(COSW),1.0E+00),COSW)
        WA=ACOS(COSW)*RADI
C
C     -- CALCULATE CORRECT SIGN FOR TORSION ANGLE --
C
        WASIGN=AA11*(AA22*AA33-AA23*AA32)-AA21*(AA12*AA33-AA13*AA32)
       1       +AA31*(AA12*AA23-AA13*AA22)
        WA=SIGN(WA,WASIGN)
        LL=0
        IT1=ATYNUM(MTII)
        IT2=ATYNUM(MTJI)
        IT3=ATYNUM(MTKI)
        IT4=ATYNUM(MTLI)
C
C     -- SELECT CORRECT FORCE FIELD RECORD --
C
        FDS(I)=PERIOD(IT2,IT3)
        IF(FDS(I).EQ.-3.) THEN
          IF(PERIOD(IT1,IT2).NE.-2.0.AND.PERIOD(IT3,IT4).NE.-2.0)
      1   FDS(I)=3.
        ENDIF
        VOS(I)=BARIER(IT2,IT3)
        VO1S(I)=0.0
        IF(IT2.LE.Namide.AND.IT3.LE.Namide) THEN
          IF(CBARR(IT2,IT3).NE.0.) THEN
            DO 446 K=1,MXCN
              IF(ATMCON(MTJI,K).EQ.MTKI) GO TO 447
446         CONTINUE
447         IF(BONDML(MTJI,K).EQ.11) THEN
              FDS(I)=CPRIOD
              VOS(I)=CBARR(IT2,IT3)
              VO1S(I)=0.0
            ENDIF
          ENDIF
        ENDIF
        IF(PERIOD(IT2,IT3).EQ.0) THEN
```

```
          FDS(I)=1.0
          VOS(I)=0.0
          VO1S(I)=0.0
          TFCERR=.TRUE.
        ENDIF
C
        XFD=FDS(I)
C
C     -- CALCULATE TORSIONAL ENERGY (ET) --
C
        TA=WA*RAD1
        SGN=XFD/ABS(XFD)
        FOLD=ABS(XFD)
        ET=VOS(I)*(1.0+SGN*COS(FOLD*TA))+VO1S(I)*(1.0+COS(TA))
        SIGET=SIGET+ET
        IF(NPRINT.GT.1.AND.ET.GE.ETHRSH) THEN
          WRITE(*,17) ATMNAM(MTII),ATMNAM(MTJI),ATMNAM(MTKI),
     1    ATMNAM(MTLI),WA,ET
        ENDIF
448     CONTINUE
C
C     -- CALCULATE OUT OF PLANE BENDING ENERGY --
C     -- (IF THEIR ARE ANY O.O.P.B. RECORDS) --
C
C
        IF(NO.EQ.0) GO TO 450
        IF(NPRINT.GT.1) THEN
          WRITE(*,18)
        ENDIF
        DO 449 I=1,NO
C
C     -- CALCULATE IMPROPER TORSION ANGLE --
C
        XJB1=XO(IOPB2(I),1)
        XJB2=XO(IOPB2(I),2)
        XJB3=XO(IOPB2(I),3)
        XJC1=XO(IOPB3(I),1)
        XJC2=XO(IOPB3(I),2)
        XJC3=XO(IOPB3(I),3)
        AA11=XO(IOPB1(I),1)-XJB1
        AA12=XJC1-XJB1
        AA13=XJC1-XO(IOPB4(I),1)
        AA21=XO(IOPB1(I),2)-XJB2
        AA22=XJC2-XJB2
        AA23=XJC2-XO(IOPB4(I),2)
        AA31=XO(IOPB1(I),3)-XJB3
        AA32=XJC3-XJB3
        AA33=XJC3-XO(IOPB4(I),3)
        V11=AA21*AA32-AA31*AA22
        V21=AA22*AA33-AA32*AA23
        V12=AA31*AA12-AA11*AA32
        V22=AA32*AA13-AA12*AA33
        V13=AA11*AA22-AA21*AA12
        V23=AA12*AA23-AA22*AA13
        R1=SQRT(V11*V11+V12*V12+V13*V13)+0.000001
        R2=SQRT(V21*V21+V22*V22+V23*V23)+0.000001
        COSW=(V11/R1)*(V21/R2)+(V12/R1)*(V22/R2)+(V13/R1)*(V23/R2)
        COSW=SIGN(AMIN1(ABS(COSW),1.0E+00),COSW)
        WAOPB=ACOS(COSW)*RADI
C
C     -- CALCULATE CORRECT SIGN FOR ANGLE --
C
        OPBSGN=AA11*(AA22*AA33-AA23*AA32)-AA21*(AA12*AA33-AA13*AA32)
     1         +AA31*(AA12*AA23-AA13*AA22)
        WAOPB=SIGN(WAOPB,OPBSGN)
C
C     -- CALCULATE OUT OF PLANE BENDING ENERGY (EO) --
C
        EO=OPBK(I)*(180.0-ABS(WAOPB))**2
C
        IF(NPRINT.GT.1.AND.EO.GE.ETHRSH) THEN
          WRITE(*,17) ATMNAM(IOPB1(I)),ATMNAM(IOPB2(I)),
     1    ATMNAM(IOPB3(I)),ATMNAM(IOPB4(I)),WAOPB,EO
        ENDIF
        SIGEO=SIGEO+EO
449     CONTINUE
C
C     -- CALCULATE TOTAL ENERGY --
C
450     ETOT=SIGEB+SIGEV+SIGEA+SIGET+SIGEO+SIGEQ
        IF(NPRINT.GT.0) THEN
          WRITE(*,19) SIGEB,SIGEV,SIGEA,SIGET,SIGEO,SIGEQ
          WRITE(*,20) ETOT
        ENDIF
C
        RETURN
        END
```

```
      PROGRAM NODEMIN

                  IMPLICIT NONE

      INCLUDE 'CHMCM3.INC'
      include 'nodeequ.inc'

c      INCLUDE 'HNCOM.INC'
c                  INCLUDE '\\BOARD\\TPR\\NODELINK.INC'

                  include 'chan.inc'
                  include 'node.inc'

      INTEGER NDIV,NMOD,NFIRST,LAST,BFLENG,J,MF,JINDX,
     1        K,ITRCMP,LM,L,M,ii,X,xx,nfirst4

      REAL XO,PEO,XS,PEP,PEN,V,PESP,PESN,SGD1SQ,XSL,PESPL,
     1     PESPL1,PESPL2,XSM,PE2P,AM,SHIFT2,AMI,DET,PD,PDK,
     2     PE0,offset

                  LOGICAL first_iter,error

      DIMENSION AMI(MXATT6),PESP(3),PESN(3),AM(6),V(3),PD(3)
C
C     INITIALIZES NODES

                  call initialize()
C
C     RECEIVES BUFFERS FROM HOST.

                  call receive(host,buffer_atmdat0,1,total_atmdat0,error)
                  call receive(host,buffer_atmdat1,2,total_atmdat1,error)
                  call receive(host,buffer_moldat,3,total_moldat,error)
                  call receive(host,buffer_ffp,4,total_ffp,error)
                  call receive(host,buffer_cffp,5,total_cffp,error)
                  call receive(host,buffer_contrl,6,total_contrl,error)
                  call receive(host,buffer_constn,7,total_constn,error)

c     RECEIVE BYTE VALUES SEPARATELY

                  CALL RECEIVE (HOST,ATYNUM,8,LENGTH9,ERROR)
                  CALL RECEIVE (HOST,BONDML,9,LENGTH10,ERROR)
                  CALL RECEIVE (HOST,MOLNUM,10,LENGTH9,ERROR)

C     ERROR CHECKING. SENDS BUFFERS BACK TO HOST.

C
C
C  -- CALCULATE ATOMIC INTERACTION LISTS ( QUICKER THAN --
C  -- CALCULATING THEM ON THE HOST AND PASSING THEM DOWN --
C  -- THE LINKS ) --
C
      CALL MNINIT1
      SHIFT2 = shiftx * shiftx
C
C  -- ALLOCATE ATOMS TO EACH NODE PROCESSOR --
C
      NDIV = NUMATS / NUMPROC
      NMOD = MOD (NUMATS,NUMPROC)

      IF(me.lt.NMOD)THEN
        NFIRST = (me*NDIV)+me+1
        LAST = ((me+1)*NDIV)+me+1
      ELSE IF(me.eq.NMOD)THEN
         NFIRST = (me*NDIV)+me+1
         LAST = ((me+1)*NDIV)+me
      ELSE IF(me.gt.NMOD)THEN
         NFIRST = (me*NDIV)+NMOD+1
         LAST = ((me+1)*NDIV)+NMOD
      ENDIF

      BFLENG=((LAST+1)-NFIRST)
                  nfirst4 = (nfirst * 4) - 3

                  ITRCMP = 0
                  first_iter = .TRUE.

191   call receive(HOST,XO1,42,length7,error)

C
C   -- CALCULATE FIRST DERIVATIVES USING --
C   -- F'(XI)=[F(XI+DX)-F(XI-DX)]/2DX

C
                  IF (first_iter) THEN
                        CALL LIST_CALC
                        first_iter = .FALSE.
                  END IF
```

```
C
                    sgd1sq = 0.0
      DO 160 J=NFIRST,LAST
      IF(NUMMFX.NE.0) THEN
        DO 105 MF=1,NUMMFX
          IF(MOLNUM(J).EQ.KMOL(MF)) GO TO 160
105     CONTINUE
      ENDIF
      JINDX=(J-1)*6
C
C    -- CALCULATE ENERGY OF JTH ATOM --
C
      CALL ENERGL(J,PE0)


      DO 110 K=1,3
      XS=XO(J,K)
C
C    -- INCREMENT KTH COORDINATE OF JTH ATOM AND RECALCULATE ENERGY --
C
        XO(J,K)=XS+SHIFTX
        CALL ENERGL(J,PEP)
C
C    -- DECREMENT COORDINATE AND RECALCULATE ENERGY --
C
        XO(J,K)=XS-SHIFTX
        CALL ENERGL(J,PEN)
        XO(J,K)=XS
C
C    -- CALCULATE FIRST DERIVATIVES (V) --
C
        V(K)=(PEP-PEN)/(2.0*SHIFTX)
        PESP(K)=PEP
        PESN(K)=PEN
110     CONTINUE

C
C    -- CALCULATE SUM OF SQUARES OF FIRST DERIVATIVES --
C
        DO 115 K=1,3
        SGD1SQ=SGD1SQ+V(K)*V(K)
115     CONTINUE
        IF(NDERIV)125,120,125
C
C    -- IF NOT ON 1ST, 5TH, 9TH ETC. ITERATION, SKIP THE NEXT SECTION --
C
120     IF(MOD(ITRCMP,4).NE.0) GO TO 150
C
C    -- CALCULATE SECOND DERIVATIVES --
C    -- USING  F*(XI,XI)=[F(XI+DX)+F(XI-DX)-2F(XI)]/DX**2 AND --
C    -- F*(XI,XJ)=[F(XI+DX,XJ+DX)-F(XI,XJ+DX)-F(XI+DX,XJ)+F(XI,X)]/DX**2
C
125     LM=1
        DO 145 L=1,3
        XSL=XO(J,L)
        XO(J,L)=XSL+SHIFTX
        PESPL=PESP(L)
        PESPL1=PESPL-PE0
        PESPL2=PESPL1-PE0
        DO 140 M=L,3
        IF(L.EQ.M) GO TO 130
        XSM=XO(J,M)
        XO(J,M)=XSM+SHIFTX
        CALL ENERGL(J,PE2P)
        XO(J,M)=XSM
C
C    -- CALCULATE SECOND DERIVATIVES (AM) --
C
        AM(LM)=(PE2P-PESPL1-PESP(M))/SHIFT2
        GO TO 135
130     AM(LM)=(PESPL2+PESN(M))/SHIFT2
135     LM=LM+1
140     CONTINUE
        XO(J,L)=XSL
145     CONTINUE

C
C    -- INVERT MATRIX OF SECOND DERIVATIVES --
C
        AMI(JINDX+1)=AM(4)*AM(6)-AM(5)*AM(5)
        AMI(JINDX+2)=AM(2)*AM(6)-AM(3)*AM(5)
        AMI(JINDX+3)=AM(2)*AM(5)-AM(3)*AM(4)
        DET=AM(1)*AMI(JINDX+1)-AM(2)*AMI(JINDX+2)+AM(3)*AMI(JINDX+3)
C
C    -- IF DETERMINANT EQUALS ZERO THEN SWITCH TO STEEPEST DESCENTS --
C
```

```
      IF(DET.EQ.0.0) THEN
        AMI(JINDX+1)=1.0E-03
        AMI(JINDX+2)=0.0
        AMI(JINDX+3)=0.0
        AMI(JINDX+4)=1.0E-03
        AMI(JINDX+5)=0.0
        AMI(JINDX+6)=1.0E-03
      ELSE
        AMI(JINDX+4)=AM(1)*AM(6)-AM(3)*AM(3)
        AMI(JINDX+5)=AM(1)*AM(5)-AM(2)*AM(3)
        AMI(JINDX+6)=AM(1)*AM(4)-AM(2)*AM(2)
        AMI(JINDX+1)=AMI(JINDX+1)/DET
        AMI(JINDX+2)=(-1.0)*AMI(JINDX+2)/DET
        AMI(JINDX+3)=AMI(JINDX+3)/DET
        AMI(JINDX+4)=AMI(JINDX+4)/DET
        AMI(JINDX+5)=(-1.0)*AMI(JINDX+5)/DET
        AMI(JINDX+6)=AMI(JINDX+6)/DET
      ENDIF
C
C     -- CALCULATE CORRECTIONS TO COORDINATES (PD) --
C
150     PD(1)=AMI(JINDX+1)*V(1)+AMI(JINDX+2)*V(2)+AMI(JINDX+3)*V(3)
        PD(2)=AMI(JINDX+2)*V(1)+AMI(JINDX+4)*V(2)+AMI(JINDX+5)*V(3)
        PD(3)=AMI(JINDX+3)*V(1)+AMI(JINDX+5)*V(2)+AMI(JINDX+6)*V(3)
C
C     -- CALCULATE NEW COORDINATES --
C
        DO 155 K=1,3
        PDK=PD(K)
        IF(ABS(PDK).GT.SHFTMX) PDK=SIGN(SHFTMX,PDK)
        XO(J,K)=XO(J,K)-PDK
155     CONTINUE
160     CONTINUE


                    call send(HOST,xo1(nfirst4),43,bfleng*4,error)

                    call send(HOST,xo2(nfirst4),44,bfleng*4,error)

                    call send(HOST,xo3(nfirst4),45,bfleng*4,error)

                    call send(HOST,sgd1sq,46,4,error)

                    ITRCMP = ITRCMP + 1
GOTO 191
STOP
END
```

```
C
C
C    -- SUBROUTINE ENERGL(I,EL)
C
C
C    -- THIS SUBROUTINE CALCULATES THE TOTAL ENERGY (EL) --
C    -- OF ALL INTERACTIONS INVOLVING THE ITH ATOM --
C
      SUBROUTINE ENERGL(I,EL)

                  IMPLICIT NONE

      INCLUDE 'CHMCM3.INC'
                  include 'nodeequ.inc'

      INTEGER J,NBMTIJ,ITYPI,ITYPJ,K,ICL,NAMTIK,JAMAI,
     1        JBMAJ,JCMAK,ICA,L,NTMTIL,IT1,IT2,IT3,IT4,
     2        ICT,N,IOPB1N,IOPB2N,IOPB3N,IOPB4N
      REAL ECX,DIR1,DIR2,DIR3,DIST2,DIST,RLITIJ,SCITIJ,
     1     DOIST,EB,RDIST2,RDIST4,RDIST6,RDIS12,EV,EQ,
     2     DISTC,ECL,XJB1,XJB2,XJB3,DC11,DC12,DC21,DC22,
     3     DC31,DC32,RM1,RM2,R12,COSA,A,THSX,DELTH2,DELTH3,
     4     DELTH5,EA,COSAC,AC,ECA,XJC1,XJC2,XJC3,AA11,AA12,
     5     AA13,AA21,AA22,AA23,AA31,AA32,AA33,V11,V21,V12,
     6     V22,V13,V23,R1,R2,COSW,WA,WASIGN,XFD,TA,SGN,
     7     FOLD,ET,WC,WCSIGN,ECT,WAOPB,EO


      EL=0.0
C
C   -- CALCULATE PSEUDO-ENERGIES FOR FIXED ATOMS (IF ANY) --
C
      IF(CONMIN) THEN
        IF(ATCONS(I,1)) THEN
          ECX=FATSEV(I)*(
     1          (FATXYZ(I,1)-XO(I,1))**2+
     2          (FATXYZ(I,2)-XO(I,2))**2+
     3          (FATXYZ(I,3)-XO(I,3))**2)
          EL=EL+ECX
        ENDIF
      ENDIF
C
C    -- CALCULATE THE BONDED AND NON-BONDED ENERGIES --
C
        DO 101 J=1,NUMATS
        NBMTIJ=NBMAT(J,I)
C
C    -- SORT OUT WHICH PAIRS OF ATOMS ARE 1,1 OR 1,3 OR TOO --
C    -- LONG (NBMTIJ=5), SKIP THESE AND GO ON TO NEXT PAIR --
C
        IF(NBMTIJ.EQ.1.OR.NBMTIJ.EQ.3.OR.NBMTIJ.EQ.5) GO TO 101
C
C    -- CALCULATE DISTANCE I - J --
C
        DIR1=XO(I,1)-XO(J,1)
        DIR2=XO(I,2)-XO(J,2)
        DIR3=XO(I,3)-XO(J,3)
        DIST2=DIR1*DIR1+DIR2*DIR2+DIR3*DIR3
        ITYPI=ATYNUM(I)
        ITYPJ=ATYNUM(J)
C
C    -- IF ATOMS I AND J ARE NON BONDED, --
C    -- GO ON TO VAN DER WAALS SECTION --
C
        IF(NBMTIJ.EQ.4) GO TO 100
        DIST=SQRT(DIST2)
C
C    -- GET REFERENCE LENGTHS AND FORCE CONSTANTS --
C
      RLITIJ=REFLEN(ITYPI,ITYPJ)
      SCITIJ=STRCON(ITYPI,ITYPJ)
      IF(ITYPI.LE.Namide.AND.ITYPJ.LE.Namide) THEN
        IF(CREFLN(ITYPI,ITYPJ).NE.10.) THEN
          DO 98 K=1,MXCN
            IF(ATMCON(I,K).EQ.J) GO TO 99
98        CONTINUE
99        IF(BONDML(I,K).EQ.11) THEN
            RLITIJ=CREFLN(ITYPI,ITYPJ)
            SCITIJ=CSTCON(ITYPI,ITYPJ)
          ENDIF
        ENDIF
      ENDIF
      DOIST=RLITIJ-DIST
C
C    -- CALCULATE BOND ENERGY --
C
      EB=SCITIJ*DOIST*DOIST
```

```
      EL=EL+EB
      GO TO 101
C
C   -- VAN DER WAALS SECTION --
C
100   RDIST2=1.0/DIST2
      RDIST4=RDIST2*RDIST2
      RDIST6=RDIST2*RDIST4
      RDIS12=RDIST6*RDIST6
C
C   -- CALCULATE VAN DER WAALS ENERGY (EV) --
C
      EV=B12(ITYPI,ITYPJ)*RDIS12 - A6(ITYPI,ITYPJ)*RDIST6
      EL=EL+EV
C
C   -- CALCULATE COULOMBIC ENERGY (EQ) --
C
      EQ=332.17*CHARGE(I)*CHARGE(J)*RDIST2
      EL=EL+EQ
101   CONTINUE
C
C  -- CALCULATE PSEUDO-ENERGIES FOR FIXED LENGTHS (IF ANY) --
C
      IF(CONMIN) THEN
        DO 102 ICL=1,NUMLFX
        IF(ATCONS(I,2).AND.
     1        (KLATM1(ICL).EQ.I.OR.KLATM2(ICL).EQ.I)) THEN
          DIR1=XO(KLATM1(ICL),1)-XO(KLATM2(ICL),1)
          DIR2=XO(KLATM1(ICL),2)-XO(KLATM2(ICL),2)
          DIR3=XO(KLATM1(ICL),3)-XO(KLATM2(ICL),3)
          DIST2=DIR1*DIR1+DIR2*DIR2+DIR3*DIR3
          DISTC=SQRT(DIST2)
          ECL=FLNSEV(ICL)*((FIXLEN(ICL)-DISTC)**2)
          EL=EL+ECL
        ENDIF
102     CONTINUE
      ENDIF
C
C   -- CALCULATE THE ANGLE ENERGY --
C
      DO 103 K=1,18
C
C   -- CHECK IF ATOM IS INVOLVED IN ANY ANGLES --
C   -- WHEN DONE GO TO ANGLE CONSTRAINTS SECTION --
C
      NAMTIK=NAMAT(K,I)
      IF(NAMTIK.EQ.0) GO TO 104
      JAMAI=MAI(NAMTIK)
      JBMAJ=MAJ(NAMTIK)
      JCMAK=MAK(NAMTIK)
C
C   -- CALCULATE ANGLE --
C
      XJB1=XO(JBMAJ,1)
      XJB2=XO(JBMAJ,2)
      XJB3=XO(JBMAJ,3)
      DC11=XO(JAMAI,1)-XJB1
      DC12=XO(JCMAK,1)-XJB1
      DC21=XO(JAMAI,2)-XJB2
      DC22=XO(JCMAK,2)-XJB2
      DC31=XO(JAMAI,3)-XJB3
      DC32=XO(JCMAK,3)-XJB3
      RM1=DC11*DC11+DC21*DC21+DC31*DC31
      RM2=DC12*DC12+DC22*DC22+DC32*DC32
      R12=DC11*DC12+DC21*DC22+DC31*DC32
      RM1=SQRT(RM1)+0.000001
      RM2=SQRT(RM2)+0.000001
      COSA=R12/(RM1*RM2)
      COSA=SIGN(AMIN1(ABS(COSA),1.0E+00),COSA)
      A=ACOS(COSA)*RADI
      THSX=THS(NAMTIK)-A
      DELTH2=THSX*THSX
      DELTH3=ABS(DELTH2*THSX)
      DELTH5=DELTH3*DELTH2
C
C   -- CALCULATE ANGLE ENERGY --
C
      EA=BKS(NAMTIK)*(DELTH2-BKAS(NAMTIK)*(DELTH3-(0.0004*DELTH5)))
      EL=EL+EA
103   CONTINUE
C
C   -- CALCULATE PSEUDO-ENERGIES FOR FIXED ANGLES (IF ANY) --
C
104   IF(CONMIN) THEN
        DO 105 ICA=1,NUMAFX
        IF(ATCONS(I,3).AND.
     1        (KAATM1(ICA).EQ.I.OR.KAATM2(ICA).EQ.I.OR.
     2    KAATM3(ICA).EQ.I)) THEN
          XJB1=XO(KAATM2(ICA),1)
          XJB2=XO(KAATM2(ICA),2)
          XJB3=XO(KAATM2(ICA),3)
          DC11=XO(KAATM1(ICA),1)-XJB1
```

```
            DC12=XO(KAATM3(ICA),1)-XJB1
            DC21=XO(KAATM1(ICA),2)-XJB2
            DC22=XO(KAATM3(ICA),2)-XJB2
            DC31=XO(KAATM1(ICA),3)-XJB3
            DC32=XO(KAATM3(ICA),3)-XJB3
            RM1=DC11*DC11+DC21*DC21+DC31*DC31
            RM2=DC12*DC12+DC22*DC22+DC32*DC32
            R12=DC11*DC12+DC21*DC22+DC31*DC32
            RM1=SQRT(RM1)+0.000001
            RM2=SQRT(RM2)+0.000001
            COSAC=R12/(RM1*RM2)
            COSAC=SIGN(AMIN1(ABS(COSAC),1.0E+00),COSAC)
            AC=ACOS(COSAC)*RADI
            ECA=FANSEV(ICA)*((FIXANG(ICA)-AC)**2)
            EL=EL+ECA
          ENDIF
105       CONTINUE
        ENDIF
C
C     -- CALCULATE TORSIONAL ENERGY --
C
          DO 106 L=1,50
C
C     -- CHECK IF ATOM IS INVOLVED IN ANY TORSION ANGLES --
C     -- WHEN DONE GO TO TORSION ANGLE CONSTRAINTS SECTION --
C
          NTMTIL=NTMAT(L,I)
          IF(NTMTIL.EQ.0) GO TO 107
C
C     -- CALCULATE TORSION ANGLE --
C
          IT1=MTI(NTMTIL)
          IT2=MTJ(NTMTIL)
          IT3=MTK(NTMTIL)
          IT4=MTL(NTMTIL)
          XJB1=XO(IT2,1)
          XJB2=XO(IT2,2)
          XJB3=XO(IT2,3)
          XJC1=XO(IT3,1)
          XJC2=XO(IT3,2)
          XJC3=XO(IT3,3)
          AA11=XO(IT1,1)-XJB1
          AA12=XJC1-XJB1
          AA13=XJC1-XO(IT4,1)
          AA21=XO(IT1,2)-XJB2
          AA22=XJC2-XJB2
          AA23=XJC2-XO(IT4,2)
          AA31=XO(IT1,3)-XJB3
          AA32=XJC3-XJB3
          AA33=XJC3-XO(IT4,3)
          V11=AA21*AA32-AA31*AA22
          V21=AA22*AA33-AA32*AA23
          V12=AA31*AA12-AA11*AA32
          V22=AA32*AA13-AA12*AA33
          V13=AA11*AA22-AA21*AA12
          V23=AA12*AA23-AA22*AA13
          R1=SQRT(V11*V11+V12*V12+V13*V13)+0.000001
          R2=SQRT(V21*V21+V22*V22+V23*V23)+0.000001
          COSW=(V11/R1)*(V21/R2)+(V12/R1)*(V22/R2)+(V13/R1)*(V23/R2)
          COSW=SIGN(AMIN1(ABS(COSW),1.0E+00),COSW)
          WA=ACOS(COSW)*RADI
C
C     -- CALCULATE CORRECT SIGN FOR TORSION ANGLE --
C
          WASIGN=AA11*(AA22*AA33-AA23*AA32)-AA21*(AA12*AA33-AA13*AA32)
         1       +AA31*(AA12*AA23-AA13*AA22)
          WA=SIGN(WA,WASIGN)
          XFD=FDS(NTMTIL)
C
C     -- CALCULATE TORSIONAL ENERGY --
C
          TA=WA*RAD1
          SGN=XFD/ABS(XFD)
          FOLD=ABS(XFD)
          ET=VOS(NTMTIL)*(1.0+SGN*COS(FOLD*TA))+VO1S(NTMTIL)*
         1(1.0+COS(TA))
          EL=EL+ET
106       CONTINUE
C
C   -- CALCULATE PSEUDO-ENERGIES FOR FIXED TORSION ANGLES (IF ANY) --
C
107       IF(CONMIN) THEN
            DO 108 ICT=1,NUMTFX
            IF(ATCONS(I,4).AND.
     1          (KTATM1(ICT).EQ.I.OR.KTATM2(ICT).EQ.I.OR.
     2     KTATM3(ICT).EQ.I.OR.KTATM4(ICT).EQ.I)) THEN
              IT1=KTATM1(ICT)
              IT2=KTATM2(ICT)
```

```
      IT3=KTATM3(ICT)
      IT4=KTATM4(ICT)
      XJB1=XO(IT2,1)
      XJB2=XO(IT2,2)
      XJB3=XO(IT2,3)
      XJC1=XO(IT3,1)
      XJC2=XO(IT3,2)
      XJC3=XO(IT3,3)
      AA11=XO(IT1,1)-XJB1
      AA12=XJC1-XJB1
      AA13=XJC1-XO(IT4,1)
      AA21=XO(IT1,2)-XJB2
      AA22=XJC2-XJB2
      AA23=XJC2-XO(IT4,2)
      AA31=XO(IT1,3)-XJB3
      AA32=XJC3-XJB3
      AA33=XJC3-XO(IT4,3)
      V11=AA21*AA32-AA31*AA22
      V21=AA22*AA33-AA32*AA23
      V12=AA31*AA12-AA11*AA32
      V22=AA32*AA13-AA12*AA33
      V13=AA11*AA22-AA21*AA12
      V23=AA12*AA23-AA22*AA13
      R1=SQRT(V11*V11+V12*V12+V13*V13)+0.000001
      R2=SQRT(V21*V21+V22*V22+V23*V23)+0.000001
      COSW=(V11/R1)*(V21/R2)+(V12/R1)*(V22/R2)+(V13/R1)*(V23/R2)
      COSW=SIGN(AMIN1(ABS(COSW),1.0E+00),COSW)
      WC=ACOS(COSW)*RADI
      WCSIGN=AA11*(AA22*AA33-AA23*AA32)-AA21*(AA12*AA33-AA13*AA32)
     1                 +AA31*(AA12*AA23-AA13*AA22)
      WC=SIGN(WC,WCSIGN)
      ECT=FTOSEV(ICT)*(
     1          (FIXTOR(ICT)-WC)**2)
      EL=EL+ECT
     ENDIF
108   CONTINUE
     ENDIF
C
C    -- CALCULATE OUT OF PLANE BENDING ENERGY --
C    -- (IF THERE SHOULD BE ANY) --
C
      IF(NO.EQ.0) RETURN
      DO 113 N=1,NO
       IOPB1N=IOPB1(N)
       IOPB2N=IOPB2(N)
       IOPB3N=IOPB3(N)
       IOPB4N=IOPB4(N)
       IF((IOPB1N.EQ.I).OR.(IOPB2N.EQ.I).OR.
     1   (IOPB3N.EQ.I).OR.(IOPB4N.EQ.I)) THEN
         GOTO 112
       ELSE
         GOTO 113
       ENDIF
C
C    -- CALCULATE IMPROPER TORSION ANGLE --
C
112    XJB1=XO(IOPB2N,1)
       XJB2=XO(IOPB2N,2)
       XJB3=XO(IOPB2N,3)
       XJC1=XO(IOPB3N,1)
       XJC2=XO(IOPB3N,2)
       XJC3=XO(IOPB3N,3)
       AA11=XO(IOPB1N,1)-XJB1
       AA12=XJC1-XJB1
       AA13=XJC1-XO(IOPB4N,1)
       AA21=XO(IOPB1N,2)-XJB2
       AA22=XJC2-XJB2
       AA23=XJC2-XO(IOPB4N,2)
       AA31=XO(IOPB1N,3)-XJB3
       AA32=XJC3-XJB3
       AA33=XJC3-XO(IOPB4N,3)
       V11=AA21*AA32-AA31*AA22
       V21=AA22*AA33-AA32*AA23
       V12=AA31*AA12-AA11*AA32
       V22=AA32*AA13-AA12*AA33
       V13=AA11*AA22-AA21*AA12
       V23=AA12*AA23-AA22*AA13
       R1=SQRT(V11*V11+V12*V12+V13*V13)+0.000001
       R2=SQRT(V21*V21+V22*V22+V23*V23)+0.000001
       COSW=(V11/R1)*(V21/R2)+(V12/R1)*(V22/R2)+(V13/R1)*(V23/R2)
       COSW=SIGN(AMIN1(ABS(COSW),1.0E+00),COSW)
       WAOPB=ACOS(COSW)*RADI
C
C    -- CALCULATE OUT OF PLANE BENDING ENERGY --
C
       EO=OPBK(N)*(180.0-ABS(WAOPB))**2
       EL=EL+EO
```

```
113     CONTINUE
        RETURN
        END
```

```
INCLUDE 'fgraph.fi'
    INCLUDE 'c:\lesley\MOUSE.FI'
    SUBROUTINE nrm_get_control_parameters
    INCLUDE 'chmcm3.inc'
                INCLUDE 'commnpcn.inc'
                INCLUDE 'commmenu.inc'
  INCLUDE 'FGRAPH.FD'
                INTEGER nboxes, chce, DISP_MIN_PARAM
                LOGICAL start
                REAL rdum
                OPEN (UNIT = 10, FILE = 'XY.OUT', STATUS = 'old')
                numitr = 10
                dxxm = 25.0
                nprint = 0
                ethrsh = -10.0
                shftmx = 0.5
                nderiv = 1
                nboxes = 8
                CALL graphicsmode()
                CALL register_fonts
 CALL init_constraints
 DISP_MIN_PARAM = 1
                CALL init_screen_area(DISP_MIN_PARAM)
                CALL init_option_box_data (nboxes,
1       'NEWTON-RAPHSON MINIMISER PARAMETERS')
                CALL init_npad_data()
                start = .FALSE.
                DO WHILE (.NOT.start)
                   CALL nrm_display_options(chce)
                       IF (chce.EQ.1) THEN
         CALL MESSAGE(1,'ENTER NO.OF ITERATIONS',13)
                                    numitr = 10
                                    CALL number_pad(rdum,numitr,2)
                                    numitr = MAX(numitr,0)
                                    CALL MESSAGE(1,'ENTER NO.OF ITERATIONS',0)
                       ELSE IF (chce.eq.2) THEN
                                       CALL MESSAGE(1,
1         'ENTER VAN DER WAALS CUTOFF DISTANCE',13)
          DXXM = 25.0
                                       CALL number_pad(DXXM,IDUM,1)
                                       CALL MESSAGE(1,
1          'ENTER VAN DER WAALS CUTOFF DISTANCE',0)
          DXXM = MAX(DXXM,2.0)
       ELSE IF ((CHCE.EQ.3).AND.(NPRINT.EQ.2)) THEN
                                       CALL MESSAGE(1,
1           'ENTER PRINT THRESHOLD ENERGY',13)
          ethrsh = -10.0
                                       CALL number_pad(ethrsh,idum,1)
                                       CALL MESSAGE(1,
1           'ENTER PRINT THRESHOLD ENERGY',0)
        ELSE IF (CHCE.EQ.4) THEN
                                       CALL MESSAGE(1,
1             'ENTER MAXIMUM COORDINATE SHIFT',13)
          shftmx = 0.5
          CALL number_pad(shftmx,IDUM,1)
                                       CALL MESSAGE(1,
1             'ENTER MAXIMUM COORDINATE SHIFT',0)
                                       shftmx = MAX(shftmx,0.0001)
        ELSE IF (CHCE.EQ.5) THEN
                           IF (nprint .EQ. 0) THEN
                                       nprint = 1
                           ELSE IF (nprint .EQ. 1) THEN
                                       nprint = 2
                           ELSE IF (nprint .EQ. 2) THEN
                                       nprint = 0
                                    END IF
      ELSE IF (CHCE.EQ.6) THEN
           call fix_parameters
                        CALL init_screen_area(DISP_MIN_PARAM)
```

```
                    CALL init_option_box_data (nboxes,
1                 'NEWTON-RAPHSON MINIMISER PARAMETERS')
      CALL init_npad_data()
    ELSE IF (CHCE.EQ.7) THEN
          CALL CLEARSCREEN($GCLEARSCREEN)
                                        call endgraphics
          START = .TRUE.
     ELSE IF (CHCE.EQ.8) THEN
                        CALL endgraphics
          STOP
     END IF
               END DO

               END


               SUBROUTINE init_option_box_data(nbox,mess)
               INTEGER nbox
               CHARACTER * (*) mess

               INCLUDE 'commobcm.inc'

   INCLUDE 'commgrap.inc'

               INTEGER txtwd,txtht,xtp,ytp,tsx,xs1,xs2,tsy,ys1,ys2,yt,ty,
1    px,py,nb,btx,bty,i

               LOGICAL error

               CALL set_font(ch12w9)
               CALL text_info(txtwd,txtht)

               write(10,*) message_spacey

               xtp = viewport_xspace
               ytp = viewport_yspace

   optnbox_width = MAX(txtwd * 2,txtht * 2)
   optnbox_height = optnbox_width
               nb = MIN(nbox,ytp / (optnbox_height + 1))
               IF (nb .LT. nbox) THEN
                   error = .TRUE.
                   RETURN
END IF
               bty = nb * optnbox_height
               tsy = ytp - bty
ys1 = MIN(MAX(tsy / (nb + 1),1),txtht * 2)
               ys2 = (tsy - ((nb - 1) * ys1)) / 2
               xs1 = 1
btx = optnbox_width
               tsx = xtp - btx
               optnbox_maxm = MIN((tsx - (xs1 * 3)) / txtwd,50)
               write(10,*) 'optnbox_maxm is', optnbox_maxm

xs1 = MIN(MAX(tsx - (optnbox_maxm * txtwd),1),txtwd * 2)
               write (10,*) 'xs1 is',xs1
xs2 = MAX((tsx - ((optnbox_maxm * txtwd) + xs1)) / 2,1)
               write (10,*) 'xs2 is',xs2
               optnbox_max1 = INT((FLOAT(optnbox_maxm) * 3.0) / 5.0)
               optnbox_max2 = optnbox_maxm - optnbox_max1
               optnbox_tx1 =  xs2
               optnbox_tx2 = optnbox_tx1 + (optnbox_max1 * txtwd)
optnbox_x1 = optnbox_tx1 + (optnbox_maxm * txtwd) + xs1
               yt = 0 + ys2
               px = optnbox_x1 + (optnbox_width / 2)
DO i = 1,nb
    ty = yt + (txtht / 2)
                py = yt + (optnbox_height / 2)
    optnbox_yt(i) = yt
                    optnbox_ty(i) = ty
                    optnbox_px(i) = px
                    optnbox_py(i) = py
                    yt = yt + optnbox_height + ys1
END DO
               optnbox_nbox = nb
               optnbox_titmess = mess

END


               SUBROUTINE nrm_display_options(chce)

               INCLUDE 'chmcm3.inc'

               INCLUDE 'commobcm.inc'

               INTEGER chce
```

```
                    INTEGER*2 mx,my
                    CHARACTER*30 msg1(20)
                    CHARACTER*20 msg2(20)

                    optnbox_col(1) = 3
                    msg1(1) = 'NUMBER OF ITERATIONS'
                    WRITE (msg2(1),10) numitr
10        FORMAT ('    (',2X,I10,3X,')')

                    optnbox_col(2) = 3
                    msg1(2) = 'VAN DER WAALS CUTOFF DISTANCE'
                    WRITE (msg2(2),30) dxxm
30        FORMAT ('    (',8X,F7.2,')')

                   msg1(3) = 'ENERGY THRESHOLD FOR PRINTING'
                   IF (nprint .EQ. 2) THEN
                       optnbox_col(3) = 3
                           WRITE (msg2(3),40) ethrsh
40                         FORMAT ('    (',9X,F6.2,')')
                   ELSE
                       optnbox_col(3) = 8
                           msg2(5) = ' '
                   END IF

                   msg1(4) = 'MAXIMUM SHIFT'
                   IF (numitr .GT. 0) THEN
                       optnbox_col(4) = 3
                           WRITE (msg2(4),20) shftmx
20                         FORMAT ('    (',11X,F4.2,')')
                   ELSE
                       optnbox_col(4) = 8
                           msg2(2) = ' '
                   END IF

                   optnbox_col(5) = 3
                   msg1(5) = 'TYPE OF OUTPUT'
                   IF (nprint .EQ. 0) THEN
                       msg2(5) = '    (           SHORT)'
                   ELSE IF (nprint .EQ. 2) THEN
                       msg2(5) = '    (           LONG )'
                   ELSE IF (nprint .EQ. 1) THEN
                       msg2(5) = '    (   ABBREIVIATED)'
                   END IF

                   optnbox_col(6) = 3
                   msg1(6) = 'FIX ATOMS OR PARAMETERS'
                   msg2(6) = ' '

                   optnbox_col(7) = 3
                   msg1(7) = 'START MINIMISER'
                   msg2(7) = ' '

                   optnbox_col(8) = 12
                   msg1(8) = 'EXIT MINIMISER'
                   msg2(8) = ' '

                   CALL draw_option_boxes(msg1,msg2)
                   CALL mouse()

                   chce = 0
                   DO WHILE(chce.EQ.0)
CALL chms(mx,my)
                         CALL find_box_select(mx,my,chce)
                   END DO

                   END

                   SUBROUTINE draw_option_boxes(msgs1,msgs2)

                   CHARACTER*(*) msgs1(*),msgs2(*)

                   INCLUDE 'commobcm.inc'

                   INCLUDE 'FGRAPH.FD'

              INCLUDE 'commgrap.inc'

C                  INCLUDE 'colour.inc'
                   INTEGER i,x1,x2,y1,y2

              CALL CLEARSCREEN($GCLEARSCREEN)
                        CALL set_font(ch12w9)
                        CALL MESSAGE(0,optnbox_titmess,9)

                        DO i = 1,optnbox_nbox
                               x1 = optnbox_x1
                               x2 = x1 + optnbox_width
                               y1 = optnbox_yt(i)
```

```
                    y2 = y1 + optnbox_height
                    write(10,*) x1,y1,x2,y2
                CALL colour(optnbox_col(i))
                    CALL box(x1,y1,x2,y2)
                    CALL text(optnbox_tx1,optnbox_ty(i),msgs1(i))
                    CALL text(optnbox_tx2,optnbox_ty(i),msgs2(i))
END DO

            END

            SUBROUTINE chms(xpos,ypos)

INCLUDE 'c:\lesley\MOUSE.FD'
INCLUDE 'FGRAPH.FD'

            INTEGER*2 XPOS,YPOS,BPOS

            bpos = 0

            CALL showmousecursor

            DO WHILE(BPOS.EQ.0)
               CALL GETMOUSECURSORPOSITION(XPOS,YPOS,BPOS)
END DO

            DO WHILE(BPOS.EQ.1)
               CALL GETMOUSECURSORPOSITION(XPOS,YPOS,BPOS)
END DO

            CALL convert_to_viewport_coords(XPOS,YPOS)

            CALL hidemousecursor

            END

            SUBROUTINE find_box_select(xpos,ypos,chce)

INCLUDE 'c:\lesley\MOUSE.FD'
INCLUDE 'FGRAPH.FD'
               INCLUDE 'commobcm.inc'

               INTEGER chce,half_width,half_height,i
               INTEGER*2 XPOS,YPOS

               half_width = optnbox_width/2
               half_height = optnbox_height/2

               chce = 0

               DO i = 1,optnbox_nbox
                     IF (XPOS.GT.(optnbox_px(i) - half_width).AND.
1   XPOS.LT.(optnbox_px(i) + half_width).AND.
2   YPOS.GT.(optnbox_py(i) - half_height).AND.
3   YPOS.LT.(optnbox_py(i) + half_height)) THEN
                                          chce = i

                     END IF

               END DO

               END

               SUBROUTINE find_menu_select(xpos,ypos,num_box,xcentre,ycentre,
1                    xhalf_width,yhalf_height,kk)

   INCLUDE 'c:\lesley\MOUSE.FD'
   INCLUDE 'FGRAPH.FD'

               INTEGER num_box,xcentre(num_box),ycentre(num_box),
1    xhalf_width,yhalf_height,kk

               INTEGER*2 XPOS,YPOS

               kk = 0

               DO i = 1,num_box
                     IF (XPOS.GT.(xcentre(i) - xhalf_width).AND.
1   XPOS.LT.(xcentre(i) + xhalf_width).AND.
2   YPOS.GT.(ycentre(i) - yhalf_height).AND.
3   YPOS.LT.(ycentre(i) + yhalf_height)) THEN
                                       kk = i
                     END IF

               END DO

               END
```

```
              SUBROUTINE init_npad_data

              INCLUDE 'commnpcn.inc'

              INCLUDE 'commgrap.inc'

    INTEGER ht2,wd2,padrowp,padcolp,padysp,padxsp,
1    pxsp2,pysp2,wd22,ht22,row,col,rowp,colp,tlen,tlenp,
2    tlenp2,txsp,txsp2,
3    i,j,xtp,ytp

    CALL set_font(ch15w12)
                  CALL text_info(wd2,ht2)
                  CALL get_screen_coords(xtp,ytp)
    padrowp = pad_row * ht2
    padcolp = pad_col * wd2
    padysp = ((ytp - message_spacey) - padrowp) / (pad_row + 1)
    padxsp = (menu_spacex - padcolp) / (pad_col + 1)
    pxsp2 = padxsp / 3
    pysp2 = padysp / 3
    wd22 = wd2 / 2
    ht22 = ht2 / 2
    DO i = 1,num_box
        row = (i - 1) / pad_col
        col = MOD(i - 1,pad_col)
        rowp = 0 + ((row * (ht2 + padysp)) + padysp)
        colp = (0 - menu_spacex) + (col * (wd2 + padxsp)) + padxsp
        np_pbxl(i) = colp - pxsp2
        np_pbyt(i) = rowp - pysp2
        np_pbxr(i) = np_pbxl(i) + (wd22 + pxsp2) * 2
        np_pbyb(i) = np_pbyt(i) + (ht22 + pysp2) * 2
        np_ppkx(i) = colp + wd22
        np_ppky(i) = rowp + ht22
                       np_ptxx(i) = colp
                       np_ptxy(i) = rowp
    END DO
    np_pxtl = wd22 + pxsp2
    np_pytl = ht22 + pysp2
    tlen = LEN(other_text(1))
    tlenp = tlen * wd2
    tlenp2 = tlenp / 2
    txsp = (menu_spacex - tlenp) / 2
    txsp2 = txsp / 3
    DO j = 1,num_other
        rowp = 0 + (((j + 3) * (ht2 + padysp)) + padysp)
        colp = (0 - menu_spacex) + txsp
        np_obxl(j) = colp - txsp2
        np_obyt(j) = rowp - pysp2
        np_obxr(j) = np_obxl(j) + (tlenp2 + txsp2) * 2
        np_obyb(j) = np_obyt(j) + (ht22 + pysp2) * 2
        np_opkx(j) = colp + tlenp2
        np_opky(j) = rowp + ht22
                       np_otxx(j) = colp
                       np_otxy(j) = rowp
    END DO
    np_oxtl = tlenp2 + txsp2
    np_oytl = ht22 + pysp2

              END

              BLOCK DATA number_pad_data

    INCLUDE 'commnpcn.inc'

    DATA padn_text  / '7894561230-.' /
    DATA other_text / ' RESET ',' ENTER ','DEFAULT' /

              END

    SUBROUTINE number_pad(pad_value,ipad_value,option)

    REAL pad_value
    INTEGER ipad_value,option

              INCLUDE 'commnpcn.inc'

    CHARACTER * 10 value_text
    CHARACTER * 20 def_text
    INTEGER kt,kk,idef,noth
              INTEGER*2 mx,my
              REAL rdef
    LOGICAL exitf,fraction,negative,default

              IF (option .EQ. 0) THEN
                      noth = 2
                      def_text = ' '
              ELSE
                      noth = 3
```

```
                        IF (option .EQ. 1) THEN
                                        rdef = pad_value
                                        WRITE (def_text,10) rdef
10                                      FORMAT ('DEFAULT : ',F10.4)
                        ELSE IF (option .EQ. 2) THEN
                                        idef = ipad_value
                                        WRITE (def_text,20) idef
20                                      FORMAT ('DEFAULT : ',I10)
                        END IF
        END IF
                    pad_value = 0.0
                    ipad_value = 0
        value_text = ' '
        fraction = .FALSE.
        negative = .FALSE.
                    default = .FALSE.
        kt = 1
                    exitf = .FALSE.
                    CALL draw_npad_boxes(13,noth)
                    CALL draw_npad_text(13,noth)
                    CALL message(3,def_text,10)
                    DO WHILE (.NOT. exitf)
                       CALL chms(mx,my)
                            CALL find_menu_select(mx,my,num_box,np_ppkx,np_ppky,
1                                   np_pxtl,np_pytl,kk)
         IF ((kk .GE. 1) .AND. (kk .LE. 10)) THEN
                             IF (kt .LT. 10) THEN
                    CALL message(2,value_text,0)
                    kt = kt + 1
                    value_text(kt:kt) = padn_text(kk:kk)
                    CALL message(2,value_text,15)
                                END IF
        ELSE IF (kk .EQ. 11) THEN
                    CALL message(2,value_text,0)
                                IF (negative) THEN
                    value_text(1:1) = ' '
            ELSE
                    value_text(1:1) = '-'
                             END IF
                    CALL message(2,value_text,15)
                                negative = .NOT. negative
        ELSE IF ((kk .EQ. 12) .AND. (.NOT. fraction) .AND.
1        (kt .LT. 9)) THEN
                    CALL message(2,value_text,0)
                    fraction = .TRUE.
                    kt = kt + 1
                    value_text(kt:kt) = padn_text(kk:kk)
                    CALL message(2,value_text,15)
                       ELSE
                            CALL find_menu_select(mx,my,noth,np_opkx,np_opky,
1                                                   np_oxtl,np_oytl,kk)
                    IF (kk .EQ. 1) THEN
                    CALL message(2,value_text,0)
                    kt = 1
                    negative = .FALSE.
                    fraction = .FALSE.
                    value_text = ' '
                                ELSE IF (kk .EQ. 2) THEN
                                    exitf = .TRUE.
                                            ELSE IF (kk. EQ. 3) THEN
                                    exitf = .TRUE.
                                                    default = .TRUE.
                                END IF
                       END IF
                  END DO
    IF (default) THEN
                        IF (option .EQ. 1) THEN
                                        pad_value = rdef
                        ELSE IF (option .EQ. 2) THEN
                                        ipad_value = idef
            END IF
                        ELSE
            READ(value_text,30) pad_value
30          FORMAT(F10.0)
                                ipad_value = INT(pad_value)
                        END IF
                        CALL draw_npad_boxes(0,noth)
                        CALL draw_npad_text(0,noth)
          CALL message(2,value_text,0)
                        CALL message(3,def_text,0)

                        END

                        SUBROUTINE draw_npad_boxes(col,noth)

                        INTEGER col,noth

                        INCLUDE 'commnpcn.inc'
```

```
      INTEGER i

      CALL colour(col)
      DO i = 1,num_box
            CALL BOX(np_pbxl(i),np_pbyt(i),np_pbxr(i),np_pbyb(i))
      END DO
      DO i = 1,noth
            CALL BOX(np_obxl(i),np_obyt(i),np_obxr(i),np_obyb(i))
      END DO

      END


SUBROUTINE draw_npad_text(col,noth)

INTEGER col,noth
            INCLUDE 'commnpcn.inc'
INCLUDE 'commgrap.inc'


INTEGER i

CALL set_font(ch15w12)
CALL colour(col)
DO i = 1,num_box
    CALL text(np_ptxx(i),np_ptxy(i),padn_text(i:i))
END DO
DO i = 1,noth
    CALL text(np_otxx(i),np_otxy(i),other_text(i))
END DO

END

            SUBROUTINE fix_parameters

INCLUDE 'CHMCM3.INC'

integer num_opt,disp_mol,chce,I,J,disp_min_param
INTEGER*2 XPOS,YPOS
 LOGICAL exitf


 NUMLFX=0
 NUMAFX=0
 NUMTFX=0
 NUMMFX=0
 CONMIN = .FALSE.


 DO I = 1,4
    DO J = 1,NUMATS
       ATCONS(j,i) = .false.
    END DO
 END DO

            num_opt = 6
 DISP_MOL = 2
            CALL init_screen_area(DISP_MOL)
            CALL init_npad_data()
CALL get_mol_screen_coords
            CALL draw_simple_stick_molecule
            CALL draw_atom_numbers(15)
            CALL init_menu_data(num_opt)

            exitf = .FALSE.
          DO WHILE(.NOT.exitf)
               CALL init_menu_data(num_opt)
               CALL display_fix_menu(num_opt)
               CHCE = 0
               DO WHILE (CHCE.EQ.0)
                  call chms(xpos,ypos)
                  call find_side_menu_select(xpos,ypos,chce,num_opt)
               END DO

               IF (CHCE.EQ.1) THEN
                  CALL ATOM_CONSTRAINT
                  CONMIN = .TRUE.
               ELSE IF (CHCE.EQ.2) THEN
                  CALL length_constraint
                 CONMIN = .TRUE.
ELSE IF (CHCE.EQ.3) THEN
   CALL angle_constraint
   CONMIN = .TRUE.
ELSE IF (CHCE.EQ.4) THEN
```

```
      CALL torsion_constraint
      CONMIN = .TRUE.
   ELSE IF (CHCE.EQ.5) THEN
      CALL molecule_constraint
      CONMIN = .TRUE.
                ELSE
                  EXITF = .TRUE.
                END IF
           END DO

           END

            SUBROUTINE INIT_CONSTRAINTS

            INCLUDE 'CHMCM3.inc'

INTEGER I,J

NUMLFX=0
NUMAFX=0
NUMTFX=0
NUMMFX=0
CONMIN = .FALSE.

DO I = 1,4
    DO J = 1,NUMATS
       ATCONS(j,i) = .false.
    END DO
END DO

END


 SUBROUTINE FIND_SIDE_MENU_SELECT(XPOS,YPOS,CHCE,NUM_OPT)

 INCLUDE 'COMMMENU.INC'

 INTEGER CHCE,NUM_OPT,I
 INTEGER*2 XPOS,YPOS

 chce = -1

 DO I = 1,NUM_OPT
    IF (XPOS.GT.MENU_OBXL(I).AND.
1   XPOS.LT.MENU_OBXR(I).AND.
2   YPOS.GT.MENU_OBYT(I).AND.
3   YPOS.LT.MENU_OBYB(I)) THEN
        CHCE = I
    END IF
   END DO

   END


   SUBROUTINE ATOM_CONSTRAINT

   INCLUDE 'commgrap.inc'

   LOGICAL exitf
               INTEGER idum,ikon,ITOK,num_opt
               REAL severity

   num_opt = 1
   ITOK = 1

   exitf = .FALSE.

                ikon = -1
      CALL CLEAR_MENU
      CALL set_font(ch12w9)
      CALL message(1,'PICK ATOM TO FIX',13)
      CALL FIND_ATOM_SELECT(ITOK,IKON,IDUM,IDUM,IDUM)
      CALL message(1,'PICK ATOM TO FIX',0)
      CALL CLEAR_MENU

      CALL get_severity(severity)
                CALL enter_atom_constraint(ikon,severity)

         END

SUBROUTINE FIND_ATOM_SELECT(ITOK,IKON,JKON,KKON,LKON)

include 'chmcm3.inc'
INTEGER IAT(4),ITOK,I,J,IKON,JKON,KKON,LKON,chce,num_opt
INTEGER*2 XPOS,YPOS

ikon = 0
            num_opt = 1
```

```
      do i = 1,4
         iat(i) = 0
      end do

      DO J = 1,ITOK
         DO WHILE (IAT(j).EQ.0)
            CALL CHMS(XPOS,YPOS)
            DO I = 1,NUMATS
               IF (XPOS.GT.(ISX(I) - 3).AND.
     1        XPOS.LT.(ISX(I) + 3).AND.
     2        YPOS.GT.(ISY(I) - 3).AND.
     3        YPOS.LT.(ISY(I) + 3)) THEN
                     CALL mark_atom(I,15)
                     IAT(J) = I
               END IF
            END DO
         END DO

         IKON = IAT(1)
         JKON = IAT(2)
         KKON = IAT(3)
         LKON = IAT(4)

         END

                  SUBROUTINE enter_atom_constraint(ikon,severity)

                  INTEGER ikon
                  REAL severity

         INCLUDE 'chmcm3.inc'

         atcons(ikon,1) = .TRUE.
C                   num_atm_fix =  num_atm_fix + 1
         fatxyz(ikon,1) = xo(ikon,1)
         fatxyz(ikon,2) = xo(ikon,2)
         fatxyz(ikon,3) = xo(ikon,3)
         fatsev(ikon) = severity * atsfac

                  END

                  SUBROUTINE length_constraint

      INCLUDE 'chmcm3.inc'
      INCLUDE 'commgrap.inc'

     LOGICAL exitf
                  INTEGER idum,ikon,jkon,itok
                  REAL fixed_val,severity

      itok = 2

      ikon = -1
      exitf = .FALSE.
          CALL CLEAR_MENU
          CALL message(1,
     1     'PICK ATOMS DEFINING FIXED LENGTH',13)
          CALL find_atom_select(itok,ikon,jkon,idum,idum)
          CALL message(1,
     1     'PICK ATOMS DEFINING FIXED LENGTH',0)
               CALL set_font(ch12w9)
               CALL message(1,
     1         'CHOOSE VALUE OF FIXED LENGTH',13)
                                          CALL atom_distance(ikon,jkon,fixed_val)
               CALL number_pad(fixed_val,idum,1)
               CALL set_font(ch12w9)
               CALL message(1,
     1         'CHOOSE VALUE OF FIXED LENGTH',0)
                 CALL get_severity(severity)
                                        CALL enter_length_constraint(ikon,jkon,
     1                                                fixed_val,severity)
            CALL mark_atom(ikon,15)
            STOP
            CALL mark_atom(jkon,12)

                 END



     SUBROUTINE enter_length_constraint(ikon,jkon,fixed_val,severity)

                  INTEGER ikon,jkon
                  REAL fixed_val,severity
```

```
   INCLUDE 'chmcm3.inc'


atcons(ikon,2) = .TRUE.
atcons(jkon,2) = .TRUE.
               numlfx = numlfx + 1
               klatm1(numlfx) = ikon
               klatm2(numlfx) = jkon
               fixed_val = ABS(fixed_val)
               fixlen(numlfx) = fixed_val
flnsev(numlfx) = severity

               END

SUBROUTINE angle_constraint


INCLUDE 'chmcm3.inc'
INCLUDE 'commgrap.inc'

 LOGICAL exitf
               INTEGER idum,ikon,jkon,kkon,itok
               REAL fixed_val,severity

 itok = 3
 ikon = -1
 exitf = .FALSE.
                    CALL CLEAR_MENU
      CALL message(1,
1          'PICK ATOMS DEFINING FIXED ANGLE',13)
      CALL find_atom_select(itok,ikon,jkon,kkon,idum)
      CALL message(1,
1          'PICK ATOMS DEFINING FIXED ANGLE',0)
          CALL set_font(ch12w9)
          CALL message(1,
1              'CHOOSE VALUE OF FIXED ANGLE',0)
                          CALL bond_angle(ikon,jkon,kkon,fixed_val)
          CALL number_pad(fixed_val,idum,1)
          CALL set_font(ch12w9)
          CALL message(1,
1              'CHOOSE VALUE OF FIXED ANGLE',0)
          CALL get_severity(severity)
                                      CALL enter_angle_constraint(ikon,jkon,kkon,
1                                                 fixed_val,severity)
                END

 SUBROUTINE enter_angle_constraint(ikon,jkon,kkon,fixed_val,
1                                                  severity)

               INTEGER ikon,jkon,kkon
               REAL fixed_val,severity

 INCLUDE 'chmcm3.inc'

               atcons(ikon,3) = .TRUE.
               atcons(jkon,3) = .TRUE.
               atcons(kkon,3) = .TRUE.
               numafx = numafx + 1
               kaatm1(numafx) = ikon
               kaatm2(numafx) = jkon
               kaatm3(numafx) = kkon
               fixed_val = ABS(fixed_val)
               fixang(numafx) = fixed_val
               fansev(numafx) = severity * ansfac

               END

 SUBROUTINE torsion_constraint

   INCLUDE 'chmcm3.inc'
   INCLUDE 'commgrap.inc'

 LOGICAL exitf
               INTEGER idum,ikon,jkon,kkon,lkon,itok
               REAL fixed_val,severity

 itok = 4
 ikon = -1
 exitf = .FALSE.
      CALL CLEAR_MENU
      CALL message(1,
1          'PICK ATOMS DEFINING FIXED TORSION ANGLE',13)
      CALL find_atom_select(itok,ikon,jkon,kkon,lkon)
      CALL message(1,
1          'PICK ATOMS DEFINING FIXED TORSION ANGLE',0)
          CALL set_font(ch12w9)
```

```
            CALL message(1,
     1           'CHOOSE FIXED VALUE OF TORSION ANGLE',13)
                          CALL torsion_angle(ikon,jkon,kkon,lkon,fixed_val)
            CALL number_pad(fixed_val,idum,1)
            CALL set_font(ch12w9)
            CALL message(1,
     1           'CHOOSE FIXED VALUE OF TORSION ANGLE',0)
            CALL get_severity(severity)

                                                                    CALL
enter_torsion_constraint(ikon,jkon,kkon,lkon,
     1                                          fixed_val,severity)

      END

      SUBROUTINE enter_torsion_constraint(ikon,jkon,kkon,lkon,
     1                                          fixed_val,severity)

            INTEGER ikon,jkon,kkon,lkon
            REAL fixed_val,severity

INCLUDE 'chmcm3.inc'

            atcons(ikon,4) = .TRUE.
            atcons(jkon,4) = .TRUE.
            atcons(kkon,4) = .TRUE.
            atcons(lkon,4) = .TRUE.
            numtfx = numtfx + 1
            ktatm1(numtfx) = ikon
            ktatm2(numtfx) = jkon
            ktatm3(numtfx) = kkon
            ktatm4(numtfx) = lkon
            fixtor(numtfx) = fixed_val
            ftosev(numtfx) = severity * TOSFAC

END

SUBROUTINE molecule_constraint

INCLUDE 'commgrap.inc'
INCLUDE 'chmcm3.inc'

LOGICAL exitf
            INTEGER idum,ikon,itok
 itok = 1

                        ikon = -1
      CALL CLEAR_MENU
      CALL message(1,
1          'PICK ANY ATOM IN MOLECULE TO FIX',13)
      CALL find_atom_select(itok,ikon,idum,idum,idum)
      CALL message(1,
1          'PICK ANY ATOM IN MOLECULE TO FIX',0)

                                    CALL enter_molecule_constraint(molnum(ikon))

            END

 SUBROUTINE enter_molecule_constraint(imol)

            INTEGER*1 imol

 INCLUDE 'commgrap.inc'
 INCLUDE 'chmcm3.inc'

 nummfx = nummfx + 1
 kmol(nummfx) = imol

            END


      SUBROUTINE atom_distance(i,j,dist)

      INTEGER i,j
      REAL dist

      INCLUDE 'chmcm3.inc'

      REAL dir1,dir2,dir3,dist2

      dir1 = xo(i,1) - xo(j,1)
      dir2 = xo(i,2) - xo(j,2)
      dir3 = xo(i,3) - xo(j,3)
      dist2 = (dir1 ** 2) + (dir2 ** 2) + (dir3 ** 2)
      dist = SQRT(dist2)

      END
```

```
      SUBROUTINE bond_angle(i,j,l,bangl)

      INTEGER i,j,l
      REAL bangl

      INCLUDE 'chmcm3.inc'

      REAL xb,yb,zb,dc11,dc12,dc21,dc22,dc31,dc32,rm1,rm2,r12,cosa,
     1     res
C
C    -- CALCULATE BOND ANGLE --
C
      xb = xo(j,1)
      yb = xo(j,2)
      zb = xo(j,3)
      dc11 = xo(i,1) - xb
      dc12 = xo(l,1) - xb
      dc21 = xo(i,2) - yb
      dc22 = xo(l,2) - yb
      dc31 = xo(i,3) - zb
      dc32 = xo(l,3) - zb
      rm1 = ((dc11 * dc11) + (dc21 * dc21)) + (dc31 * dc31)
      rm2 = ((dc12 * dc12) + (dc22 * dc22)) + (dc32 * dc32)
      r12 = ((dc11 * dc12) + (dc21 * dc22)) + (dc31 * dc32)
      rm1 = SQRT(rm1)
      rm2 = SQRT(rm2)
      res = r12 / (rm1 * rm2)
      cosa = SIGN(MIN(ABS(res),1.0),res)
      bangl = ACOS(cosa) * radi

      END

      SUBROUTINE torsion_angle(i,j,l,m,tangl)

      INTEGER i,j,l,m
      REAL tangl

      INCLUDE 'chmcm3.inc'

      REAL xj,yj,zj,xl,yl,zl,aa11,aa12,aa13,aa21,aa22,aa23,ta_sign,
     1     aa31,aa32,aa33,v11,v12,v21,v22,v13,v23,r1,r2,dotpr,cosw
C
C    -- CALCULATE TORSION ANGLE --
C
      xj = xo(j,1)
      yj = xo(j,2)
      zj = xo(j,3)
      xl = xo(l,1)
      yl = xo(l,2)
      zl = xo(l,3)
      aa11 = xo(i,1) - xj
      aa12 = xl - xj
      aa13 = xl - xo(m,1)
      aa21 = xo(i,2) - yj
      aa22 = yl - yj
      aa23 = yl - xo(m,2)
      aa31 = xo(i,3) - zj
      aa32 = zl - zj
      aa33 = zl - xo(m,3)
      v11 = (aa21 * aa32) - (aa31 * aa22)
      v21 = (aa22 * aa33) - (aa32 * aa23)
      v12 = (aa31 * aa12) - (aa11 * aa32)
      v22 = (aa32 * aa13) - (aa12 * aa33)
      v13 = (aa11 * aa22) - (aa21 * aa12)
      v23 = (aa12 * aa23) - (aa22 * aa13)
      r1 = SQRT(((v11 * v11) + (v12 * v12)) + (v13 * v13))
      r2 = SQRT(((v21 * v21) + (v22 * v22)) + (v23 * v23))
      dotpr = (((v11 / r1) * (v21 / r2)) +
     1         ((v12 / r1) * (v22 / r2))) +
     2         ((v13 / r1) * (v23 / r2))
      cosw = SIGN(MIN(ABS(dotpr),1.0),dotpr)
      tangl = ACOS(cosw) * RADI
C
C    -- CALCULATE CORRECT SIGN FOR TORSION ANGLE --
C
      ta_sign = ((aa11 * ((aa22 * aa33) - (aa23 * aa32))) -
     1           (aa21 * ((aa12 * aa33) - (aa13 * aa32)))) +
     2           (aa31 * ((aa12 * aa23) - (aa13 * aa22)))
      tangl = SIGN(tangl,ta_sign)

      END

        SUBROUTINE get_severity(severity)

        REAL severity

        INCLUDE 'commgrap.inc'
```

```
            INTEGER idum

CALL set_font(ch12w9)

CALL message(1,'CHOOSE SEVERITY OF CONSTRAINT',13)
CALL number_pad(severity,idum,0)
IF (severity .LT. 10.0) THEN
    severity = 10.0
ELSE IF (severity .GT. 3000.0) THEN
    severity = 3000.0
END IF
CALL set_font(ch12w9)
CALL message(1,'CHOOSE SEVERITY OF CONSTRAINT',0)

END

SUBROUTINE fill_circle (x1,y1,x2,y2)

INCLUDE 'fgraph.FD'

INTEGER*2 dummy
INTEGER x1,y1,x2,y2

dummy = ellipse($GFILLINTERIOR,x1,y1,x2,y2)

END

SUBROUTINE mark_atom(atm_num,c_colour)

INCLUDE 'chmcm3.inc'

INTEGER atm_num,x1,y1,x2,y2,c_colour,i

x1 = isx(atm_num) - 3
y1 = isy(atm_num) - 3
x2 = isx(atm_num) + 3
y2 = isy(atm_num) + 3


CALL xorstyle

call colour(c_colour)

      call fill_circle(x1,y1,x2,y2)

                  CALL presetstyle

      END

      SUBROUTINE CLEAR_MENU

      INCLUDE 'COMMGRAP.INC'

      CALL COLOUR(0)

                CALL FILL_BOX (0 - MENU_SPACEX,0,
     1            0,MENU_SPACEY)

      END

      SUBROUTINE GRAPHICSMODE()

      INCLUDE 'FGRAPH.FD'

            INTEGER * 2 MODESTATUS

C     SET VIDEOMODE TO MAXRESOLUTION
      MODESTATUS = SETVIDEOMODE($MAXRESMODE)
      IF (MODESTATUS.EQ.0) STOP 'ERROR :
     1       CANNOT SET GRAPHICS MODE'

      CALL CLEARSCREEN($GCLEARSCREEN)

            END


      SUBROUTINE ENDGRAPHICS()

      INCLUDE 'FGRAPH.FD'

      INTEGER * 2 MODESTATUS

      MODESTATUS = SETVIDEOMODE($DEFAULTMODE)

      END
```

```
SUBROUTINE MOUSE

INCLUDE 'c:\lesley\MOUSE.FD'
INCLUDE 'FGRAPH.FD'

INTEGER START
INTEGER*2 BUTTONS

CALL COLOUR(7)
START = INITIALISEMOUSE(BUTTONS)

IF (START.EQ.0) THEN
    CALL ENDGRAPHICS()
    WRITE(*,*) 'MOUSE DRIVER NOT INSTALLED'
END IF

CALL SHOWMOUSECURSOR()

      END

              SUBROUTINE set_font(type)

      INCLUDE 'FGRAPH.FD'

              INTEGER * 2 STATUS

              INTEGER type

C     SET FONTS

              IF (type.eq.1) THEN
         STATUS = SETFONT("T'COURIER'"//'h12w9b')
              ELSE IF (type.eq.2) THEN
         STATUS = SETFONT("T'COURIER'"//'h15w12b')
      ELSE
         STATUS = SETFONT("T'COURIER'"//'h10w8b')
              END IF

      IF (STATUS.LT.0) THEN
          STOP 'ERROR:CANNOT SET FONT'
      END IF

      END

              SUBROUTINE register_fonts

      INCLUDE 'FGRAPH.FD'

              INTEGER * 2 DUMMY

              INTEGER type

C     SET FONTS

      DUMMY = REGISTERFONTS('C:\MSF\LIB\*.FON')

      IF (DUMMY.LT.0) THEN
          STOP 'ERROR:CANNOT FIND FONT FILES'
      END IF

      END

              SUBROUTINE text_info(textwd,textht)

              INCLUDE 'fgraph.fd'

              RECORD /fontinfo/myfont

              INTEGER*2 status
              INTEGER textwd,textht

              status = GETFONTINFO(myfont)

              IF (STATUS.NE.0) THEN
                 STOP 'ERROR:CANNOT FIND FONT CHARACTERISTICS'
              END IF

              textht = myfont.pixheight
              textwd = myfont.pixwidth

              write(10,*) textwd,textht

              END

SUBROUTINE get_screen_coords (maxx,maxy)

              INTEGER maxx,maxy
```

```
            INCLUDE 'FGRAPH.FD'

RECORD/VIDEOCONFIG/MYSCREEN

CALL GETVIDEOCONFIG(MYSCREEN)

MAXX = MYSCREEN.NUMXPIXELS - 1
MAXY = MYSCREEN.NUMYPIXELS - 1

            END

            SUBROUTINE convert_to_viewport_coords(XPOS,YPOS)

            INTEGER*2 XPOS,YPOS

            INCLUDE 'FGRAPH.FD'

            RECORD /XYCOORD/viewport

                  CALL getviewcoord(XPOS,YPOS,viewport)

                  XPOS = viewport.xcoord
                  YPOS = viewport.ycoord

                  END

      SUBROUTINE init_screen_area(screen_type)

      INCLUDE 'commgrap.inc'

      INTEGER txtwd,txtht,xtp,ytp,screen_type

      CALL get_screen_coords(xtp,ytp)

C     set message area at top of screen

      CALL set_font(ch12w9)
      CALL text_info(txtwd,txtht)

      message_spacey = ((no_of_messages*txtht) +
     1          ((no_of_messages - 1) * txtht/2))

C     Set menu area at side of screen

      IF (screen_type.eq.2) THEN
          menu_spacex = (MAX_LENGTH_MENU_STRING*txtwd) +
     1          txtwd
      ELSE
           call set_font(ch15w12)
           CALL text_info(txtwd,txtht)
           menu_spacex = (7 * txtwd) + txtwd
      END IF

      menu_spacey = ytp - message_spacey

C     set display area

      CALL set_display_area(menu_spacex,message_spacey)

C     calculate centre coordiantes of display area

                  viewport_xspace = xtp - menu_spacex
                  viewport_yspace = ytp - message_spacey

                    x_viewport_cent = (viewport_xspace)/2
      y_viewport_cent = (viewport_yspace)/2

      END

      SUBROUTINE set_display_area(xshift,yshift)

      include 'FGRAPH.FD'

      INTEGER xshift,yshift

      RECORD /xycoord/org

      CALL setvieworg(xshift,yshift,org)

      END

      SUBROUTINE MESSAGE(ROW,STRING,TEXT_COLOUR)

            INCLUDE 'commgrap.inc'

            INTEGER row,mposx,mposy,text_colour,txtwd,txtht,xtp,ytp,
     1      centre_pixel,length,length_pixels,centre_string
```

```
                CHARACTER*(*) string

                IF (ROW.GT.no_of_messages) THEN
                   STOP 'ERROR: NO SPACE FOR MESSAGE'
                END IF

                CALL text_info (txtwd,txtht)

                MESPOSY = 0 - message_spacey + (row * (txtht + txtht/2))

                IF (row.eq.0) THEN
                   CALL get_screen_coords(xtp,ytp)
                       centre_pixel = INT((xtp + 1)/2)
                       length = LEN_TRIM(string)
                       length_pixels = length * txtwd
                       centre_string = INT(length_pixels/2)
                       MESPOSX = 0 - menu_spacex + centre_pixel - centre_string
ELSE
                   MESPOSX = 0 - menu_spacex
                END IF

                call colour (TEXT_COLOUR)
                call text (mesposx,mesposy,string)

    END

                SUBROUTINE init_menu_data(num_opt)

    INCLUDE 'commgrap.inc'
    INCLUDE 'commmenu.inc'

    INTEGER textwd,textht,text_spacey,free_spacey,
1           free_spacey2,tlen,num_opt,
2           total_length_text,length_half_text,
3           free_spacex,free_spacex2,i

    CHARACTER*20 menu_text(20)

                  CALL set_font(ch12w9)
    CALL text_info (textwd,textht)

    text_spacey = num_opt * textht
                  free_spacey = (menu_spacey - text_spacey)/
1                   num_opt + 1
    free_spacey = MIN(free_spacey,(3 * (textht/2)))
    free_spacey2 = free_spacey/3

                tlen = LEN(menu_text(1))
    total_length_text = tlen * textwd
    length_half_text = total_length_text/2
                   free_spacex = (menu_spacex - total_length_text)/2
    free_spacex2 = free_spacex/3


    DO I = 1,num_opt
       xpos_text(i) = (0 - menu_spacex) + free_spacex
       ypos_text(i) = (((I-1) * (textht + free_spacey)) +
1              free_spacey)
       menu_obx1(i) = xpos_text(i) - free_spacex2
       menu_obxr(i) = menu_obx1(i) + (length_half_text +
1               free_spacex2) * 2
       menu_obyt(i) = ypos_text(i) - free_spacey2
       menu_obyb(i) = menu_obyt(i) + (textht/2 +
1              free_spacey2) * 2

       middle_text(i) = xpos_text(i) + length_half_text
    END DO

    END

                SUBROUTINE display_fix_menu(num_opt)

                INTEGER num_opt
                CHARACTER*20 menu_text(20)

                menu_text(1) = 'FIX ATOM'
menu_text(2) = 'FIX DISTANCE'
menu_text(3) = 'FIX VALENCE ANGLE'
menu_text(4) = 'FIX TORSION ANGLE'
menu_text(5) = 'FIX MOLECULE'
menu_text(6) = 'DONE'

                CALL draw_menu(num_opt,menu_text)

                END

                SUBROUTINE draw_menu(num_opt,menu_text)
```

```
      INCLUDE 'commmenu.inc'
                INCLUDE 'commgrap.inc'

      INTEGER num_opt,len,len2,textwd,textht,len2_pixels
      CHARACTER*(*) menu_text(*)

      DO I = 1,num_opt
         IF (menu_text(i).EQ.'DONE') THEN
            CALL COLOUR(done_box_colour)
         ELSE
            CALL colour(box_menu_colour)
         END IF
         CALL BOX(menu_obxl(i),menu_obyt(i),
     1     menu_obxr(i),menu_obyb(i))

         IF (menu_text(i).EQ.'DONE') THEN
            CALL COLOUR(done_text_colour)
         ELSE
            CALL colour(text_menu_colour)
         END IF

C        calculate start position of centralised text

         len = LEN_TRIM(menu_text(i))
         len2 = len/2
                        call text_info(textwd,textht)
         len2_pixels = len2 * textwd
         xpos_text(i) = middle_text(i) - len2_pixels
         CALL TEXT(xpos_text(i),ypos_text(i),menu_text(i))
      END DO

      END

      SUBROUTINE get_mol_screen_coords

      INCLUDE 'chmcm3.inc'
                INCLUDE 'commgrap.inc'

      REAL xcent,ycent
      COMMON /SCCDCN/ xcent,ycent

      REAL xleft,xright,ylow,yhigh,xt,yt,
     1     xsize,ysize,rxl,rxr,ryl,ryh,rzf,rzb,size_mean,
     2     xot,yot
      INTEGER i,iatomxl,iatomxr,iatomyl,iatomyh

         DO i = 1,numats
                     xt = xo(i,1)
                     yt = xo(i,2)
                     IF (xt .LT. xleft) THEN
                         xleft = xt
                         iatomxl = i
                     END IF
                     IF (xt .GT. xright) THEN
                         xright = xt
                         iatomxr = i
                     END IF
                     IF (yt .LT. ylow) then
                         ylow = yt
                         iatomyl = i
                     END IF
                     IF (yt .GT. yhigh) THEN
                         yhigh = yt
                         iatomyh = i
                     END IF
         END DO
         xsize = xright - xleft + 0.2
         ysize = yhigh - ylow + 0.2
         xcent = xleft + (xsize / 2.0)
         ycent = ylow + (ysize / 2.0)
         size_mean = (xsize * xsize) + (ysize * ysize)
         size_mean = SQRT(size_mean)
         IF (size_mean .GT. 0.001) THEN
             scale_ang_screen = FLOAT(MIN(viewport_xspace,
     1                    viewport_yspace))/size_mean
         ELSE
             scale_ang_screen = 0.0
         END IF
C
C    -- CALCULATE SCREEN COORDINATES AFTER CENTERING MOLECULE ON ORIGIN
C    -- OF USER COORDINATES. --
C

         DO i = 1,numats
             xot = xo(i,1) - xcent
             yot = xo(i,2) - ycent
             zot = xo(i,3) - zcent
```

```
      isx(i) = INT(xot * scale_ang_screen) + x_viewport_cent
      isy(i) = INT(yot * scale_ang_screen) + y_viewport_cent
END DO

END

SUBROUTINE draw_simple_stick_molecule

INCLUDE 'chmcm3.inc'
              INCLUDE 'commgrap.inc'
              INCLUDE 'fgraph.FD'

INTEGER bnd_cnt,point(6),idelx,idely,iacol,colour_atom
LOGICAL drawf,hydro



CALL CLEARSCREEN($GCLEARSCREEN)
                hydro = .TRUE.


                CALL init_atom_colours

    DO i = 1,numats
                        IF (atynum(i) .GT. MAXTYP) THEN
          CONTINUE
        ELSE IF ((.NOT. hydro) .AND.
1                               (atynum(i) .LE. Hh)) THEN
          CONTINUE
        ELSE
                                        bnd_cnt = 0
          DO j = 1,MXCN
              drawf = .FALSE.
              iconnij = atmcon(i,j)
              IF (iconnij .EQ. 0) THEN
                  bnd_cnt = bnd_cnt + 1
              ELSE IF ((.NOT. hydro) .AND.
1                 (atynum(iconnij) .LE. Hh)) THEN
                  bnd_cnt = bnd_cnt + 1
              ELSE IF (iconnij .GT. i) THEN
                  drawf = .TRUE.
              END IF
              IF (drawf) THEN
                  idelx = isx(iconnij) - isx(i)
                  idely = isy(iconnij) - isy(i)
                  point(1)=isx(i)
                  point(2)=isy(i)
                  point(3)=isx(i) + (idelx / 2)
                  point(4)=isy(i) + (idely / 2)
                  point(5)=isx(iconnij)
                  point(6)=isy(iconnij)

                  iacol = colour_atom(atynum(i))
                                        CALL colour(iacol)
                  CALL draw_line (point(1))

                  iacol = colour_atom(atynum(iconnij))
                                        CALL colour(iacol)
                  CALL draw_line (point(3))
                END IF
          END DO
C             IF (bnd_cnt .EQ. MXCN) THEN
C                 CALL get_z_int(isz(i),ibint)
C                 CALL setfcr(atom_color(atynum(i),ibint))
C                 CALL filcir(isx(i),isy(i),
C     1                       INT(MAX(4.0,0.1 * scale_ang_screen))
C             END IF
          END IF
    END DO

    END

              SUBROUTINE draw_atom_numbers(col)

              INCLUDE 'chmcm3.inc'
    INCLUDE 'commgrap.inc'
    CHARACTER*2 atom_number
    INTEGER textwd,textht,i,col,posx_atom_number,
1           posy_atom_number


              CALL set_font(ch12w9)

              CALL text_info(textwd,textht)

    DO I = 1,NUMATS
          posx_atom_number = isx(i) - textwd - (textwd/2)
          posy_atom_number = isy(i)
```

```
          WRITE(atom_number,'(I2)') I
                                    CALL colour(col)
                                    CALL text (posx_atom_number,posy_atom_number,
1               atom_number)
      END DO

      END

                SUBROUTINE init_atom_colours

                INCLUDE 'chmcm3.inc'
                INCLUDE 'commgrap.inc'

                DO I = 1,MAXTYP
       IF ((I.EQ.H).OR.(I.EQ.Har).OR.(I.EQ.Hh)) THEN
                                          atom_colours(I) = 15
                          ELSE IF ((I.EQ.Csp3).OR.(I.EQ.Csp2).OR.(I.EQ.Car)) THEN
                atom_colours(I) = 2
      ELSE IF ((I.EQ.Nsp).OR.(I.EQ.Namide).OR.(I.EQ.Ncation).OR.
1               (I.EQ.Nar)) THEN
                atom_colours(I) = 9
      ELSE IF ((I.EQ.Osp3).OR.(I.EQ.Osp2).OR.(I.EQ.Oanion)) THEN
                atom_colours(I) = 12
      ELSE IF ((I.EQ.F).OR.(I.EQ.Cl).OR.(I.EQ.Br).OR.
1               (I.EQ.Iod)) THEN
                atom_colours(I) = 11
                        ELSE IF (I.EQ.Piii) THEN
                                             atom_colours(I) = 13
      ELSE IF ((I.EQ.Sii).OR.(I.EQ.Siii)) THEN
           atom_colours(I) = 14
      ELSE IF ((I.EQ.Mg2).OR.(I.EQ.Ca2).OR.(I.EQ.Ba2).OR.
1               (I.EQ.Fe2).OR.(I.EQ.Fe3).OR.(I.EQ.Cu1).OR.
2               (I.EQ.Cu2).OR.(I.EQ.MET).OR.(I.EQ.MET1)) THEN
           atom_colours(I) = 8
      ELSE
           atom_colours(I) = 6
      END IF

    END DO

    END

                  INTEGER FUNCTION colour_atom(atom_type_num)
INCLUDE 'chmcm3.inc'
BYTE atom_type_num

colour_atom = atom_colours(atom_type_num)

                RETURN
END

                SUBROUTINE draw_line(points)

                INCLUDE 'FGRAPH.FD'

                RECORD /xycoord/xy

                INTEGER*2 status
                INTEGER points(4)

                CALL moveto(points(1),points(2),xy)
                status = lineto(points(3),points(4))

                END

                SUBROUTINE BOX(x1,y1,x2,y2)

                INTEGER x1,y1,x2,y2

              INCLUDE 'FGRAPH.FD'

                        INTEGER *2 STATUS

C         DRAWBOX

              STATUS = RECTANGLE($GBORDER,x1,y1,x2,y2)

              END

                        SUBROUTINE FILL_BOX(x1,y1,x2,y2)

                        INTEGER x1,y1,x2,y2

              INCLUDE 'FGRAPH.FD'

                        INTEGER *2 STATUS
```

```
C       DRAWBOX

          STATUS = RECTANGLE($GFILLINTERIOR,x1,y1,x2,y2)

          END

          SUBROUTINE TEXT(XCOORD,YCOORD,STRING)

          INCLUDE 'FGRAPH.FD'

          RECORD/XYCOORD/XY
          INTEGER XCOORD,YCOORD
          CHARACTER*(*) STRING

          CALL MOVETO(XCOORD,YCOORD,XY)
          CALL OUTGTEXT(STRING)

          END
```

```
C       ****SET COLOUR****

          SUBROUTINE COLOUR(C)

          INCLUDE 'FGRAPH.FD'


                    INTEGER C
          INTEGER*2 PICK

          PICK = SETCOLOR(C)

          IF (PICK.EQ.-1) THEN
             STOP 'ERROR:CANNOT SET COLOUR'
          END IF

          END

          SUBROUTINE XORSTYLE

                              IMPLICIT NONE

          INCLUDE 'FGRAPH.FD'

          INTEGER*2 STYLE
          STYLE = SETWRITEMODE($GXOR)

          IF (STYLE.EQ.-1) THEN
             STOP 'ERROR:CANNOT SET WRITEMODE'
          END IF

          END

          SUBROUTINE PRESETSTYLE

          INCLUDE 'FGRAPH.FD'

          INTEGER*2 STYLE
          STYLE = SETWRITEMODE($GPSET)

          IF (STYLE.EQ.-1) THEN
             STOP 'ERROR:CANNOT SET WRITEMODE'
          END IF

          END
```

Appendix D:

Photographs.

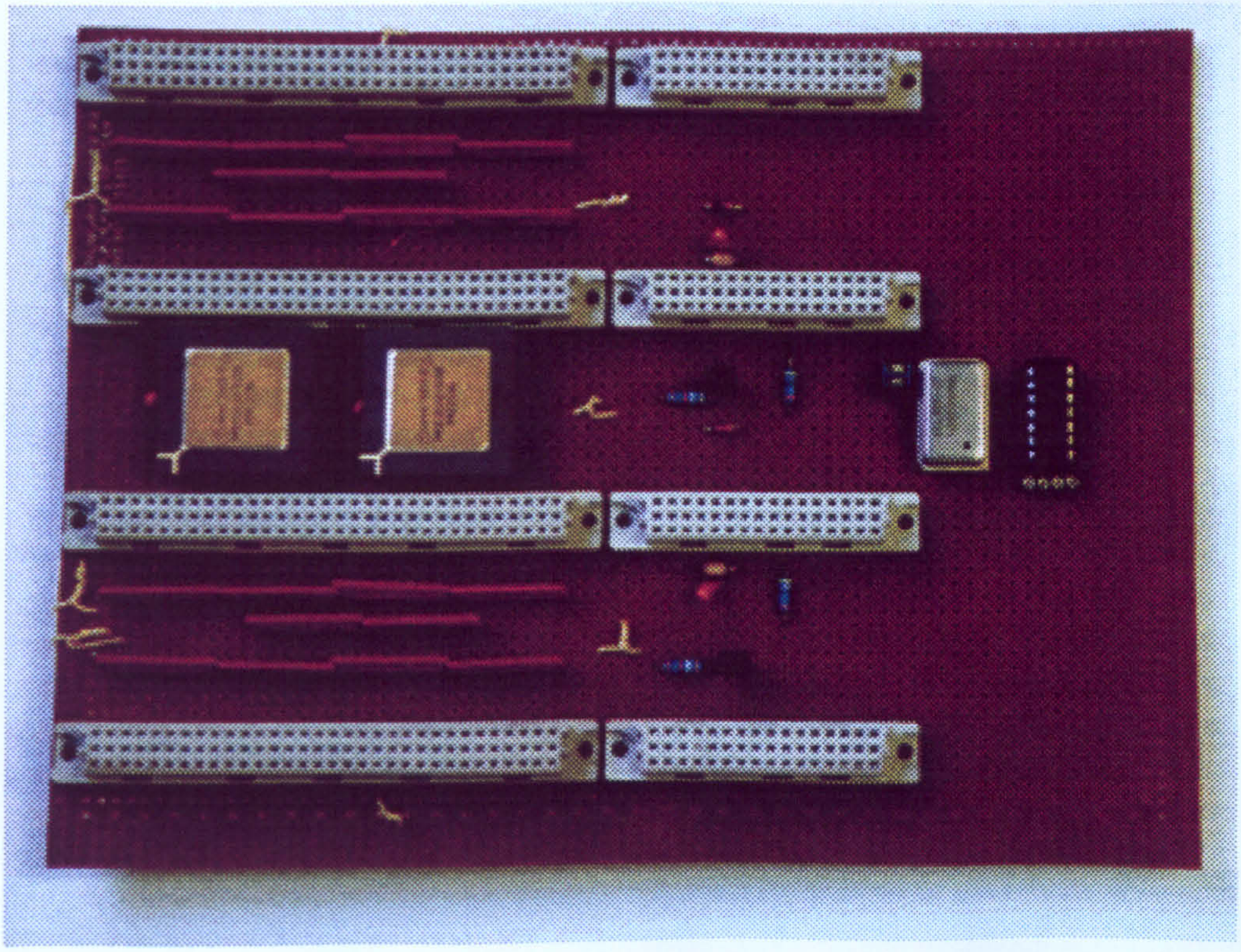


FIGURE 1. Switch Board

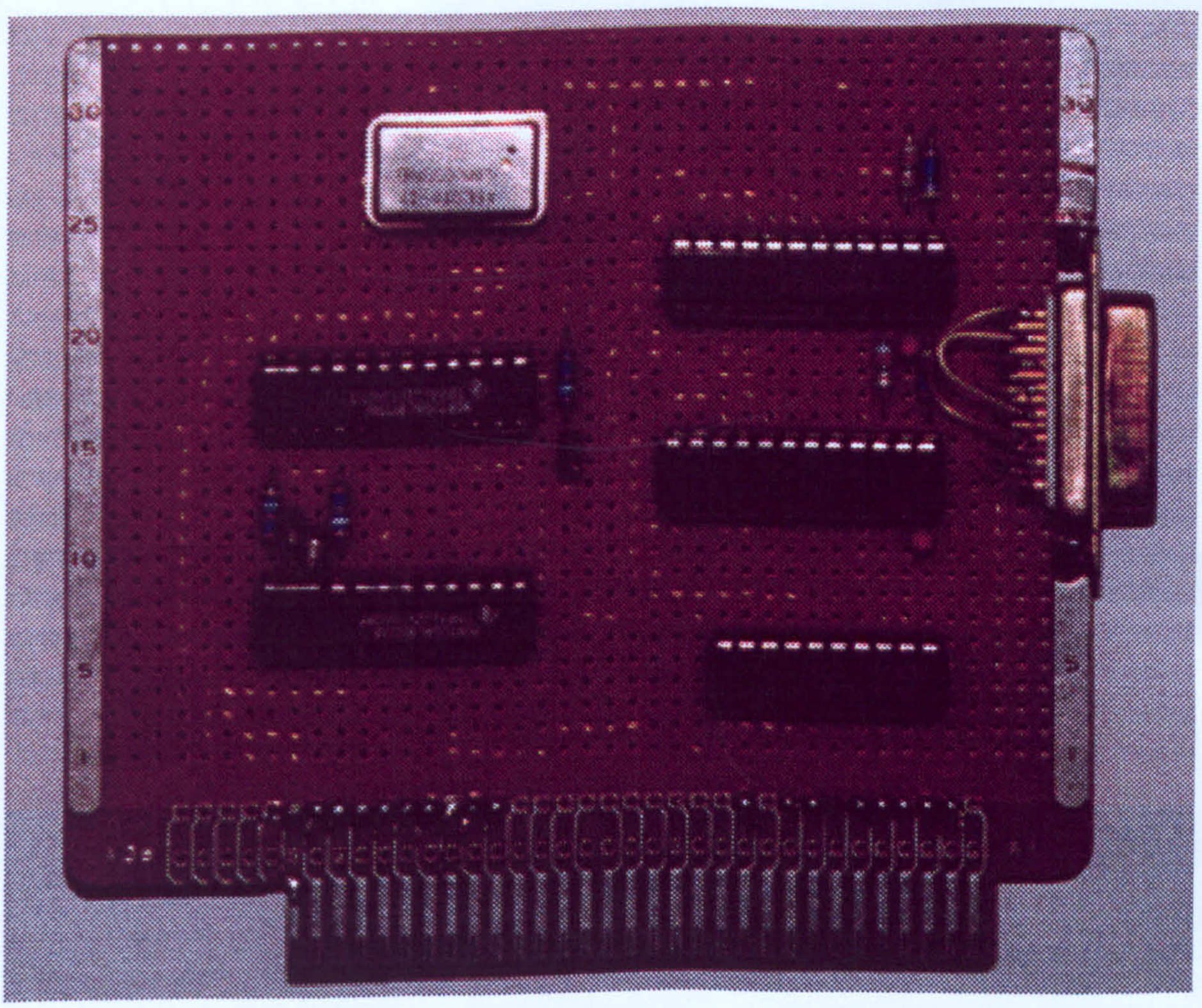


FIGURE 2. Dual Link Adapter Board

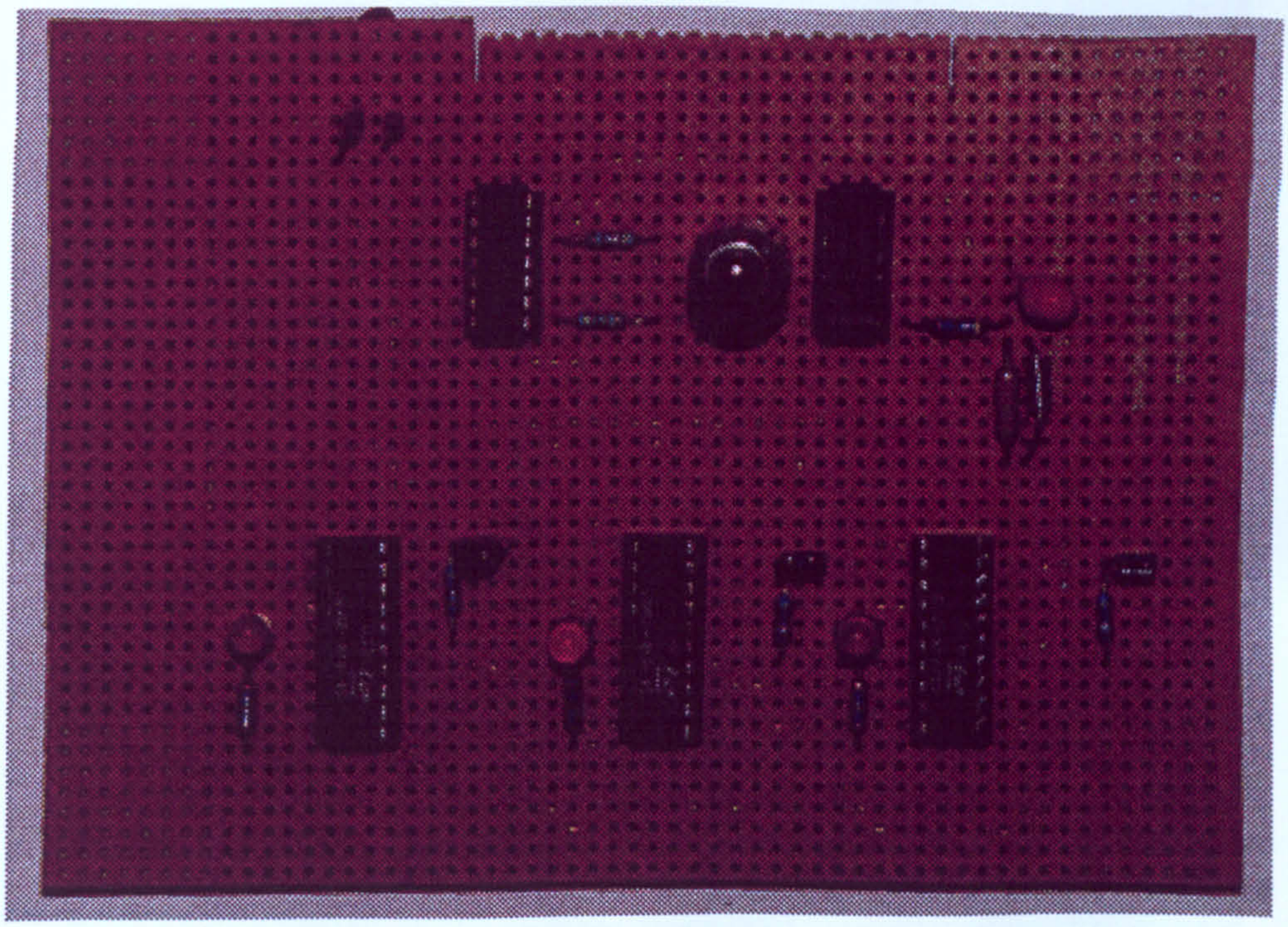


FIGURE 3. Token Passing Test Circuit

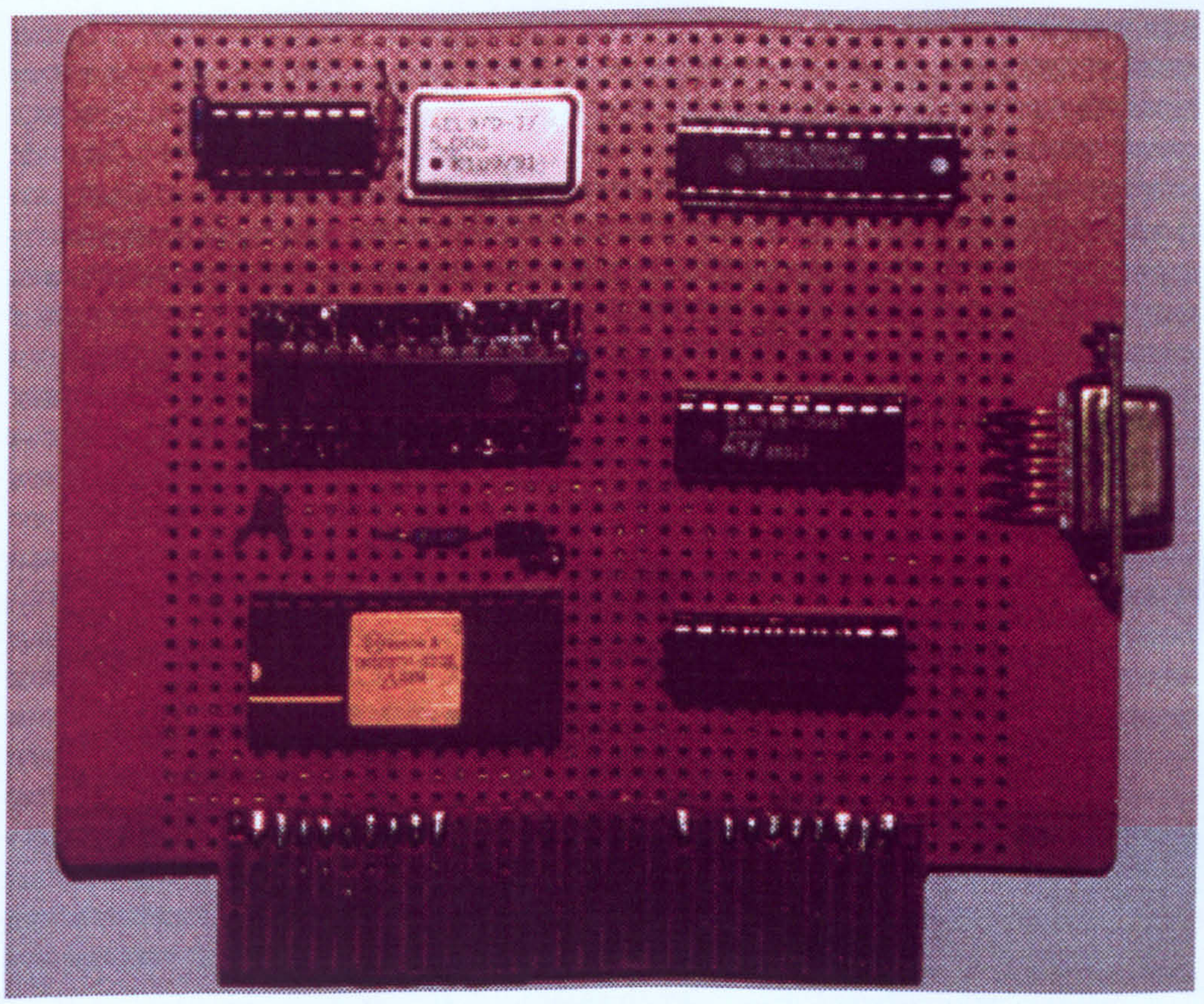


FIGURE 4. FIFO Clocking Test Circuit

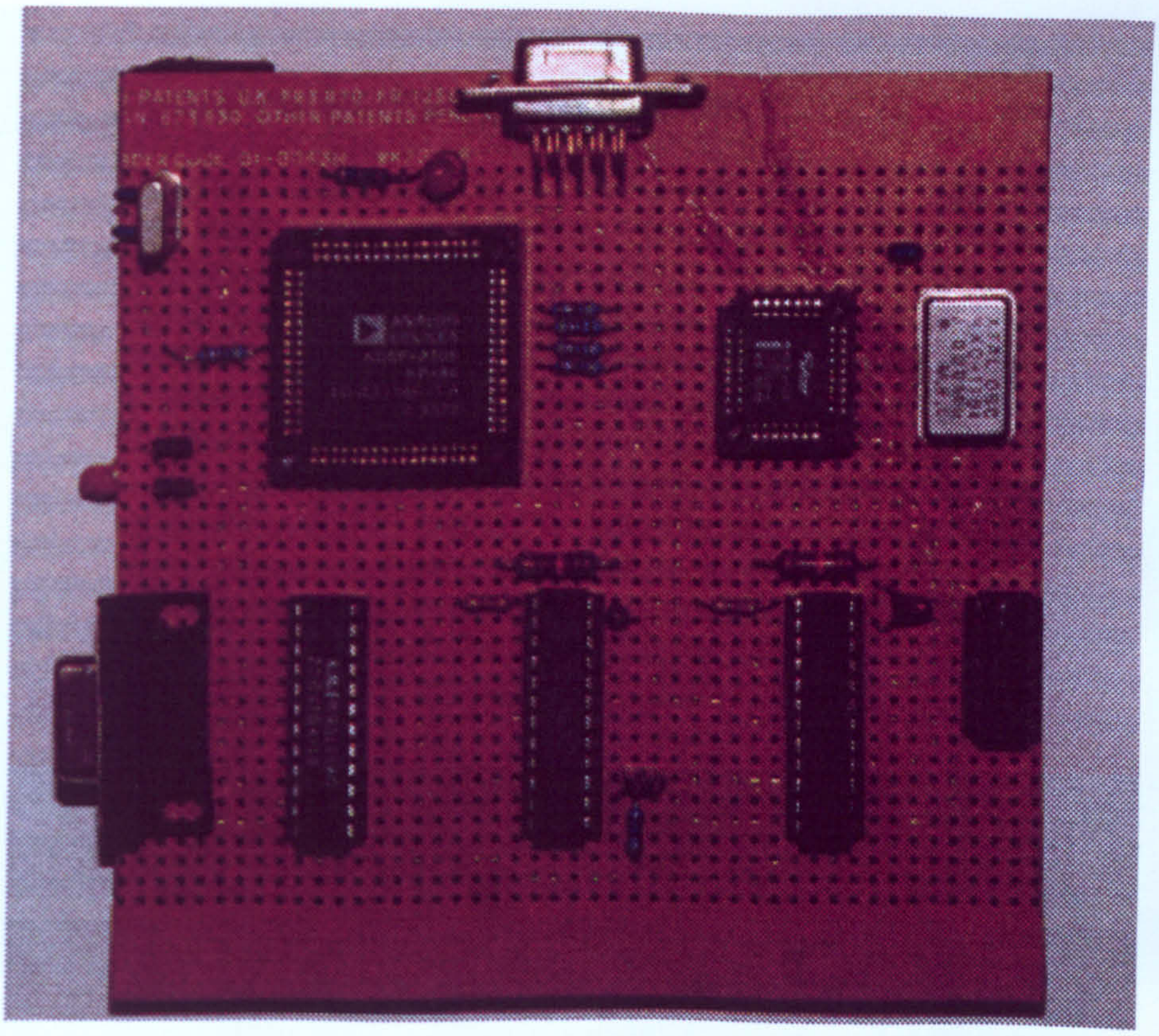


FIGURE 5. Control Processor Board

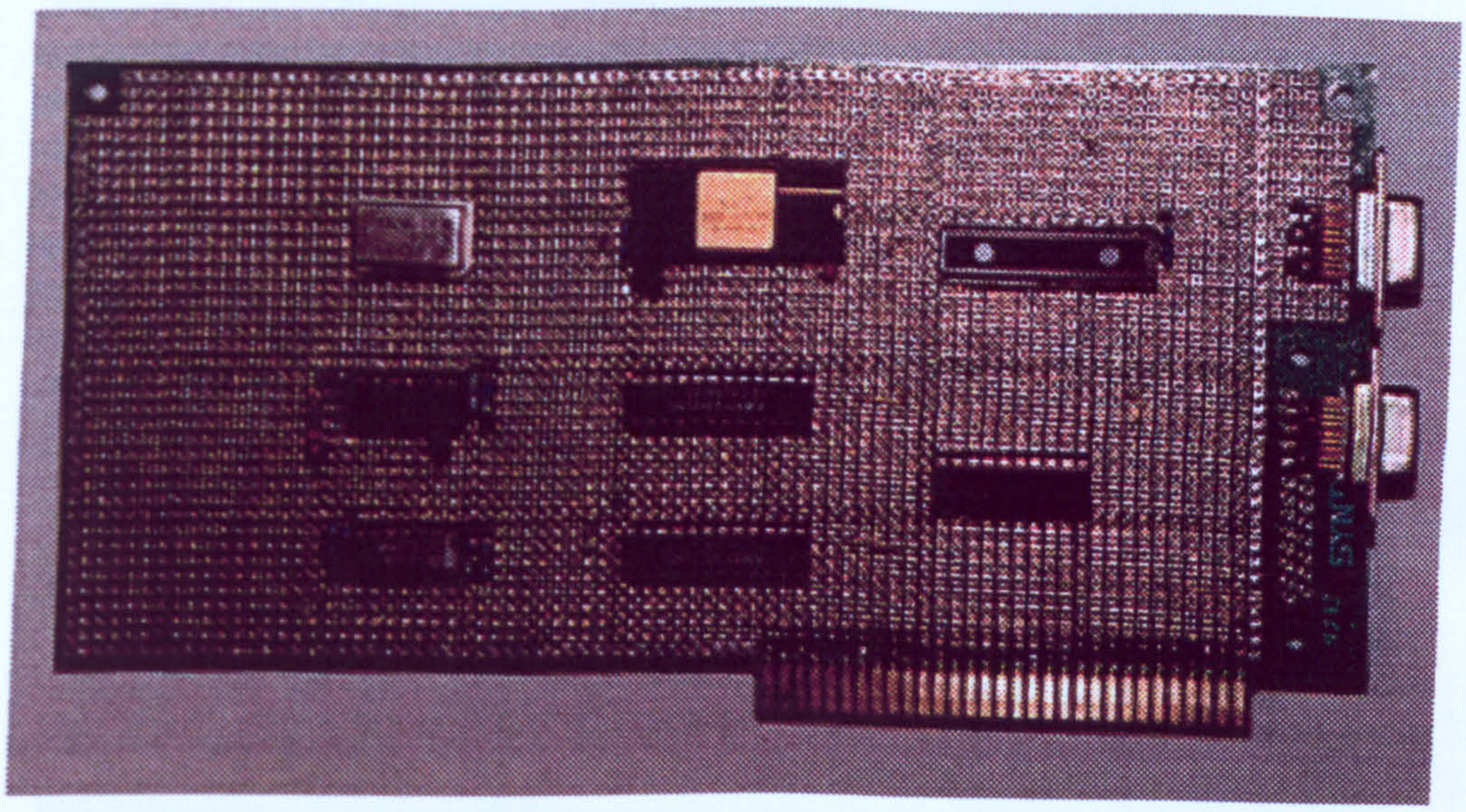


FIGURE 6. PC plug-in card to emulate node

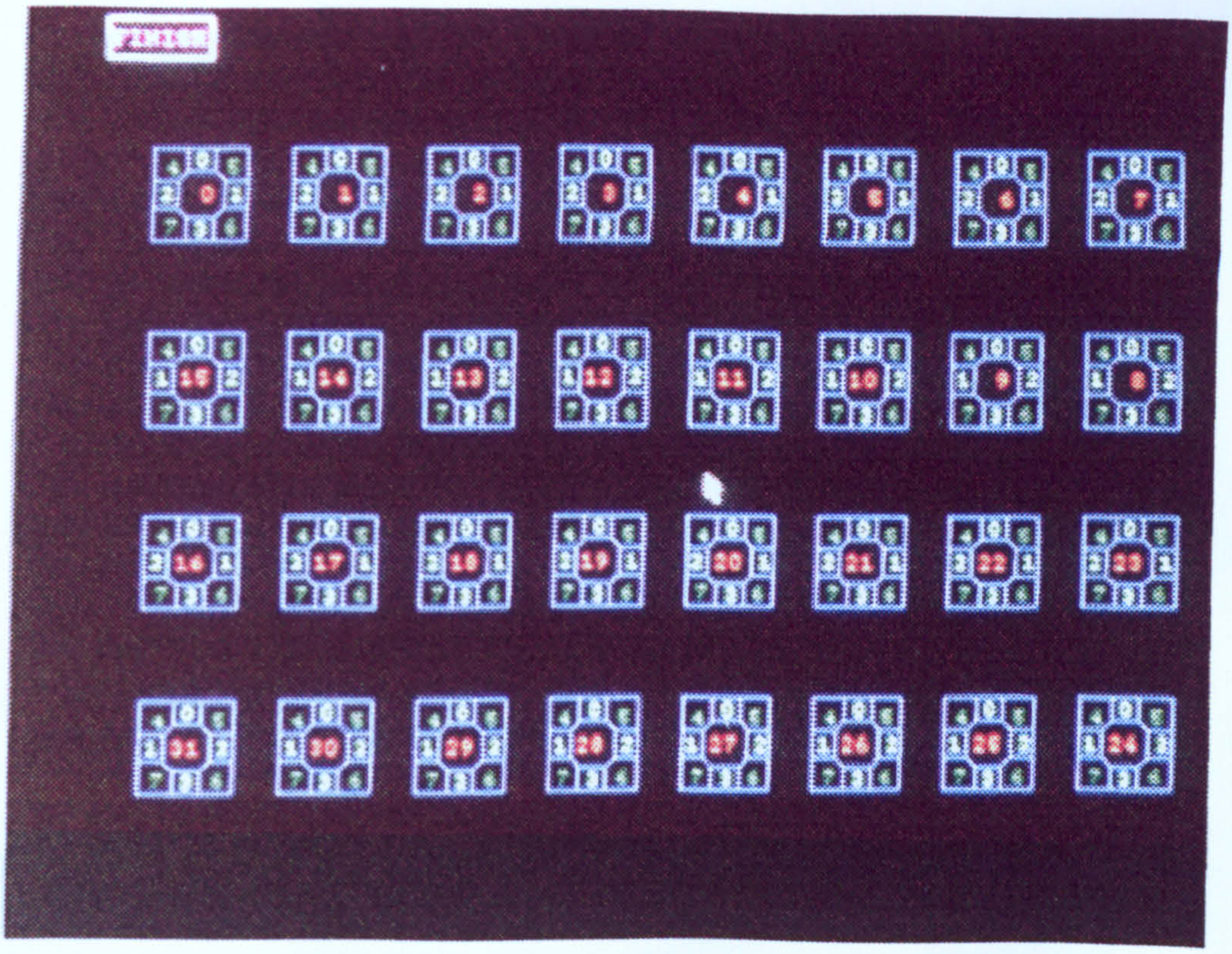


FIGURE 7. Graphical Interface allowing connections between nodes

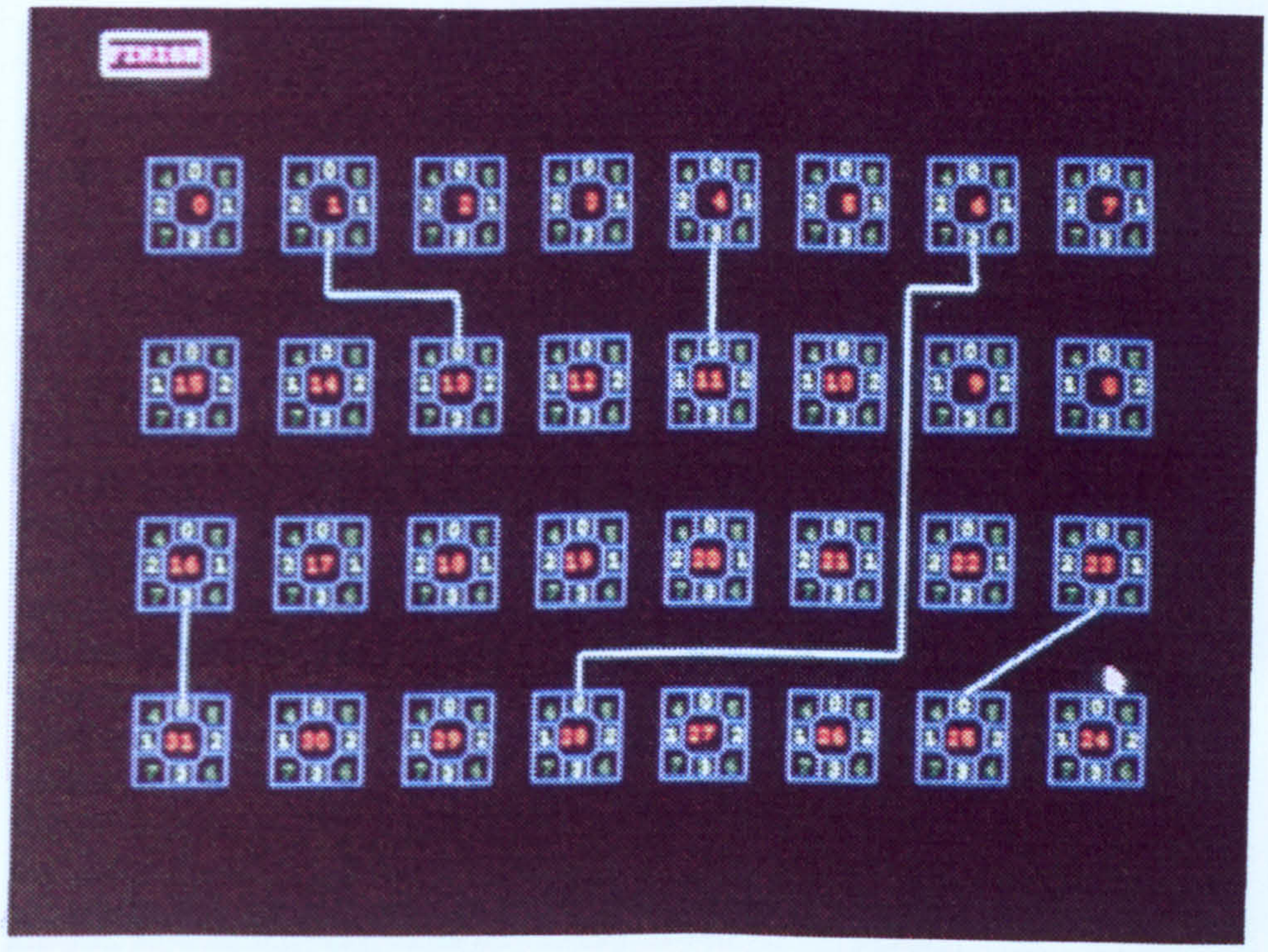


FIGURE 8. Graphical Interface showing connections between nodes

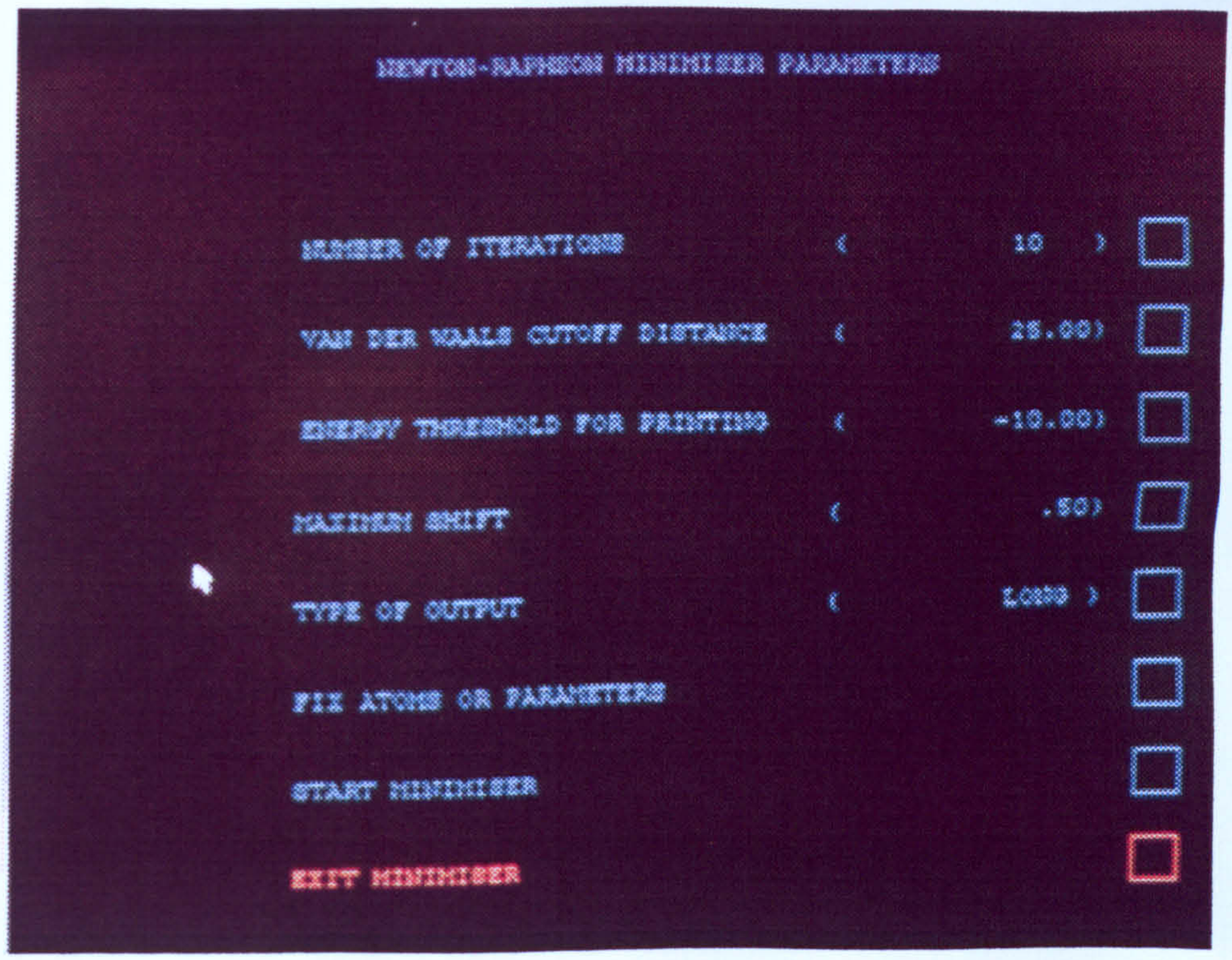


FIGURE 9. Initial Screen of minimiser

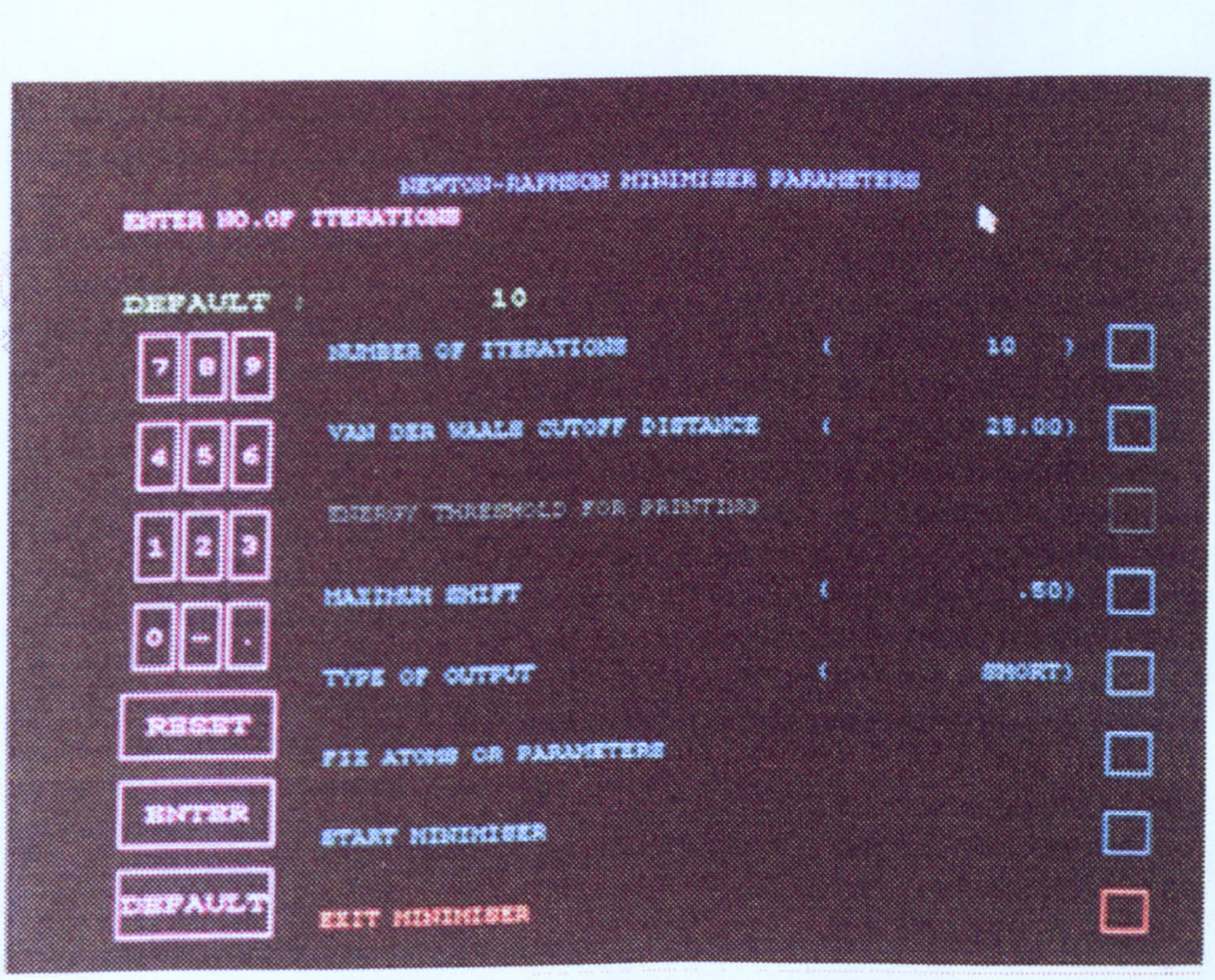


FIGURE 10. Number Pad allowing user to enter number of iterations

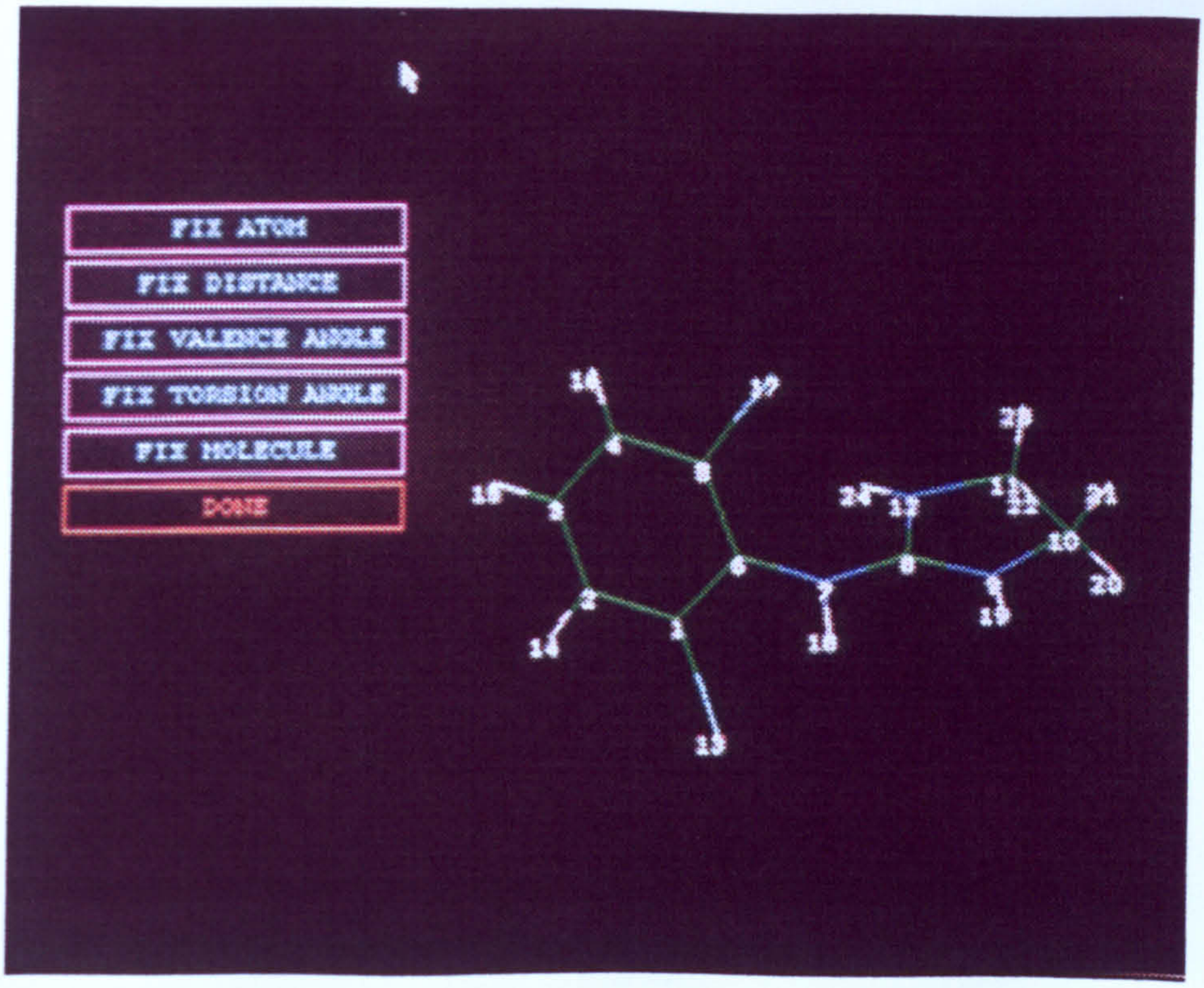


FIGURE 11. Screen allowing user to fix parameters

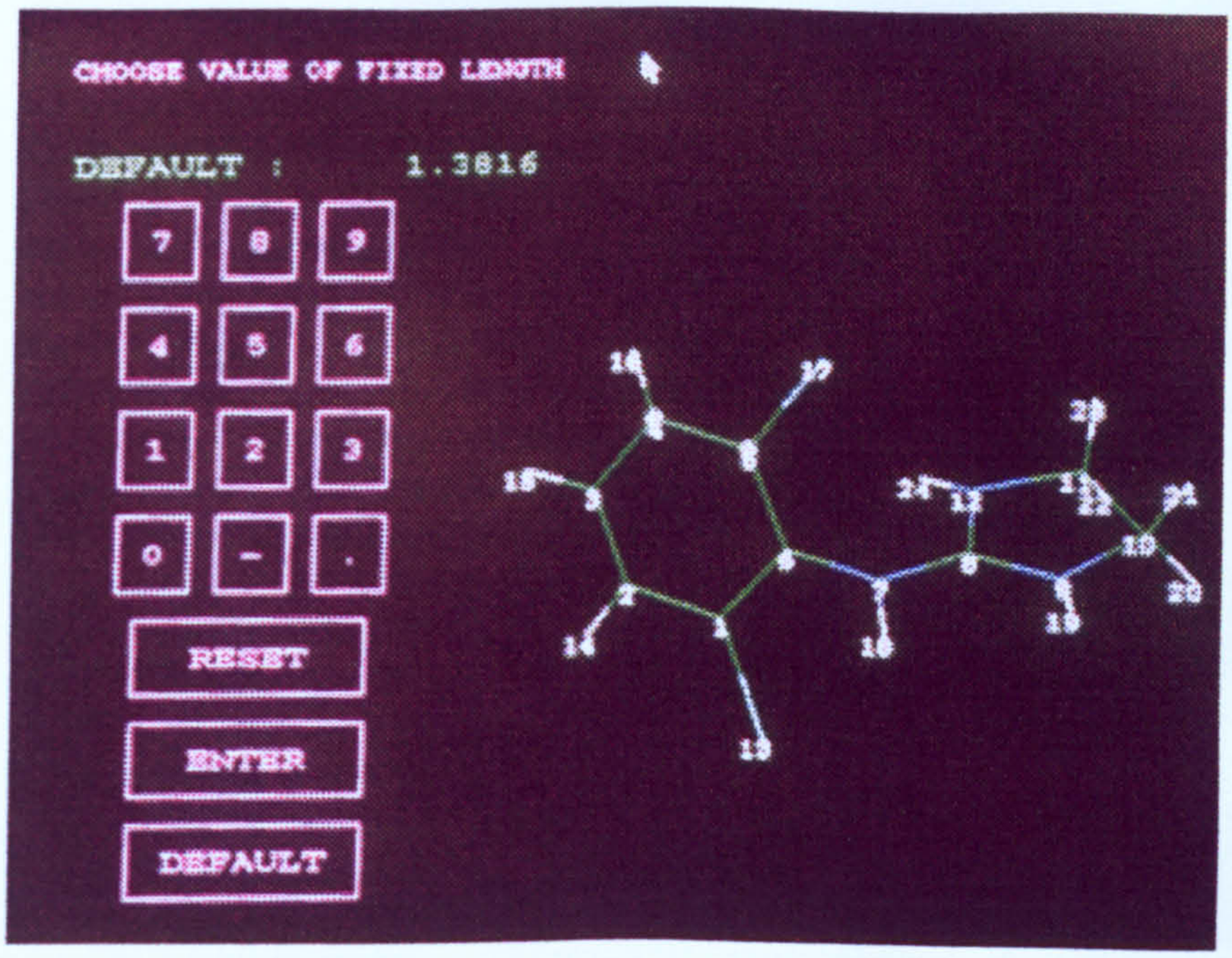


FIGURE 12. Selecting a fixed length

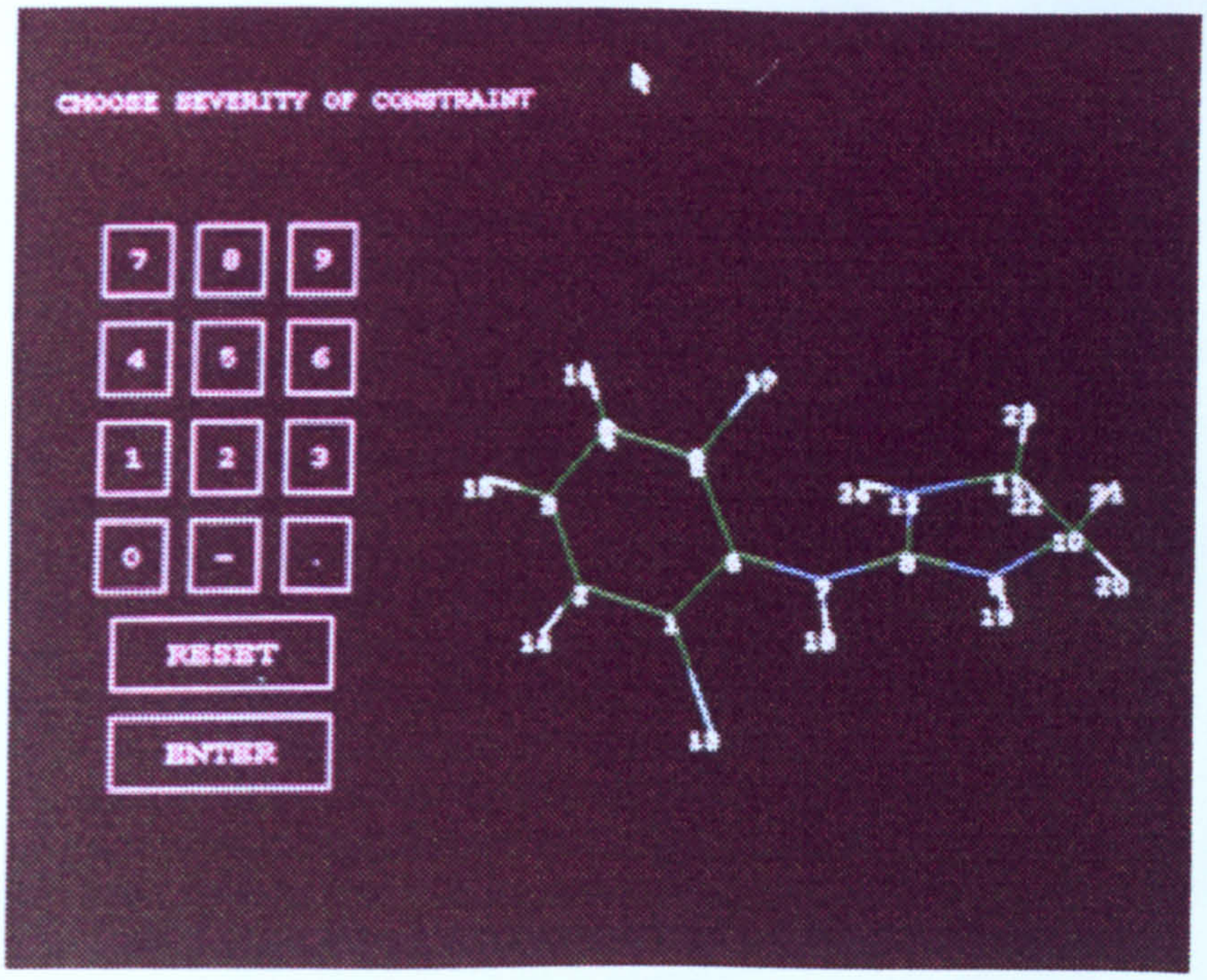


FIGURE 13. Entering severity of constraint

Appendix E:

Publications.

A circuit switched network for Inmos OSLinks

Lesley Bissland and David N.J. White
University of Glasgow, Department of Chemistry,Glasgow, Scotland
e-mail: lesley@tcrystal.gla.ac.uk

Abstract. Inter-processor communication paradigms are crucial factors in parallel computer system performance. An efficient circuit switching mechanism allowing dynamic-on-demand allocation of physical links between processing nodes is described. This cost-effective, memory-mapped system sends connection requests via an INMOS OSLink to a control processor which programs a crossbar switch. By setting up point-to-point direct physical links between nodes this allows every node to communicate directly with every other node of a parallel computer.

1.0 Introduction

The passing of messages between nodes is potentially the dominant component in the performance of multiprocessor architectures. If the communication mechanism cannot support the speed of the nodes then the full potential of a multiprocessor architecture is not realised. This has led to research into the development of better routing algorithms, hardware routers, and interconnection networks which all hope to provide high-bandwidth interprocessor communication resilient to failures, bottlenecks and deadlock.

There are three methods of interprocessor communication [1]: packet switching, circuit switching and packet switching through circuit switching. Any method which utilises packet switching usually relies on dedicated software on each node to manage the passing of the packets. INMOS provide software for the T-800 transputer which supports virtual channel routing (a packet switching scheme). Since additional software is required on each node to support the packet switching this uses up some of the computational power of the node and also introduces a large message latency. Obviously the longer the message the larger the message latency, therefore for systems where large volumes of data are transferred it is more efficient to use a circuit switching mechanism.

Circuit switching mechanisms establish a dedicated communication link between the two communicating nodes. This link is maintained until the complete message has been transmitted from the source node to the destination node. No dedicated communication software is required on each node, only on the control node which is setting up the communication links. The dedicated communication link can be set up before the execution of a program or dynamically on demand during the program run time.

A control processor programs a crossbar switch to set up the connections between nodes. To enable connections to be established during program run-time the nodes send connection requests to the control processor. In order to reduce the message latency an efficient and fast mechanism must be used to interface the nodes to the control processor.

Tudruj and Kalinowski [1] described a method whereby the nodes sent connection requests to a control transputer via a TRANSBUS [2]. The TRANSBUS is an application specific integrated circuit (ASIC) which acts as an interface between the node and the serial bus connected to the control processor. The ASIC is not commercially available and this restricts its use in systems other than [1]. Also one of the data links on the node provides the serial bus to the control transputer. In the case of the transputer this means that only three links are available for internode data transfers. This does not make good use of the total available bandwidth available from a transputer.

The dynamic-on-demand circuit switched network described in this paper employs the same principle as in [1] but instead of using an ASIC (i.e. TRANSBUS) several commonly available integrated circuits (ICs) are utilised to interface the nodes with the control processor. Connection requests from the nodes are sent via a memory mapped system which leaves all the links on the nodes free for interprocessor communication. The nodes are not restricted to transputers, the system can be used with any processor which provides an external memory interface. The control processor used is twice as fast as a T222 transputer.

2.0 Dynamic Interconnection Network

2.1 Basic Procedure

When a node (the source node) wants to communicate with an another node (the destination node) via the crossbar switch, it writes its connection request (a three byte packet) into a FIFO (First In First Out Memory), which stores the request until it is honoured. To select nodes for servicing a token passing protocol is used. The token circulates between the nodes and when a node receives the token and there is a request pending, the request is passed out of the FIFO to the control processor, via an INMOS OSLink. The control processor then decides whether the required connection is available and if so makes the connection. A message indicating success or failure to make the connection is sent to the source node. In this way the control processor processes connection requests from the nodes in a sequential manner.

2.2 Hardware subsystem

A diagram of the hardware is illustrated in Figure 1. A node requiring service writes its connection request into the First In First Out Memory (FIFO), which is mapped into the nodes memory address space. This allows the node to continue with other tasks while its connection request remains stored in the FIFO until honoured. When a node receives the token, its request is clocked out of the FIFO into a C011 Link Adapter [3]. The data is then transferred via the INMOS OSLink to the control processor which programs the crossbar switch. Access to the link is gated by a buffer ('125) which is only enabled when the node has the token and there is a request pending.

2.3 Token Passing

The token passing is achieved by a finite state machine (SM) implemented in PLDs (Programmable Logic Device). A state machine has a set of states and a set of transition rules for moving between the states at each clock edge (the clock is derived externally). The transition rules depend on the both the present state and on the particular combination of input levels present at the next clock edge.

The token passing bus consists of two lines: one which passes the token and one which acknowledges the passing of the token (See Figure 2). The state variables are TokenOut and AckOut, and the inputs are TokenIn, AckIn, and HoldToken. The token is effectively a bit

which passes between the PLDs and each node has a PLD associated with it. The clock used for the token passing is 8MHz

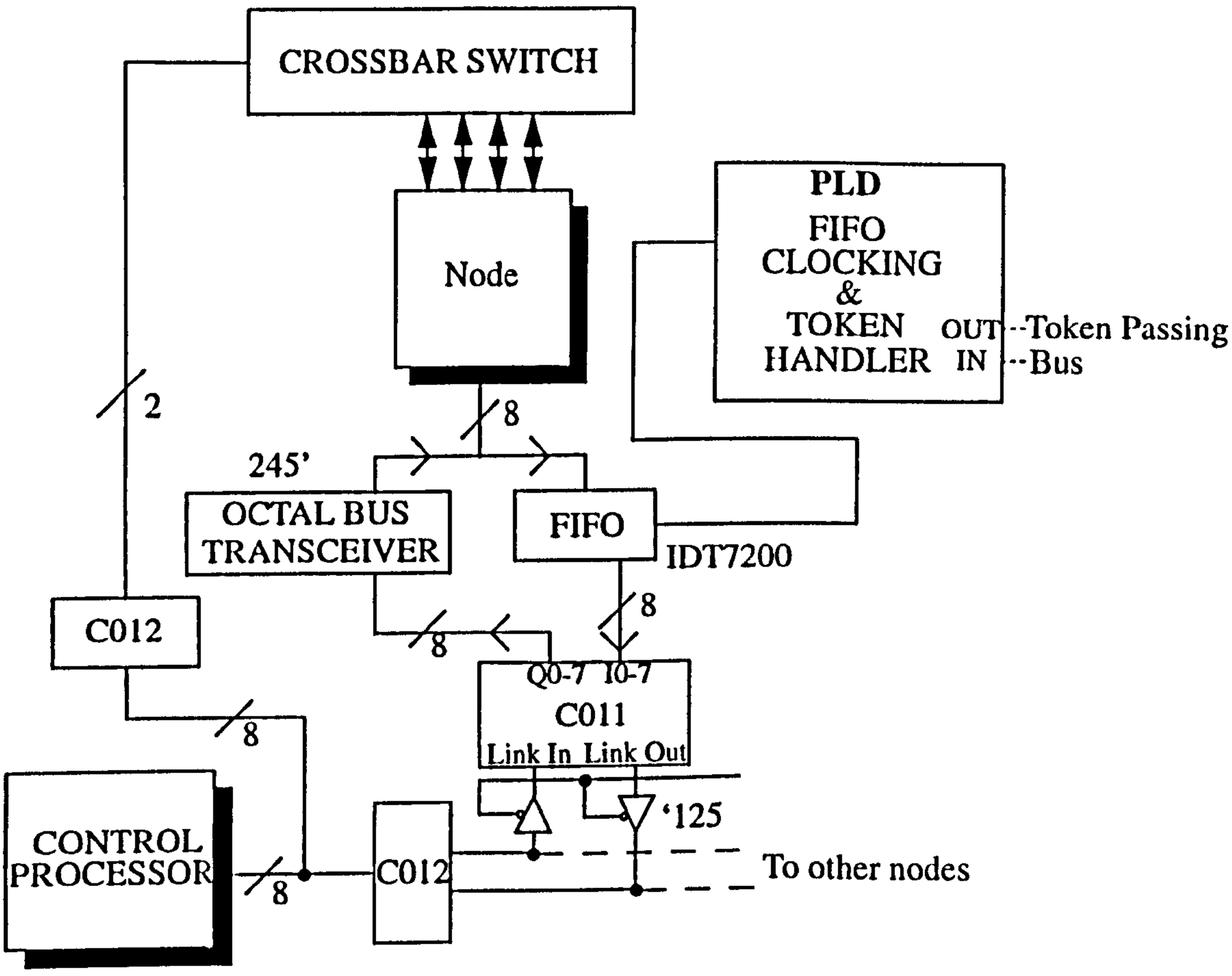


FIGURE 1. Dynamic Interconnection Network (1 node)

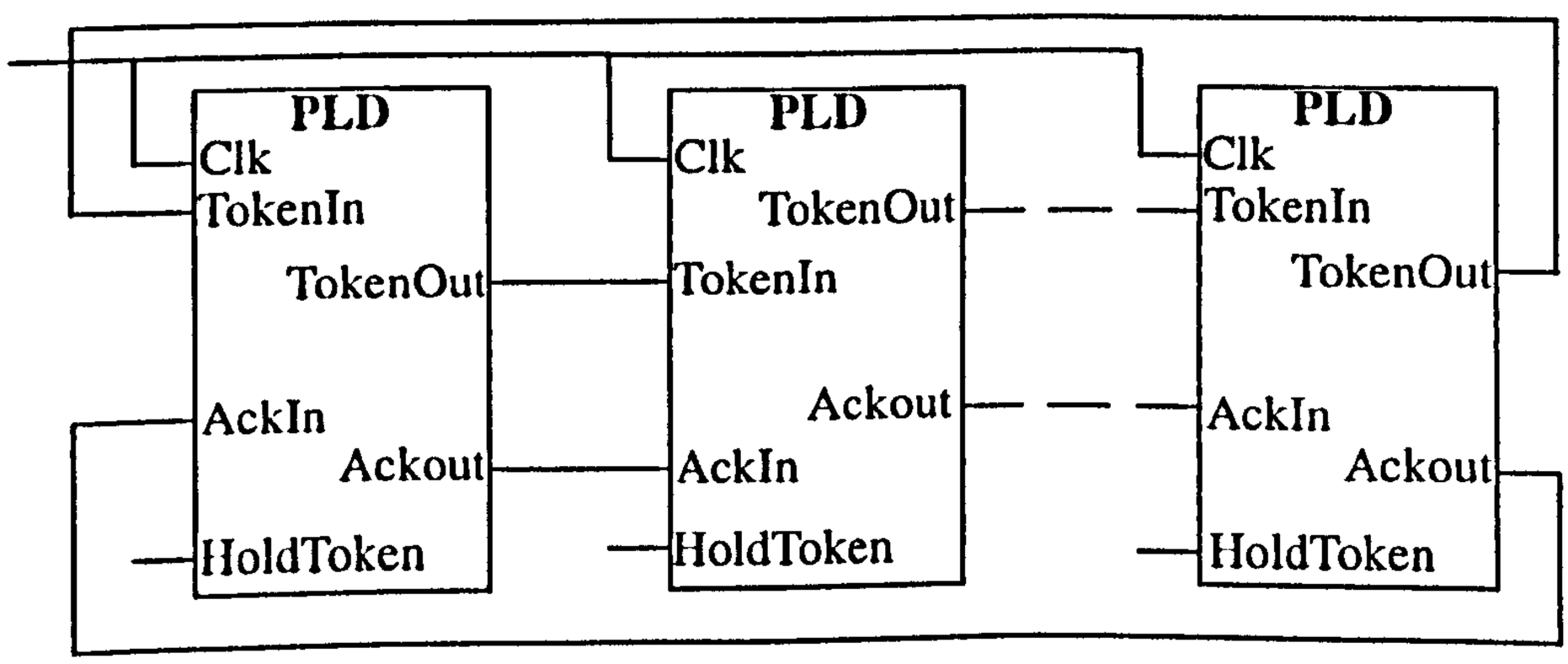


FIGURE 2. Token Passing

If a node receives the token and there is a request packet in the FIFO, then the token must be retained until the FIFO has been emptied and the node no longer requires the token. This is achieved by the HoldToken signal, which is generated by using a combination of the

Empty_Flag* (EF*) signal from the FIFO (logic false (+5V) when the FIFO contains bytes), and a D-type flip-flop (See Figure 3).

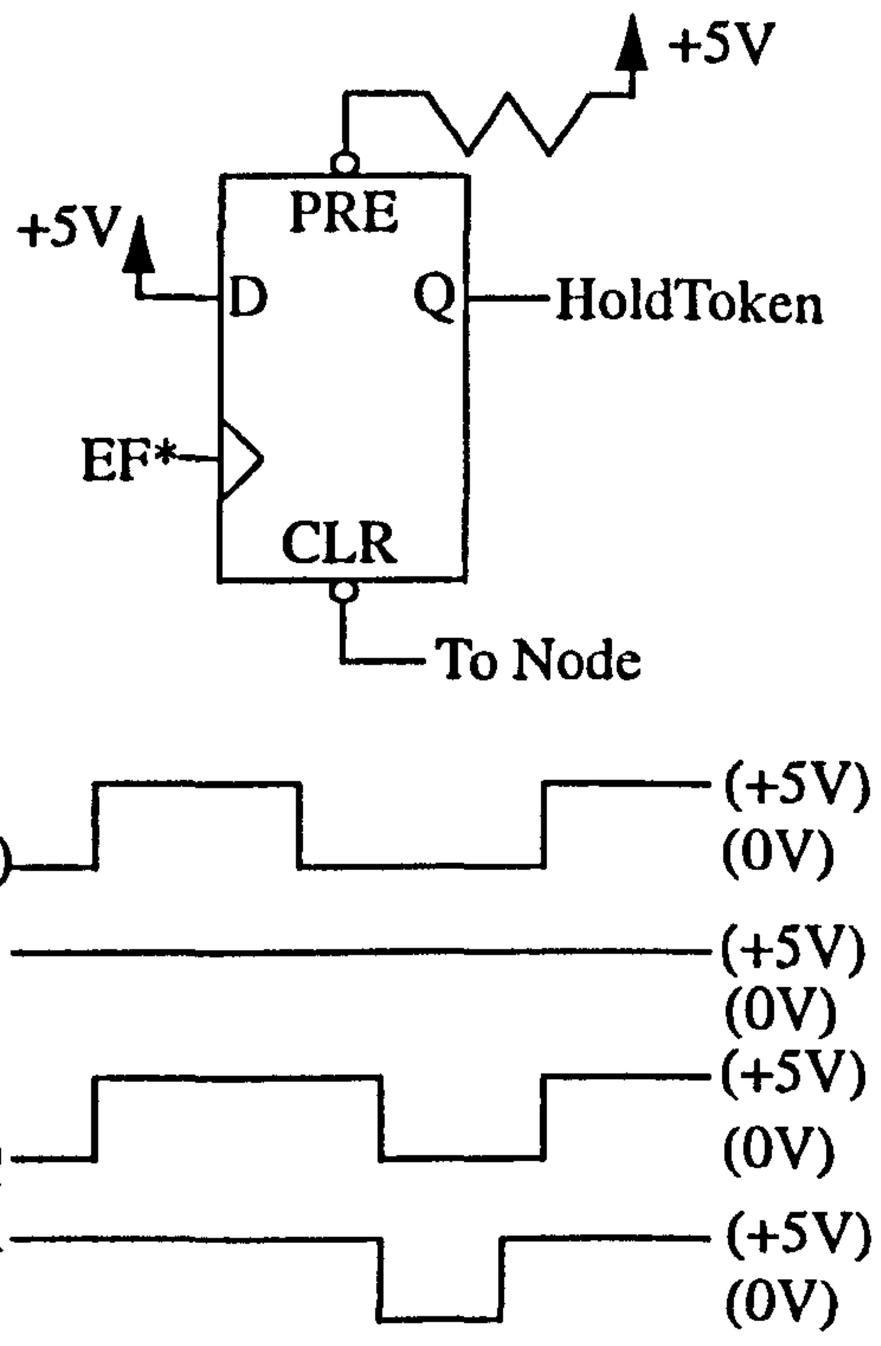


FIGURE 3. Generation of HoldToken signal

The Q-output of the flip-flop is used as the HoldToken signal. The EF* signal clocks the flip-flop: therefore when EF* becomes logic false (+5V) indicating data is in the FIFO, the level at the D-input (logic high (+5V)) is transferred to the Q-output of the flip-flop. To release the token, the node pulls the CLR* signal on the flip-flop logic low (0V) for a short period, which clears the Q-output back to logic low releasing the token.

A state diagram for the token passing state machine is shown in Figure 4. The SM remains in state zero (S0) until the token arrives (i.e TokenIn = 1) and then on the next clock edge proceeds to state one (S1) which acknowledges the arrival of the token by setting AckOut true. If HoldToken is true then the state machine remains in state one (S1), otherwise on the next clock edge it proceeds to state two (S2) which passes the token on by setting TokenOut logic true. The SM does not go back to state one until the passing of the token has been acknowledged (AckIn = 1). To inject the token into the system one state machine is programmed with the initial state holding TokenOut true (i.e the initial state is S2).

2.4 FIFO Access

The connection request in the FIFO must be clocked out byte at a time to the C011 and then sent to the control processor. This is also achieved by a finite state machine (See Figure 5). The state machine controls the RD* (read) signal on the FIFO and the IAck and IValid signals on the C011. Pulling the RD* signal low transfers a byte out of the FIFO to the C011 parallel port. In order to transmit the byte from the parallel port to the INMOS OSLink, IValid is pulled high. To indicate the byte has been transferred successfully IAck is pulled high by the C011 and then IValid returned low by the SM.

The Empty_Flag* on the FIFO signals to the state machine when data is present in the FIFO, and an output called TokenArrived from the token passing state machine indicates when the token is present. The state machine waits at S0 while the FIFO is empty or the token has

not arrived. When the token arrives and there is data in the FIFO the SM then proceeds to S1 on the next clock edge and this initiates a read cycle on the FIFO. On the next clock edge the SM then unconditionally jumps to S2 which takes IValid true and enables the buffer ('125) that restricts access to the serial bus. The SM then waits for IAck to become true, indicating the transfer of a byte to the INMOS OSLink, before proceeding back to S0.

The FIFO clocking and Token passing state machines can be implemented in the same PLD and therefore can both use the same clock. Using terminated coax wires to connect the clock inputs on the PLDs together, it would be possible to run the token passing and FIFO clocking at 40 MHz.

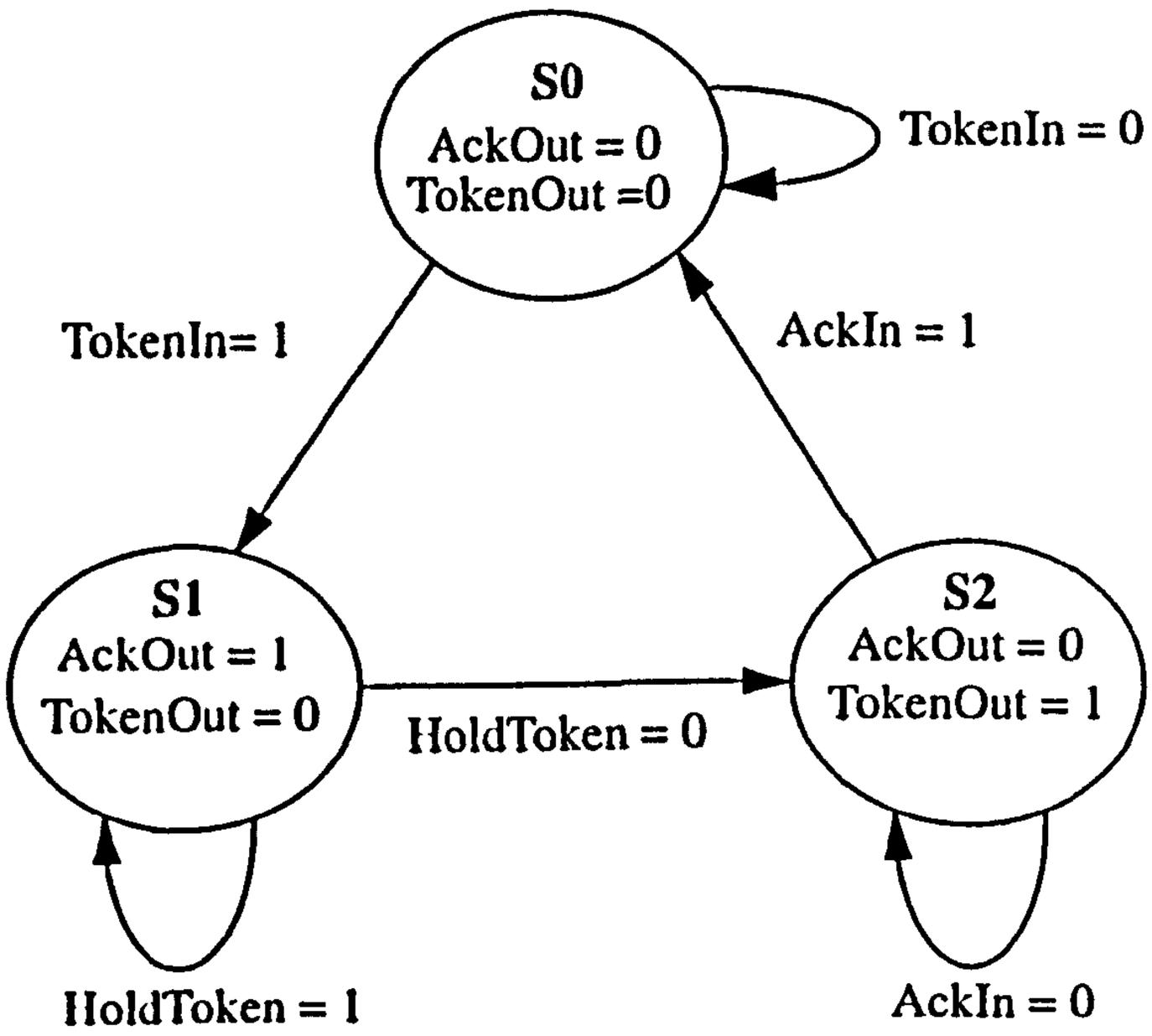


FIGURE 4. State Diagram for token passing

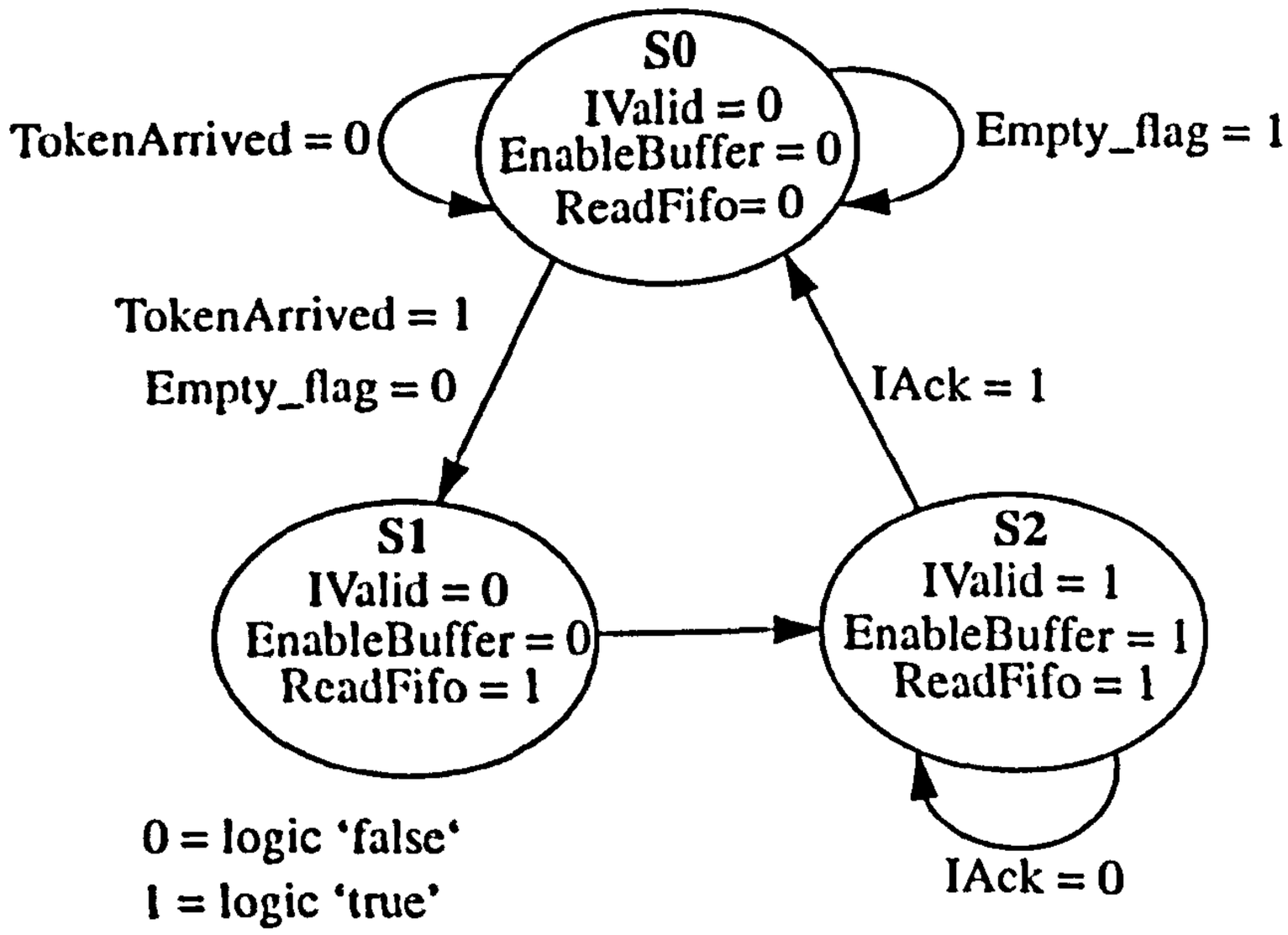


FIGURE 5. State Diagram for FIFO clocking

To summarise the steps required to communicate with the control processor a schematic representation is shown in Figure 6.

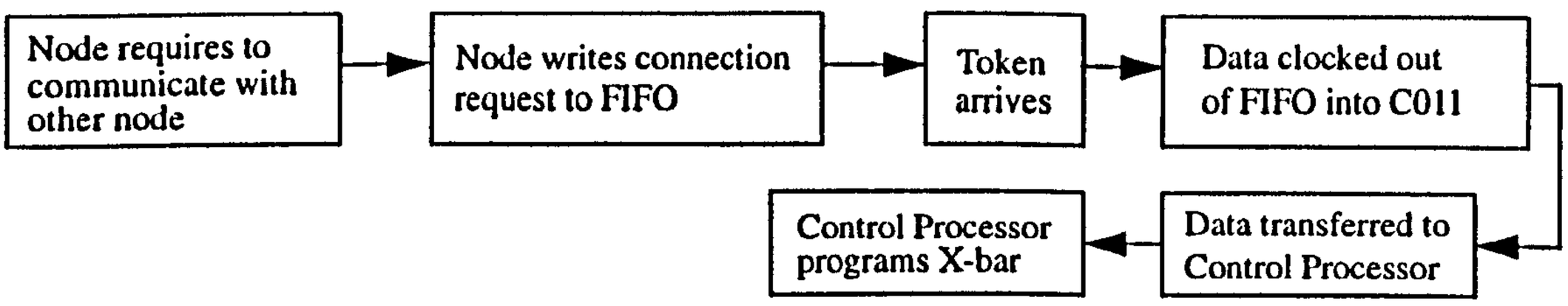


FIGURE 6. Schematic Representation

3.0 Control Processor

The function of the control processor is to receive the connection requests from the nodes and then program the required connections on the crossbar switch. The processor used is an Analog devices ADSP-2105 [4]. This is used instead of a transputer as it achieves higher performance (at least twice as fast as a T222 transputer) at much lower cost.

3.1 ADSP -2105

The ADSP-2105 is a 12MHz microcomputer suitable for high-speed numeric processing applications (higher speed versions are available). It contains 1K words of on-chip program memory RAM and 512 words of on-chip data memory RAM. The internal program memory can be automatically booted upon reset from an EPROM.

3.2 Software considerations

When the source node decides it wants to communicate with the destination node, the system level software on the source node scans the links on the node for a free link to communicate with the destination node. Once a free link is found the source node sends its connection request (consisting of three bytes) to the control processor (See Figure 7).The first byte contains the address of the source node and the second byte contains the link number on the source node. The third byte holds the address of the destination node. This protocol can be expanded for more processors by using two bytes for the addresses of the source and destination processors.

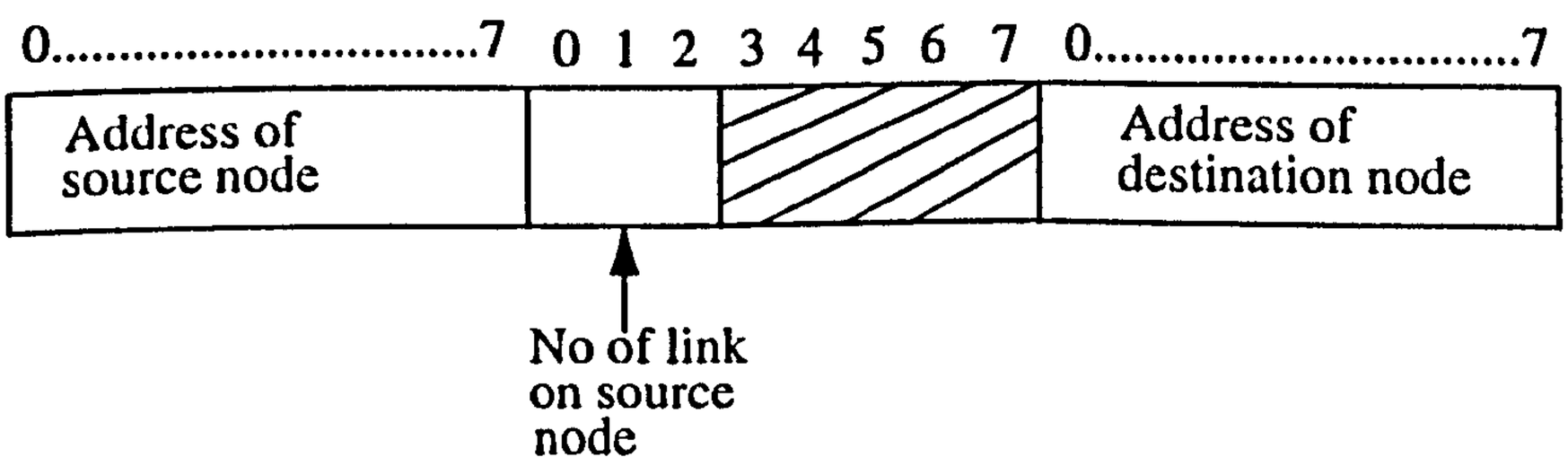


FIGURE 7. Connection request sent by node

The control processor has a table in memory which contains the connections from the nodes to the crossbar switch and a flag to indicate whether the connection is already in use (See Figure 8). When a connection request is received the control processor scans the table to find the link on the crossbar switch that the source node is connected to. It then scans the table

looking for a free link on the destination node and if one is free makes the connection on the crossbar switch which connects the source node to the destination node. The flags in the connection table are then updated and an acknowledge is returned to the source node.

ARRAYS

Node No. [32]	Link No On Node [32]	Link No On Crossbar [32]	Connection [32] Used/Unused
0	0	10	1
0	1	25	0
0	2	12	0
0	3	30	1
1	0	8	1
.....	.....	.....	.....

FIGURE 8. Connection Table in Control Processor

The format of the acknowledge byte is shown in Figure 9. The link number of the destination node is sent in order to allow the source node to make disconnection requests (more on this later). If the value of the byte returned is greater than the number of links on the destination node (i.e greater then four for a transputer), this signifies to the source node that the connection could not be established. The byte is returned via an octal bus transceiver rather than the FIFO as only one byte is returned to the requesting node. Data from the source node to the control processor is therefore transferred via the FIFO and data is returned to the source node via an octal bus transceiver.

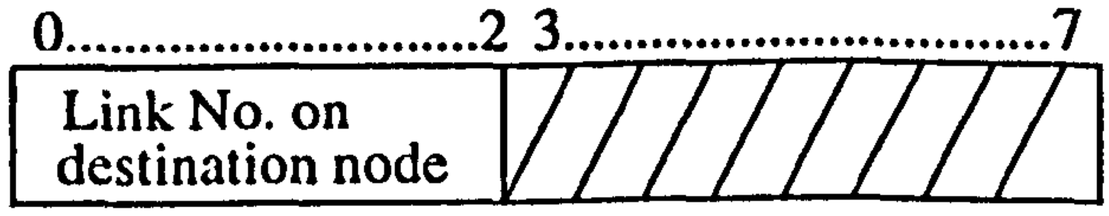


FIGURE 9. Acknowledge Byte returned to source node

Once a node receives a message indicating its connection request has been honoured it is free to send data via the crossbar switch to the destination node. The source node knows when the message has been successfully received by the destination node due to the link acknowledge protocol used by INMOS OSLinks. When the data has been completely transferred then the connection can be broken. The format of the disconnection request made by the source node is shown in Figure 10.

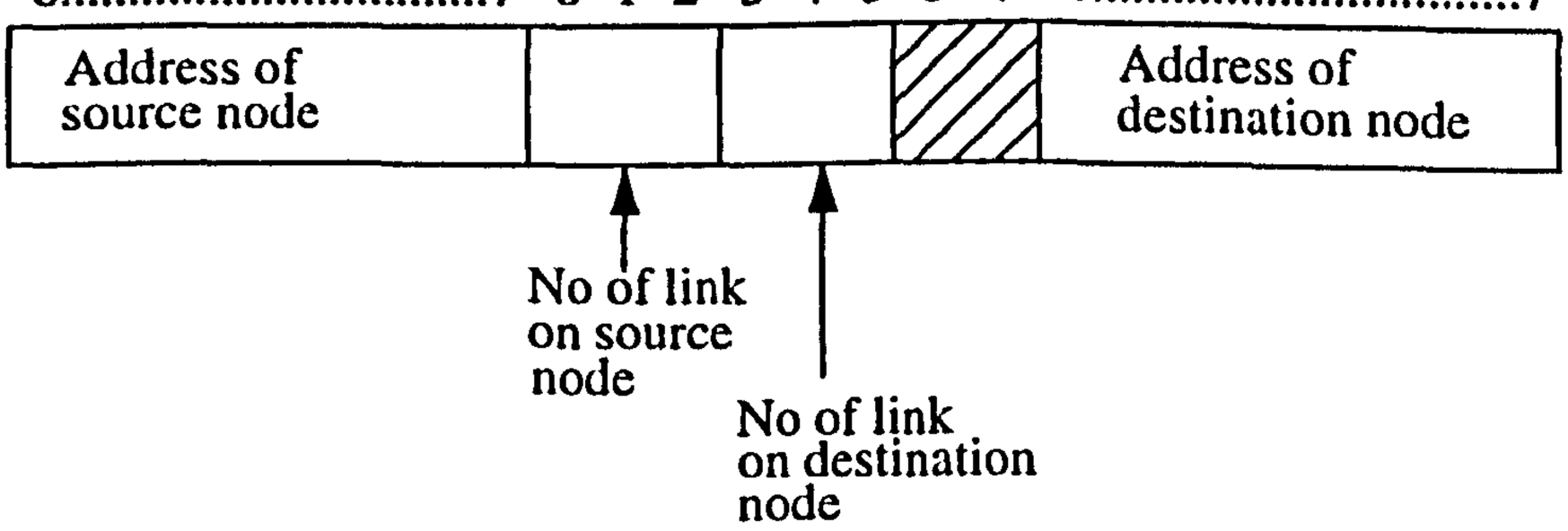


FIGURE 10. Disconnection Request

The message is basically the same as a connection request, except that the number of the link on the destination node is sent as well. The reason for this is that in the case where two nodes are connected by two or more links then the link numbers need to be specified in order to disconnect the correct link.

The control processor can distinguish between connection and disconnection requests by looking at the value of the second byte. If it is greater than the number of links on the source node then the request must be a disconnection request (i.e it contains the address of the destination node).

3.3 Crossbar switch

The crossbar switch used is the 32-way INMOS C004 [3]. Since in the case of a transputer all four links can be connected to the crossbar switch, 8 transputers can be fully connected by using a C004. A 64-way crossbar switch such as the LSILogic L64270 [5] could be used which would allow 16 transputers to be fully connected. If more transputers were required then the number of crossbar switches could be increased. Each crossbar switch would have a C012 connected to it and the C012 would be addressed by the control processor. Effectively each crossbar switch would have a unique address.

4.0 Connection Request Service Time

The four major factors involved in the time taken to service a request are:

- the time required to pass the token.
- the time taken to clock the bytes out of the FIFO
- the time to transfer the bytes from the C011 to the Control Processor
- the time required to program the crossbar switch.

If the PALS are being clocked at 20MHz and the INMOS OSLink is operating at 20Mbits/s then the connection request service time is approximately 2μs. The service time could be speeded up by using a faster token passing clock and control processor.

5.0 Conclusions

All the ICs employed in our system are commercially available at relatively low cost. The control processor used achieves a much higher performance than a transputer enabling it to process connection requests much faster. Also all the valuable communication links on the node are free for interprocessor communication and are not tied up with control information.

This cost-effective method provides deadlock free, low message latency, dynamic reconfigurability. This is especially useful in time critical applications which transmit and receive large volumes of data such as robotics and image processing. Although in our case the nodes were transputers, the hardware subsystem can be used with other processors providing they possess a high speed communication mechanism.

The system described is a prototype version which functions successfully. It is hoped that when the system is fully integrated into a multiprocessor architecture a faster token passing clock and control processor will be used.

Acknowledgements

LB would like to thank the EPSRC (Engineering and Physical Sciences Research Council) for the award of a studentship to fund this work.

References

[1] M.Tudruj, T.Kalinowski, Multi-Transputer Systems with Dynamic Link Connection Switching Controlled through a Serial Bus. *Transputer Applications and Systems* 1993, pp 803 - 818

[2] J.P. Calvez, O. Pasquier, A Transputer Interconnection Bus For Hard Real-Time Systems, *Transputers* 1992, pp 273 - 283

[3] INMOS Ltd, Transputer Databook, INMOS 1992

[4] ADSP-2100 Family Assembler Manual 1991, Analog Devices, Inc.

[5] L64270 Preliminary Data, LSI Logic Corporation 1989

Parallel Molecular Mechanics Calculations

Lesley Bissland and David N.J. White
Department of Chemistry, University of Glasgow, Glasgow, G12 8QQ
e-mail:lesley@tcrystal.gla.ac.uk

Astract. This paper describes the parallelisation of a sequential FORTRAN molecular mechanics program to run on novel hardware, where each node processor has a dedicated high speed link to the host processor, and to all of the other nodes. The host processor can broadcast code/data to the nodes over these direct links using an overhead free hardware mechanism. The broadcast hardware is supported by the COMFORT message passing subroutine library.

1. Introduction

In order to design a new molecule using a CAMD (Computer Aided Molecular Design) package [1,2,3] a crude structure is built up by combining smaller molecules or fragments of molecules into a larger overall structure. The molecule can also be constructed one atom at a time using known average bond lengths, valency angles, and torsion angles. Obviously structures built up this way are extremely crude as they do not take into account the interactions between the various molecular fragments and how they will affect the structure of the molecule as a whole. Before the modelled structure can be useful in the drug design process its structure must be computationally optimised by a procedure known as energy minimisation.

The steric energy of the molecule is calculated by adding together the potential energy contributions from bond stretching, angle bending, bond torsion, non-bonded interactions, coulombic interactions and pyramidalisation of non-planar systems. Once this energy has been found the geometry of the molecule is systematically altered (i.e. the atomic coordinates are shifted by a calculated amount) in the hope of locating the global energy minimum.

Various optimisation methods can be used in the attempt to find this global energy minimum. Unfortunately though, all of these methods are prone to locate the local energy minimum closest to the starting point of the calculation, rather than the global minimum. The difference between local and global minima is illustrated below.

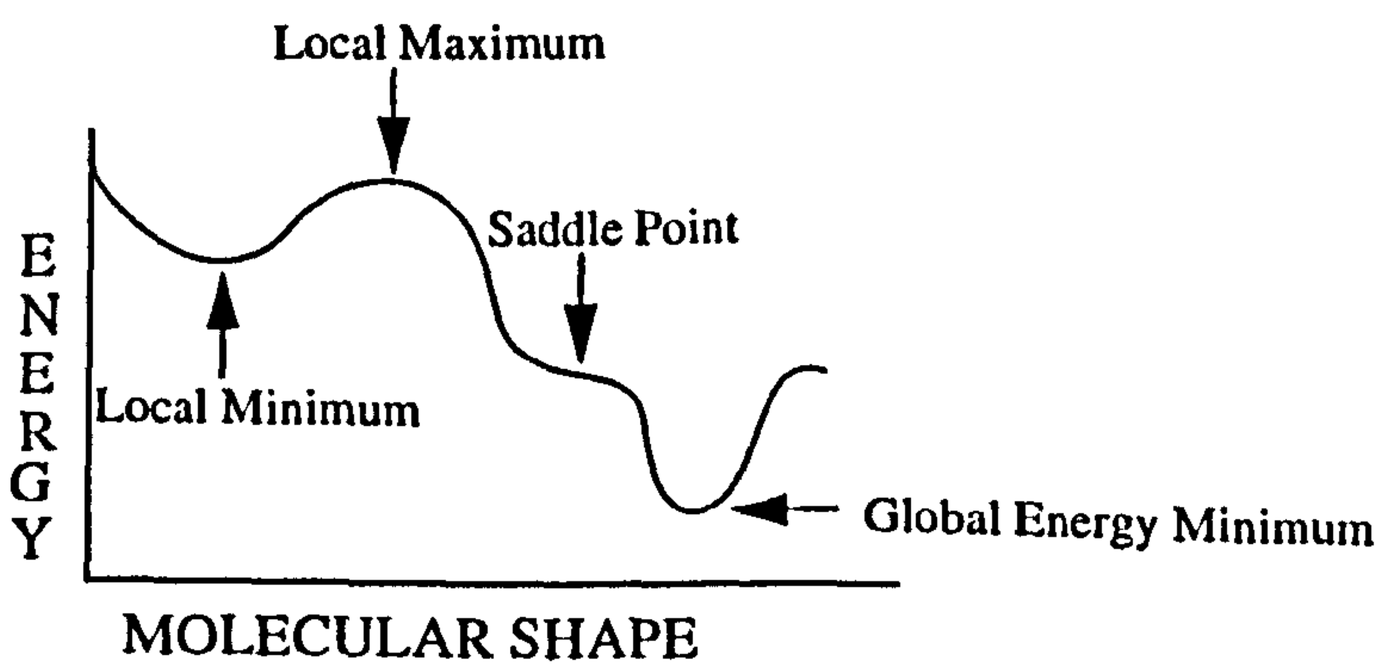


FIGURE 1. Energy of a molecule vs. Molecular Shape

There are two main categories of optimisation technique namely search and gradient methods. An example of a search method is pattern searching [4].

Pattern searching applies positive and negative shifts (~0.1Å) to each atom's atomic coordinates one at a time and then tests to see whether the steric energy has decreased or increased. If the energy has decreased then the atom is left in its new position and the new steric energy used as the current value. However, if the energy has increased then the atom is returned to its original position and the coordinate is then shifted in the opposite direction. Again the steric energy is calculated and if it has decreased then the atom is left in the new position otherwise it is returned to its original position.

The whole pattern of successful shifts built up in this way is repeated and the steric energy checked for further reduction. The pattern is repeated until it no longer works and then the pattern is repeated with half the shift value and then iteratively until the shift reaches a sufficiently small value (10^{-5}Å). When the current pattern no longer works, or the shift becomes too small, a new pattern is established and the whole process repeated until a reduction in energy is no longer possible.

This method is guaranteed to find a local energy minimum and has a large radius of convergence (i.e. even with an extremely crude starting structure a local energy minimum will be reached). However the rate of convergence is slow as the same shift size is applied to each coordinate and the shift size is refined very slowly (i.e it could take hundreds of iterations to reach an energy minimum).

Gradient based methods again apply a shift, in the search for lower energy, to each coordinate but in this case the shift is proportional to the gradient of the steric energy at this point (i.e if the gradient of the steric energy is steep then large shift is applied, if the steric energy function is flatter then a smaller shift is applied). These techniques are said to reach an energy minimum when the vector of first partial derivatives of the steric energy with respect to the atomic coordinates is zero. This is the case not only at energy minima but also at energy maxima and saddle points; a feature of gradient methods which can be useful when searching for transition state structures but an inconvenience when looking for minima.

Gradient based techniques have a fast rate of convergence as they calculate shifts based on the gradient of the steric energy function. However, the radius of convergence is small for the popular full matrix Newton Raphson (NR) iteration (see later for explanation). The radius of convergence can be increased by using approximations to the full NR such as the Block Diagonal Newton Raphson iteration and steepest descents, but at the expense of rate of convergence.

An example of a simple gradient based method is steepest descent [5] (a variation of the full NR iteration - almost all gradient based methods of optimisation are variants of the NR iteration). This involves calculating the gradient (the first partial derivative of the steric energy w.r.t. the atomic coordinates) of the steric energy function at a particular point. Once the gradient has been calculated the coordinates are shifted in the direction of lower energy by an amount proportional to the gradient. The constant of proportionality is determined empirically. This procedure is repeated until a local minima is reached.

Steepest descents has the disadvantage that it is only the gradient of the steric energy function that is considered and the curvature (the second partial derivatives of the steric energy) of the function is not taken into account when calculating the shift.

A technique which considers both the gradient and curvature of the steric energy function is the Block Diagonal Newton Raphson iteration [6]. This technique converges faster (usually in 50-200 iterations) than the steepest descent or pattern based methods and has a reasonable radius of convergence. This is the method used in our energy minimisation algorithm and it will be discussed in detail in Section 2 .

Obviously the larger the number of atoms in the molecule, the longer the optimisation takes, so for large protein structures comprising up to thousands of atoms the program run-time can be very long on a sequential computer. Energy minimisation can be parallelised by dividing up the atoms between the available nodes so that each node works on a 'slice' of atoms; i.e each node executes the same code but on a different data set.

Many implementations of parallel molecular dynamics [7,8] (the simulation of molecular motions with time) have been attempted, however little work has been published on parallel energy minimisation. Schweitzer et al. [9] parallelised the molecular mechanics MM2 package by splitting four computationally intensive subroutines over four processors on a shared memory computer. Our parallel minimiser parallelises the code by dividing the data domain onto the available processors on a distributed memory machine.

In our parallel minimiser each node has a copy of the atomic coordinates of all the atoms, as some of the atoms in it's 'slice' may interact with atoms on other nodes. The nodes consider each atom in their 'slice' one at a time. For each atom the first and second partial derivatives of the steric energy with respect to the atomic coordinates are calculated. The atom's corrected coordinates are then computed using the Newton Raphson iteration. Once the corrected coordinates for all the atoms have been computed, they are sent back to the host. The host assembles a complete set of new 'improved' coordinates from the 'slices' returned by the nodes and broadcasts this set back to all the nodes ready for the next iteration.

In a conventional 3L FORTRAN based implementation of parallel molecular mechanics the nodes would be connected together in a pipeline or more complex topology and the code would be loaded onto the nodes in the standard 'store and forward' manner [10]. Any data exchanged between the host and nodes will generally have to pass through one or more intermediate nodes before it reaches its destination.This requires the nodes to run communication tasks which reduces the raw computational power deliverable to the application.

The overheads discussed above can be eliminated by broadcasting code and data to all the nodes simultaneously via hardware. This paper describes a parallel implementation of an energy minimiser which utilises the COMFORT host/node programming environment and BB08 octal broadcast link interface [11]. Each node has a direct link to the host computer, down which code and/or data can be transmitted, received or broadcast.

COMFORT is a library of FORTRAN subroutines similar to those provided by EXPRESS [12] and MPI [13], which allow the host computer to broadcast load code onto the nodes and also facilitate communication between the nodes, amongst other things. The host/node methodology allows the host to participate in the calculation rather than act merely as a facilities server. COMFORT makes parallelisation easier because no communication tasks are required on the nodes and no configuration (in the 3L sense) is required.

The BB08 is basically an eight fold replication of the Inmos B004 interface which also allows link broadcasting (See Figure 2). The board contains eight C012 link adaptors and the links are either routable to size one TRAM slots or to a DIN41612 connector. Data can be broadcast from the PC bus simultaneously to all the link

adapters. The DIN41612 connector is utilised when other PCs are used as nodes, otherwise the transputer or other TRAM based processors are plugged into the BB08.

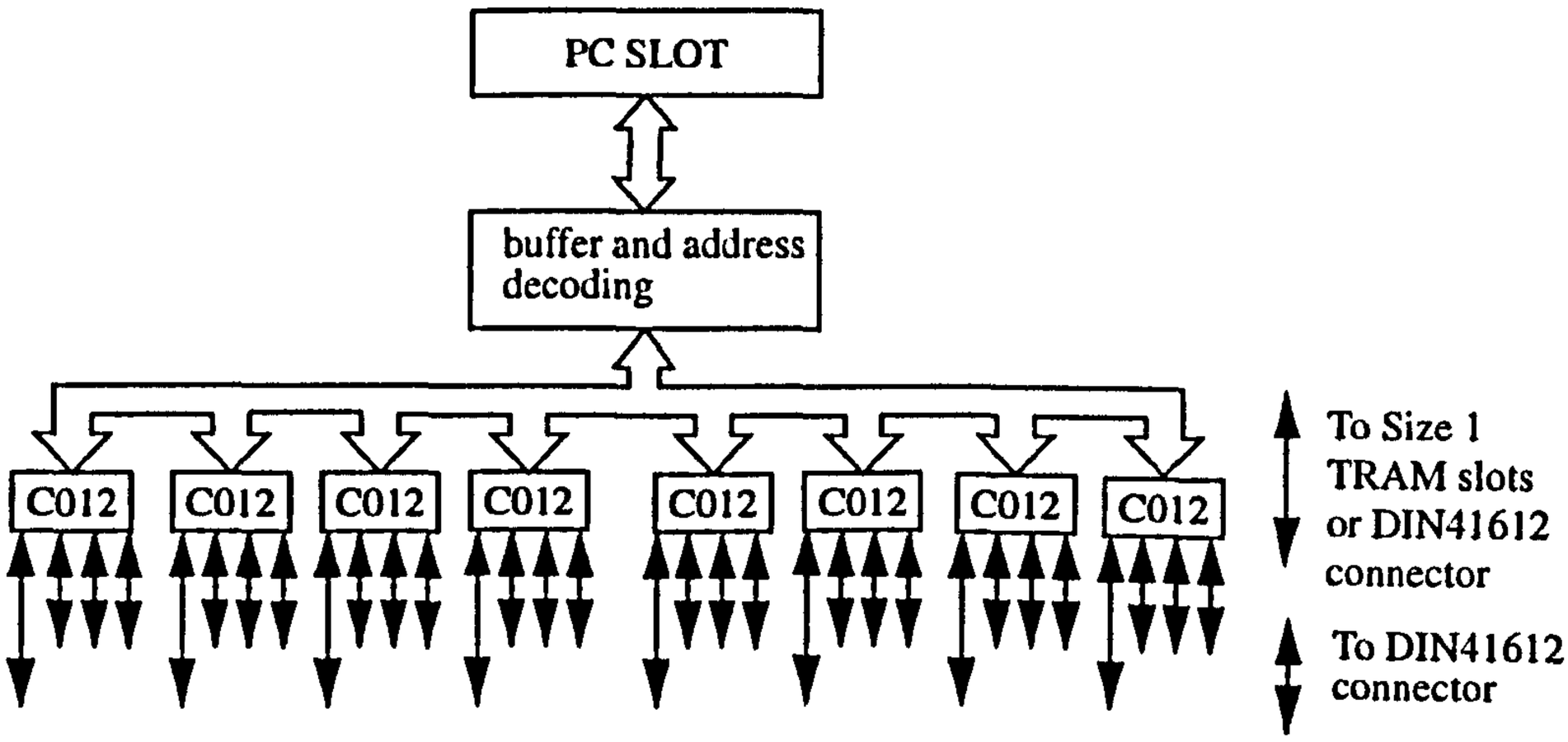


FIGURE 2. Basic diagram of BB08 board

The following sections begin with an overview of the energy minimisation procedure, and then focus on the COMFORT host/node FORTRAN code.

2. Energy Minimisation Procedure

2.1 Steric Energy Equation

The equation to calculate the total steric energy of a molecule (V_s) is:-

$$V_s = V_l + V_\theta + V_\omega + V_r + V_q + V_\chi$$

(EQ 1)

where V_l represents the summation over all the bonds in the molecule of the individual potential energies due to bond stretching or compression, and V_θ, V_ω ,V_q , V_r and V_χ represent similar terms for angle bending, bond torsion, coulombic interactions, non-bonded interactions, and out-of plane bending respectively.

The equations for the individual components are shown below:-

$$2V_l = \sum_l k_l (l - l_0)^2$$

(EQ 2)

k_l = the stretching force constant

l_o = reference bond length

$$2V_\theta = \sum_\theta k_\theta \left(\Delta\theta^2 - k'_\theta \left(|\Delta\theta^3| - 0.0004 |\Delta\theta^5| \right) \right)$$

(EQ 3)

$\Delta\theta = \theta - \theta_0$

k_θ = angle bending force constant

k'_θ = anharmonic force constant

θ_0 = reference bond angle

$$2V_w = \sum_\omega [V_n (1 + s\cos n\omega) + V_l (1 + s\cos\omega)]$$

(EQ 4)

V_n = n - fold components of the barrier to free rotation.

V_l = one - fold component of the barrier to free rotation.

ω = torsion angle.

n = periodicity

s = + 1 for staggered torsional energy minimum

s = -1 for an eclipsed minimum

$$V_r = \sum_r [Ar^{-12} - Br^{-6}]$$

(EQ 5)

A and B are constants which depend on the atom types.

$$V_q = 332 \sum_r q_i q_j / Dr$$

(EQ 6)

q_i = charge on atom i

q_j = charge on atom j

r = distance between atoms i and j

D = dielectric constant

$$2V_\chi = \sum_\chi k_\chi (180 - \chi)^2$$

(EQ 7)

k_χ = force constant for out-of-plane bending

χ = improper torsion angle in degrees

2.2 *Newton Raphson Iteration*

The basic NR iteration which minimises the steric energy of the molecule is given by:-

$$x_{k+1} = x_k - \alpha F^{+} \nabla V_s(x) \qquad \text{(EQ 8)}$$

where **x** is the 3N (N = number of atoms) long vector of cartesian coordinates, α is the step length, F^{+} is the generalised inverse of the Hessian:-

$$F = \frac{\partial^2 V_s}{\partial x_i \partial x_j} \quad ; \qquad i = 1,3N,\ j = 1,3N$$

and:-

$$\nabla V_s(x) = \frac{\partial V_s}{\partial x_j}; \qquad j = 1,3N$$

The calculation of the complete Hessian (a 3N x 3N matrix) is a very time consuming procedure and is not really suitable for molecules with over 200 atoms. Therefore an approximation known as the Block Diagonal Newton Raphson (BDNR) is used. This is so called because only the second partial derivatives in each 3x3 block along the leading diagonal of the Hessian are calculated. Therefore F is given by:-

$$F = \left(\frac{\partial^2 V_s}{\partial x_i \partial x_j} \right); \qquad i = 3m+1, 3m+3;\ j = 3m+1, 3m+3;\ m = 0, N-1$$

Each block contains second partial derivatives of the steric energy with respect to the coordinates of only one atom. The BDNR iteration can therefore be applied one atom at a time, this allows each atom to be moved to its corrected position before the calculations for the next atom are started. Each atom's position is therefore calculated on the basis of the best structure available at the time.

3. Parallel Energy Minimisation Algorithm

The procedure described uses an early version of COMFORT and hardware reserved for program development work. The host code was written in Microsoft (16-bit) FORTRAN (current versions of COMFORT use Microsoft 32-bit FORTRAN) and the node code with 3L parallel FORTRAN (the 3L FORTRAN node programs are configured with the stand alone FORTRAN run time library). Although this methodology is not without it's problems (some of which will be explained later) it does result in reasonably portable programs. The hardware set-up used was a BB08 board with four size 1 TRAMs each with a T4XX transputer and 1Mbyte of memory. The algorithm was derived from pre-existing sequential FORTRAN code.

The host program first calculates the initial steric energy of the molecule. It then sends data, which includes the atomic coordinates, to all the nodes. While the nodes are computing the new atomic coordinates the host is idle waiting for the new 'improved'

coordinates to return. Each node returns improved coordinates for the 'slice' of atoms it is responsible for. These 'slices' are assembled into a complete set of improved coordinates and broadcast back to all of the nodes. When the required number of iterations have been completed by the nodes the host recalculates the new minimised steric energy.

The node processors calculate the first and second partial derivatives described above and use these to obtain improved coordinates for their 'slice' of atoms via the BDNR iteration. The node processors do not need to communicate with each other.

3.1 Host Program

The pseudocode showing the main features of the host program is illustrated in Figure 3.

The program sets up tables of reference bond lengths, bond angles, torsion angles, non-bonded interactions and coulombic interactions. From these values, and various other constants, it is possible using various mathematical approximations to calculate the various force constants required for the calculations. Once all this information is available, the host calculates the steric energy components and adds the values together to get the initial steric energy.

```
Read file containing atomic coordinates
Read file containing various parameters
Set up various tables required for the calculation
(i.e. bond lengths, bond angles etc.)
Calculate the total potential energy of the molecule
Configure,reset,load and initialize nodes
Send arrays of data to the nodes
While (no.of iterations not complete) do
    Send atomic coordinates to nodes
    Receive modified coordinates from nodes
Calculate Final Steric Energy of the Molecule
```

FIGURE 3. Pseudocode for Host Program

The nodes require a substantial amount of information to calculate the first and second partial derivatives of the steric energy w.r.t the atomic coordinates. Some of this data is sent from the host and some is recalculated on the nodes (duplicating a host calculation) as it is quicker.

The figure overleaf (Figure 4) shows the broadcast load/broadcast data FORTRAN code.The nodes are loaded with code via the configure, reset, load and initialize routines. The configure routine defines the hardware setup and its arguments are the base address of the BB08 board, the number of processors and a value which specifies the 'tick' of the timeout clock. The reset routine resets all the nodes and the load routine loads the nodes with the file 'nodemin.app'. Each node is assigned its id number by the initialize routine and this routine tells each node the link interconnection

topology via the matrix 'ProcConn' (the latest version of COMFORT uses complete connectivity and the topology maps are unnecessary).

```
NETCAST = -1
file= 'c:\comfort\lesley\min\nodemin.app'//char(0)
call configure(#180, 4, #976f)
call reset(NETCAST)
call load(NETCAST, file, 100, error)
do i=1,4
   ProcConn(1,i)=4
   ProcConn(2,i)=-1
   ProcConn(3,i)=-1
   ProcConn(4,i)=-1
end do
call initialize(ProcConn, 100, error)
call send (NETCAST,buffer_atmdat0,1,total_atmdat0,100,error)
call send (NETCAST,buffer_atmdat1,2,total_atmdat1,100,error)
call send (NETCAST,buffer_moldat,3,total_moldat,100,error)
call send (NETCAST,buffer_ffp,4,total_ffp,100,error)
call send (NETCAST,buffer_cffp,5,total_cffp,100,error)
call send (NETCAST,buffer_contrl,6,total_contrl,100,error)
call send (NETCAST,buffer_constn,7,total_constn,100,error)
C      SEND INTEGER*1 VARIABLES/ARRAYS SEPARATELY
       call send (NETCAST,ATYNUM,8,LENGTH9,100,ERROR)
       call send (NETCAST,BONDML,9,LENGTH10,100,ERROR)
       call send (NETCAST,MOLNUM,10,LENGTH9,100,ERROR)
999    write (5,*)'No of iterations =',itrcmp +1
C      SENDS COORDINATES TO NODES
       call send(NETCAST,XO1,42,INT2(length7),100,error)
```

FIGURE 4. Code which broadcasts arrays to nodes

The data required by the nodes is sent in several arrays. The SEND routine broadcasts all the data to all the nodes simultaneously via the BB08 board. The format of the SEND statement is shown below[11]:-

i.e SEND(Destination, Buffer, Bufftype, BuffLen, Timeout, Error)

where Destination contains the id number of the node (if equal to -1 this broadcasts to all the nodes simultaneously), Buffer contains data for the node, Bufftype is a user assignable number to identify the buffer, BuffLen is the length of the buffer in bytes, Timeout specifies the time before timeout occurs, and error returns a specific number if an error occurs.

The variables/arrays downloaded to the nodes are stored in common blocks. A sample of the common block declarations is shown in Figure 5 overleaf. Sending this data to the nodes is not as simple as it might first appear; mainly due to restrictions

imposed by the Microsoft 16-bit FORTRAN which are not present with the 32-bit version.

```
      COMMON/ATMPRP/ EN(MAXTYP)
      COMMON/MOLDAT/ NUMATS,NMOLS
      COMMON/FILDAT/ DLUNIN,DLNOUT,LUNOUT
      COMMON/FILCHR/ INFILE,OUTFIL,FILTYP
      COMMON/HEADER/ TITLE
      COMMON/FFP/ REFLEN(MAXTYP,MAXTYP),STRCON(MAXTYP,MAXTYP)
     1,A6(MAXTYP,MAXTYP),B12(MAXTYP,MAXTYP),REFANG(MAXTYP)
     2,PERIOD(MAXTYP,MAXTYP),BARIER(MAXTYP,MAXTYP)
      COMMON/CFFP/ CREFLN(MXCNJ,MXCNJ),CSTCON(MXCNJ,MXCNJ)
     1,CBARR(MXCNJ,MXCNJ),CPRIOD
      COMMON/CONJTP/ ARTYPS(NARTYP),DBTYPS(NDBTYP)
```

FIGURE 5. Common Block declarations

The simplest approach may appear to be, to send a large array whose start address is the address of the first variable in the first common block. The length (in bytes) of this array would be equal to the total length of all the common blocks. This approach is not possible as although the common blocks will be stored contiguously in memory, they are each assigned to a different 64kbyte wide segment by the FORTRAN compiler and addresses do not automatically roll over from one segment to the next.

Another possible approach might be to dispense with the individual common blocks and put all of the data into one large common block. This is not possible as there is more than 64kbytes of data and the compiler limits each common block to a maximum of 64kbytes in length. In addition to this restriction the COMFORT SEND subroutine imposes a maximum message length of 64kbytes.

A dummy array is therefore EQUIVALENCED to the start of each common block (or the position in the common block where the required data starts). This dummy array is dimensioned to encompass the data by calculating the combined size (in bytes) of all the variables/arrays required from the common block (See Figure 6). An example of the statements necessary to EQUIVALENCE the common block moldat (which was shown in Figure 5) to an array are illustrated in Figure 7 overleaf. Both the variables in moldat are INTEGER*4.

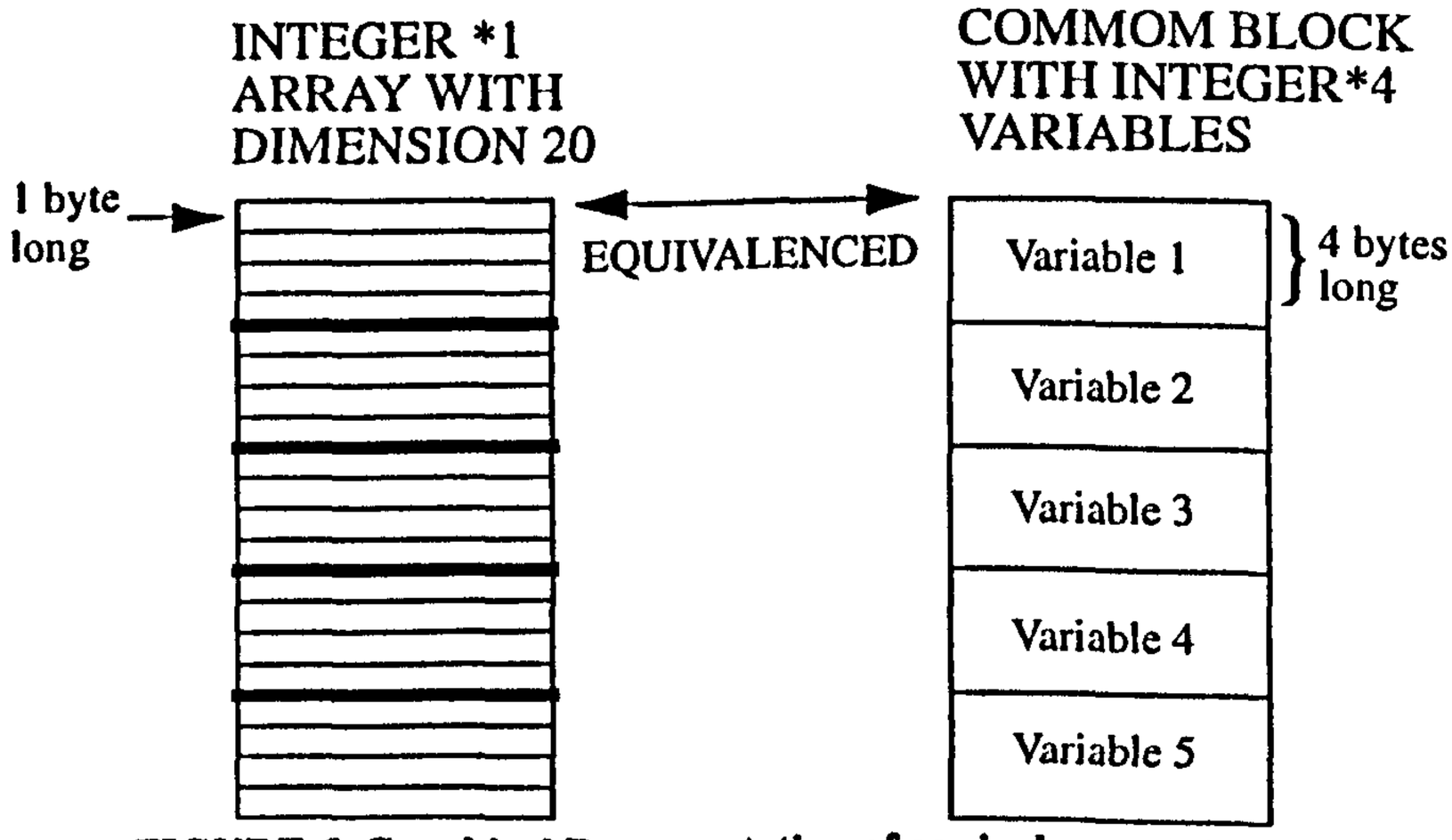


FIGURE 6. Graphical Representation of equivalence statements

The equivalence statements are written in an include file which is used on both the host and node.

```
parameter (length3 = 4)
parameter (total_moldat = length3 * 2)
dimension buffer_moldat (total_moldat)
equivalence (buffer_moldat(1),numats)
```

FIGURE 7. Example of equivalence statements

A further difficulty arises from the fact that the Microsoft FORTRAN compiler adheres rigidly to the FORTRAN standard. If the SEND subroutine is called with a message of one data type then any subsequent call with a message of a different data type will result in a run-time error. In order to overcome this difficulty SEND is always called with messages of type INTEGER*1 which are EQUIVALENCED to the real data array (which contains data of many types). Obviously a different dummy INTEGER*1 array will be required for each common block to be sent as it would be nonsense to EQUIVALENCE all of the common blocks to the one array

3.2 Node Program

The pseudocode for the node program is shown in Figure 8 overleaf.

The node program considers each atom at a time and calculates its corrected coordinates using the Newton Raphson iteration. The first and second partial derivatives are calculated numerically by finite difference methods. i.e.

$$\frac{\partial V_s}{\partial x_i} = \frac{V_s(x_i + \delta x) - V_s(x_i - \delta x)}{2\delta x} \qquad \text{(EQ 9)}$$

$$\frac{\partial V_s}{\partial x_i \partial x_j} = \frac{V_s(x_i + \delta x, x_j + \delta x) - V_s(x_i + \delta x) - V_s(x_j + \delta x) + E(x_i)}{\delta x^2} \qquad \text{(EQ 10)}$$

$$\frac{\partial^2 V_s}{\partial x_i^2} = \frac{V_s(x_i + \delta x) - V_s(x_i - \delta x) - 2E(x_i)}{\delta x^2} \qquad \text{(EQ 11)}$$

where δx is a small value (i.e. 0.001) and i = 1,3N, j = 1,3N. The steric energy is therefore calculated at (x_i), $(x_i + \delta x)$, $(x_i - \delta x)$, $(x_j + \delta x)$ and $(x_i + \delta x, x_j + \delta x)$. The second partial derivatives vary by so little after each iteration that it is sufficient to calculate them after only every 4 or 5 iterations. Once the node has the first and second partial derivatives of the steric energy with respect to the atomic coordinates, it uses the

Newton Raphson equation to calculate the new 'improved' coordinates. These coordinates are then sent back to the host.

```
Initialize node
Receive data from host
Sets up various tables required for the calculation
(i.e. bond lengths, bond angles etc.)
Decide which atoms the nodes will work on
Receive atomic coordinates from host
For J = NFIRST, LAST do
Calculate Energy of Jth atom
For k =1 , 3 do
      Increment kth coordinate of jth atom and      recalculate energy
      Decrement kth coordinate and recalculate energy
      Calculate first derivative for kth coordinate

For k = 1,3 do
       sum of squares of first derivatives =
       sum of squares of first derivatives + (first derivative for kth coordinate)^2
If Mod(Iteration,4).= 0 then
    Calculate second derivatives for jth atom
Calculate corrections to coordinates for jth atom
For k = 1,3 do
      Calculate new value for kth coordinate of jth atom
Send modified coordinates and sum of squares of first derivatives back to host
```

FIGURE 8. Pseudocode for Node Program

The arrays sent from the host are received using the COMFORT RECEIVE routine (See Figure 9 overleaf).

i.e RECEIVE (Source, Buffer, Bufftype, Bufflen, Error)

This is basically the same format as the SEND routine on the host. For every SEND call on the host there has to be an equivalent RECEIVE call on the node.The code which allocates atoms to nodes is shown in Figure 10 overleaf; me is the id number of the node, numproc is the number of nodes, numats is the number of atoms and nfirst and last are the first and last atoms a node will work on. Each node is allocated NUMATS/NUMPROC atoms, with the first NMOD nodes being allocated an extra atom. This distributes the atoms as evenly as possible across the nodes. The number of atoms in a nodes 'slice' is stored in BFLENG.

```
C  INITIALIZE NODES
   call initialize
C  RECEIVES BUFFERS FROM HOST.

   call receive(host,buffer_atmdat0,1,total_atmdat0,error)
   call receive(host,buffer_atmdat1,2,total_atmdat1,error)
   call receive(host,buffer_moldat,3,total_moldat,error)
   call receive(host,buffer_ffp,4,total_ffp,error)
   call receive(host,buffer_cffp,5,total_cffp,error)
   call receive(host,buffer_contrl,6,total_contrl,error)
   call receive(host,buffer_constn,7,total_constn,error)

c     RECEIVE BYTE VALUES SEPARATELY

   call receive(HOST,ATYNUM,8,LENGTH9,ERROR)
   call receive(HOST,BONDML,9,LENGTH10,ERROR)
   call receive(HOST,MOLNUM,10,LENGTH9,ERROR)
    .
    .
    .
191 call receive(HOST,XO1,42,length7,error)
```

FIGURE 9. Code which receives data from host

```
NDIV = NUMATS / NUMPROC
NMOD = MOD (NUMATS,NUMPROC)

IF(me.lt.NMOD)THEN
  NFIRST = (me*NDIV)+me+1
  LAST = ((me+1)*NDIV)+me+1
ELSE IF(me.eq.NMOD)THEN
   NFIRST = (me*NDIV)+me+1
   LAST = ((me+1)*NDIV)+me
ELSE IF(me.gt.NMOD)THEN
   NFIRST = (me*NDIV)+NMOD+1
   LAST = ((me+1)*NDIV)+NMOD
ENDIF

BFLENG=((LAST+1)-NFIRST)
nfirst4 = (nfirst * 4) - 3
```

FIGURE 10. Code to allocate atoms to nodes

3.3 Transfer of atomic coordinates between host and nodes

The atomic coordinates are stored in an INTEGER*4 array (XO (MXAT,3)) on the host which is effectively arranged as three columns for the x,y and z coordinates. This array

is EQUIVALENCED to three INTEGER*1 arrays X01,X02 and X03 (See Figure 11); X01 contains the x coordinates, and X02,X03 the y and z coordinates respectively.

```
equivalence (xo1(1),xo(1,1))
equivalence (xo2(1),xo(1,2))
equivalence (xo3(1),xo(1,3))
```

FIGURE 11. Equivalence statements for XO

To send the atomic coordinates to the nodes the X01 array is used in the SEND routine (See Figure 4). X01 is EQUIVALENCED to the start of X0 and the buffer length in the SEND statement is four times the length of X0. An equivalent RECEIVE statement is required on the nodes (See Figure 9).

When sending the coordinates back from the nodes to the host only the coordinates in the node's 'slice' must be returned and the host must put the returned coordinates in the correct place in XO. The code on the nodes and host which achieves this is shown in Figure 12 and Figure 13 respectively.

```
call send(HOST,xo1(nfirst4),43,bfleng*4,error)
call send(HOST,xo2(nfirst4),44,bfleng*4,error)
call send(HOST,xo3(nfirst4),45,bfleng*4,error)
call send(HOST,sgd1sq,46,4,error)
```

FIGURE 12. Node code to return 'improved' coordinates to host

```
do 321 l=0,numproc-1
      if(l.lt.nmod)then
       nfirst = (l*ndiv)+l+1
       last = ((l+1)*ndiv)+l+1
      else if(l.eq.nmod)then
       nfirst = (l*ndiv)+l+1
       last = ((l+1)*ndiv)+l
      else if(l.gt.nmod)then
        nfirst = (l*ndiv)+nmod+1
        last = ((l+1)*ndiv)+nmod
      endif

      bflength=((last+1)-nfirst)
      nfirst = nfirst*4 - 3
C     RECALCULATE NFIRST FOR XO1(INTEGER*1 SIZE ARRAY)
            call receive(L,xo1(nfirst),43,INT2(bfleng*4),100,error)
            call receive(L,xo2(nfirst),44,INT2(bfleng*4),100,error)
            call receive(L,xo3(nfirst),45,INT2(bfleng*4),100,error)
            call receive(L,temp1,46,4,100,error)
            sgd1sq = sgd1sq + temp
321 continue
```

FIGURE 13. Host code to receive 'improved' coordinates

The x,y and z coordinates are sent separately in X01,X02 and X03. Nfirst4 specifies the position of the first atom in the nodes 'slice' in X01 etc. This value is not just equal to nfirst (the first atom in a nodes slice) as X01 etc. are INTEGER*1 arrays so the value of nfirst needs to be recalculated (i.e. nfirst4 = (nfirst*4) -3). The length of X01,X02 and XO3 is set to BFLENG *4; i.e the number of atoms in a nodes slice multiplied by 4.

4. Results

Table 1 shows the run-time of the parallel minimiser on one node compared to four nodes for 24 and 45 atom molecules. The results illustrate that for a 24 atom molecule a speed-up of approximately 2.5 is obtained whereas for a 45 atom molecule a speed-up of approximately 3 is achieved. The difference in the results is due to the set-up time (i.e. the loading of the required data onto the nodes etc.) which becomes more significant for smaller numbers of atoms. The present version of the minimiser loads the data in an inefficient manner as each node receives more data then is necessary; this will be corrected in future versions.

TABLE 1. Optimisation times for 30 iterations

Number of atoms	Number of Nodes	Run-time of Minimiser
24	1	320s
24	4	129s
45	1	743s
45	4	243s

5. Conclusions

Further improvements to this algorithm could include using the host to carry out the Newton Raphson iteration on a 'slice' of atoms rather than it remaining idle while the nodes are computing. The use of Microsoft Powerstation (32-bit) Fortran would allow the host to send all the variables/arrays in one large array as the compiler 'sees' the address space as contiguous and SEND/RECEIVE operate on messages up to 4Gbytes long.

The use of the COMFORT routines and BB08 board provide faster energy minimisation than the conventional 3L FORTRAN version which uses a pipeline of transputers. Broadcasting code and data simultaneously to all the nodes reduces the run time of the minimisation program considerably.

References

[1] Chem-X. Chemical Design Inc.

[2] Sybyl. Tripos Associates

[3] White, D.N.J. Computer methods for molecular design. *Phil. Trans. R. Soc. Lond.* 1986. A **317**, 359

[4] Engler, E.M., Andose, J.D., Schleyer, P. von R. Critical Evaluation of Molecular Mechanics. *J.Amer.Chem.Soc.* 1973, **95**, 8005

[5] Witberg, K.B. A Scheme for Strain Energy Minimisation. Application to Cycloalkanes. *J.Amer. Chem. Soc.* 1965, **87**, 1070

[6] Lifson, S., Warshel. A. Consistent Force Field Calculations of Conformations, Vibrational Spectra, and Enthalpies of Cycloalkane and n-Alkane Molecules. *J.Chem.Phys.* 1968, **49**, 5116

[7] Mertz, John.E., Tobias, Douglas, J., Brooks, Charles. L., Singh, U.C. Vector and Parallel Algorithms for the Molecular Dynamics Simulation of Macromolecules on Shared Memory Computers. *J.Comp.Chem* 1991, **12**, 1070

[8] Clark, Terry.W., McCammon, J.Andrew. Parallelisation of a Molecular Dynamics non-bonded force algorithm for MIMD architecture. Computers & Chem. 1990, Vol.**14**, No3, 219

[9] Schweitzer, Robert.C., Small, Gary W., Application of Parallel Processing Techniques to Improving the Efficiency of the MM2 Molecular Mechanics Calculations. *J.Comp.Chem.* 1993, **14**, 977

[10] White, D.N.J., Ruddock, J. Noel., Edgington, Paul R. Molecular Design with Transparallel Supercomputers. *Molecular Simulation* 1989, Vol. **3**, 71

[11] White, David N.J. A Hardware & Software Environment for Parallel Processing with PCs. In press, *Computers & Chemistry.*

[12] Express User's Guide. *Parasoft Corporation*, 1990

[13] Walker, D., Dongarra, J. (Convener & Meeting Chair). MPI: A Message-Passing Interface Standard. March 1993, University of Tennesse, Knoxville, Tenessee

Interfacing Electrochromic Spectacles to Computer IO Ports

David N.J. White and Lesley Bissland
University of Glasgow, Department of Chemistry,Glasgow, Scotland
e-mail: lesley@tcrystal.gla.ac.uk

Many important properties of molecules depend on their precise three dimentional(3D) structure. It is therefore useful to be able to view a molecule in 3D on a 2D computer screen when manipulating it. An inexpensive method for viewing in 3D using liquid crystal glasses and a PC is presented. The methodology used is easily extended to other computers and workstations.

Keywords: Liquid crystal glasses, stereoscopy, PC card

1 Introduction

Although red/green stereo is a fairly simple and inexpensive method for viewing in 3D the images produced are monochromatic. This loss of colour can be important in many cases. For example when viewing a molecule, colour coding can be used to signify different atom types. Full-colour clear 3D images can be obtained by using liquid crystal glasses. These glasses are generally quite expensive (~£1000) however by using SEGA video game liquid crystal glasses (~£70) the cost can be cut dramatically.

Chelvanayagam and McKeaig[1] described a method for stereo viewing using the SEGA glasses. Their approach involved modifying the existing SEGA circuit board and connecting a line to the D0 pin on a PC parallel port to toggle the glasses. Whilst this approach is perfectly satisfactory a number of people have reported to the authors that they have been unable to get the modified SEGA circuit board to work. In order to circumvent these problems this paper contains a full circuit diagram of the SEGA control circuit and describes an alternative control circuit built on a PC plug-in card.

2 Stereoscopy

Binocular stereoscopy is a method of generating pairs of two dimensional images which deceive the human eye and brain onto perceiving a three dimensional image. The pairs of images can be generated either side by side or full screen sequentially on the computer screen. The second image is generated from the first by a rotation of 2-6^{o} around the y-axis (x axis horizontal, y axis vertical, both in the plane of the screen). Various methods are

employed to ensure that the left eye sees only one of the images and the right eye only the other. The images can be either stationary or rotating.

The method of stereoscopy used by the SEGA glasses is known as tachistoscopy. This a binocular process in which the left and right eye images are displayed on the screen alternately and the view of each eye is obscured in synchronisation with the display of the 'wrong image'. To achieve this alternate lenses of the glasses are turned opaque at the appropriate moment by an electric field. The frequency of switching between images must be approximately 40Hz to obtain a flicker free image.

2.1 Liquid Crystal Glasses

The liquid crystal glasses [2,3,4] consist of a thin layer of liquid crystals sandwiched between two glass plates. Liquid crystals differ from other compounds in that in a normal crystalline solid the atoms or molecules are in an ordered fixed state, and when the crystal melts the substance goes directly to the disordered liquid phase. In a liquid crystal there is an intermediate phase when there is partial order over a range of temperatures before the liquid phase. This disorder in the liquid crystal, known as the nematic phase, consists of molecules out of position but with the same orientations they had in the solid. (See Figure 1(a)).

Transparent electrodes are evaporated onto the inner surfaces of the glass plates. Tiny parallel scratches on the plates cause the nematic molecules to orient themselves in the direction of the scratches. Since the scratches on one glass plate are perpendicular to those on the other this gives the molecules a twisted structure with the molecules in successive planes turning continuously through 90^{o}. This is known as a "twisted nematic" cell.

Polarising filters sandwich the cell with their axes of polarisation at right angles to each other. Light is polarised as it enters the cell and can then escape from the other side only if the plane of polarisation is rotated through a right angle.

When the electrodes are not applying an electric field, the liquid crystals rotate the plane of polarisation of the light and the light passes through the cell. The glasses are therefore transparent as in Figure 1(b). Applying an electric field forces the molecules to lie parallel to it and to the direction of the light. The plane of polarisation, which is at right angles to the direction of the light is therefore unaffected by the molecules: it is not rotated and light cannot pass through the cell. This causes the glasses to become opaque as illustrated in Figure 1(c).

3 SEGA Circuit

The glasses are operated by a simple circuit controlled by a Z80 microprocessor in the game console. The glasses circuit board plugs into the SEGA console and the glasses are attached to the circuit board by a 3.5mm jack connector. See Figure 2.

Address decoding is achieved by a 13-Input NAND gate (74HCT133') which outputs a logic low (for the purposes of the following discussion logic low means 0V electrical and logic high means 5V electrical) when all the address lines connected to it are pulled high (address 0FFF8h). The output of the 74HCT133 is fed to the least significant address bit (A0) of the 74HCT259 addressable latch. The 74HCT259 will direct the signal on its data input (D) to the latch (Q0-Q7) addressed by its A0-A2 inputs. When the 74HCT259 enable input (E*) is logic low the addressed latch will follow the data input whilst all unaddressed latches will retain their previous state. When E* goes high the logic level on the addressed Q output will be latched (i.e memorized) and unaffected by any further changes in D. As stated previously when the SEGA circuit is being addressed A0 on the latch will be logic low. Since MREQ* is low during a write cycle and the address FFF8 is being used, A0-A2 on the latch will all be logic low which selects Q0 as the output. The logic level on the data pin of the 74HCT259 will therefore be latched on Q0 when the write cycle finishes and WR* goes to logic high. Q1-Q7 are ignored.

Gates 1 and 2 of the 74HCT86 form a simple RC oscillator which produces an approximately 400Hz square wave output at point (A). The waveform is shown in Figure 3(a). Whilst it is possible to switch the glasses from opaque to transparent with a simple DC voltage, this will greatly reduce the life of the liquid crystal cells. Using a 400Hz square wave will prolong life of the cells almost indefinitely.

Section 1 of the LM324 quad operational amplifier is configured to act as a non-inverting level shifting comparator and section 3 as an inverting level shifting comparator. The reference voltage for the two comparators is set to 2.5V by the two 100kΩ resistors. A logic high at point (A) will drive each comparator into positive or negative saturation, depending on whether the comparator is inverting or non-inverting, and a logic low at point (A) will cause each comparator to saturate in the opposite sense. As the LM324 has a 12V power supply the outputs of the comparators will be 400Hz square waves with amplitudes of 12V as shown in Figure 3(b).

The waveform at point (D) depends on whether a logic high ('1') or low ('0') has been written into the 74HCT259 addressable latch. If Q0 of the 74HCT259 is high then gate 3 of the 74HCT86 will invert the output of the oscillator (point (A)) and feed it to section 4 of the LM324 for a further inversion and level shift to an amplitude of 12V. If on the other hand Q0 is low the '86 behaves as a non-inverting buffer and the output of the oscillator is inverted and level shifted by section 4 of the LM324. The waveforms are shown in Figure (b).

The signal at point (D) is applied to both the left and right eye liquid crystal cells of the SEGA glasses, whilst the signal at point (B) is applied only to the left eye cell and the signal at point (C) to the right eye cell only. When Q0 is high, points (C) and (D) are out of phase and an electric field reversing direction 400 times a second will be applied to the right eye liquid crystal cell of the glasses, turning it opaque. On the other hand points (B) and (D) are in phase, the field applied to the left eye cell is zero, and it remains transparent. If Q0 is low the situation is reversed; the left eye cell is opaque and the right eye cell transparent. So by writing a '1' or a '0' to the '259 addressable latch either the left or right eye liquid crystal cell is rendered transparent whilst the other cell remains opaque.

As the voltage needed to run the glasses is approximately 12V, a voltage tripler (Part 2 of Figure 2) is required to increase the LM324 supply voltage from 5V to 12V. The capacitors (C4-C6) charge in parallel and the diodes direct the current so the capacitors discharge in series. This triples the +5V supply voltage, however as 0.6V is lost over each diode the result is an output of approximately 12V.

The modifications to the circuit proposed by C&M are indicated by the dotted lines on Figure 2. The C&M modifications dispense with the address decoder & addressable latch so that the switch signal for the glasses feeds directly into pin 5 of the '86 (gate 3) (from one of the data lines of a PC printer port). The SEGA controller uses the pulsing RD* and WR* strobes of the Z80 processor as a source of alternating current (AC) for the voltage tripler. C&M do not connect the controller direct to a computer bus, so there are no RD* and WR* signals and another source of AC must be found for the voltagè tripler. The 400Hz output of the RC oscillator formed by gate 1 and 2 of the '86 is ideal for this purpose.

However the SEGA controller uses surface mount components and the PCB tracks are very fine and well hidden under a black solder resist. It is easy to see how people unused to surface mount fabrication techniques (i.e most molecular modellers) could make mistakes. In any event we preferred to make up a PC plug in card so that we could still use the printer port, and because the voltage tripler is unnecessary if one uses the 12 volts already available on the PC bus.

3.1 PC Card

The circuit built on a PC plug-in card is shown Figure 4. It varies very little from the one used by the SEGA console. Address decoding is achieved by using an Octal Comparator enabled by the PC bus input/output write signal IOW*, rather than a 13-Input NAND gate. When the address on the P side is equal to that on the Q side and G* is low the logic low output from the comparator is used to enable the latch. The signals A0, A1 and AEN* select the output Q0. As in the SEGA circuit D0 is used to toggle the glasses. This circuit does not require a voltage tripler as an input of 12V can be taken directly from the PC bus.

Another possible method of interfacing the glasses to a PC is to clamp the existing SEGA controller PCB onto a PC prototyping card and use the IBM PC bus signals more or less unchanged. All the data and control signals can be taken directly from the PC apart from MREQ* for which AEN* can be used instead. The address lines from the PC can also be used but in order so as not to restrict the card to a single address, inverters may be selectively added to the address lines as shown in Figure 5. Since the usable addresses of the I/O ports on the PC lie in the range 0200-3FF, A9 is always going to be high.

4 Software

The code to operate the glasses has been incorporated into the CHEMMOD and COMMET [5,6,7]molecular modelling packages. Both packages were developed in house at Glasgow University. COMMET is running on a 386 PC with a DATAPATH graphics

controller and CHEMMOD is running on a PC/AT with a DIGISOLVE graphics controller.

As described by Chelvanayagam and McKeaig the vertical retrace(VR) of the electron gun is used as a signal to switch the glasses. For CHEMMOD the monitor is interlaced, therefore the glasses are switched after every second vertical retrace. The DIGISOLVE graphics controller contains two display buffers in which images can be stored. The left and right eye images are drawn on separate display buffers and then the contents of each buffer are displayed on the screen alternately. Code to poll the bit in the register associated with the VR and to toggle the glasses was written in 8086 assembler for CHEMMOD. (See Figure 6).

Bit 2 of the status register signals when a vertical retrace is occurring (i.e when bit 2 is logic high a VR is occurring, when logic low the image is being drawn). This bit is polled to establish when an image is complete. See Figure 7. When the electron beam has finished drawing the image the display buffer is switched in order that the other half of the stereo image is displayed next time round the loop. The glasses are then flipped by sending a logic high or logic low (depending on the value in CL) to the glasses port. The state of the mouse buttons are polled in order to exit the routine by pressing a mouse button.

For COMMET the code was written in C and was almost the same as Chelvanayagam and McKeaigs'. The monitor used was non-interlaced and the graphics card contained four display buffers but for stationary images only two were needed.

The routines to draw the left and right eye images for both CHEMMOD and COMMET were written in FORTRAN. Since both packages contained code for red/green stereo it was relatively simple to modify this code for colour stereo images. The main difference in the algorithms of the stereo systems is that, for the liquid crystal glasses, both images have to be drawn in colour and in separate display buffers whereas for red/green stereo both images are drawn in the same display buffer.

5 Discussion

Even when viewing complex molecules (1800 atoms) the 3D stationary images produced are excellent. However for proteins with many double bonds drawn as such the images appear cluttered, therefore better results are obtained by representing the double bonds with single vectors. When viewing complex rotating images on a 386' PC using four display buffers the rotation appears jerky. This problem could be solved by using a faster PC.

Our method has the advantage over Chelvanayagam and McKeaigs' that the card is housed inside the PC so there are no stray wires present. Chelvanayagam and McKeaig also mentioned that it would be possible to have more than one viewer using their system, and this is possible for the PC card plug in system. An additional LM324 and glasses jack

is required for each extra viewer. It is also relatively straightforward to use an IR light beam rather than wires to connect the glasses to the controller.

6 Conclusions

Although SEGA are no longer manufacturing the glasses, some pairs are still available commercially (through Molecular Design, Oxford), various US companies also have hoards, and it should be possible to obtain supplies via newspaper small ads. As a last resort you could always buy some of the rather expensive models still being made by manufacturers other than SEGA. Future developments could involve combining the glasses with a spaceball in order to manipulate the molecule in 3D space: spaceballs are, however, relatively expensive.

Acknowledgments

The authors would like to thank Mr. J N Ruddock and Dr. K J Tyler for their help and advice. L.B would also like to thank SERC for the award of a Research Studentship.

References

[1] Chelvanayagam, G. and McKeaig, L. Stereo viewing on the PC/AT with EGA graphics. *J. Mol. Graphics* 1991, **9**, 111-114

[2] Harris, M.R., Geddes, A.J and North, A.C.T. Frame sequential stereoscopic system for use in television and computer graphics. *Displays - Technology and Applications* 1986, Vol. **7** No 1, 12-16

[3] Harris, M.R., Geddes, A.J and North, A.C.T. A liquid crystal stereo viewer for molecular graphics. *J. Mol. Graphics* 1985, **3**, 121-2

[4] Guinier, A. *The Structure of Matter: from the blue sky to liquid crystals.* Edward Arnold (Publishers) Ltd., London, 1984 (ISBN 0 7131 3489 5)

[5] White, D.N.J., Computer methods for molecular design. *Phil. Trans. R. Soc. Lond. A* 1986, **317**, 359-369

[6] White, D.N.J.,Tyler, J. Kelvin, Lindley, Matthew R., High Performance Microcomputer Molecular Modelling. *Computers & Chemistry* 1986, Vol. **10**, No.3, 193-199

[7] White, D.N.J., Pearson J.E, *J.Mol.Graphics* 1986, **4**, 132-142

Figure 1(a):Liquid Crystals

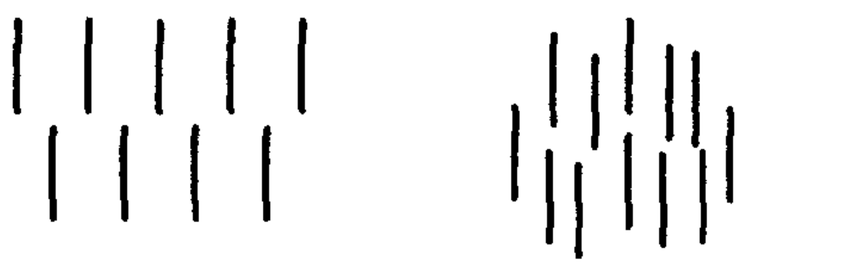


Figure 1(b):Twisted Nematic Cell

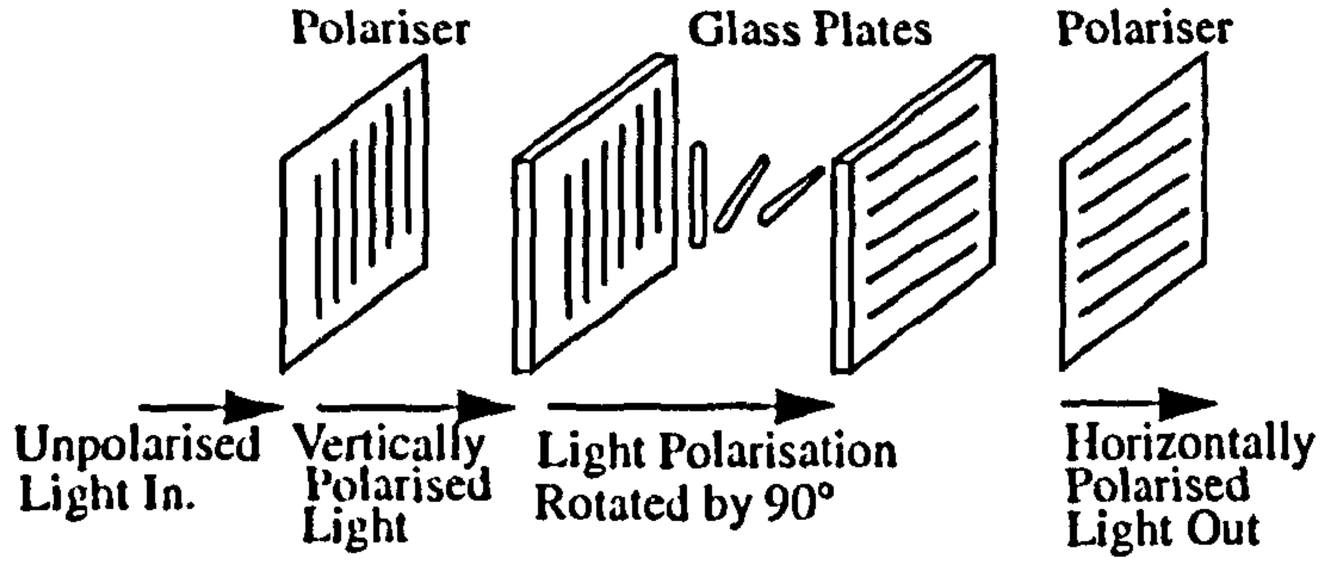


Figure 1(c):Cell with Electric Field On:

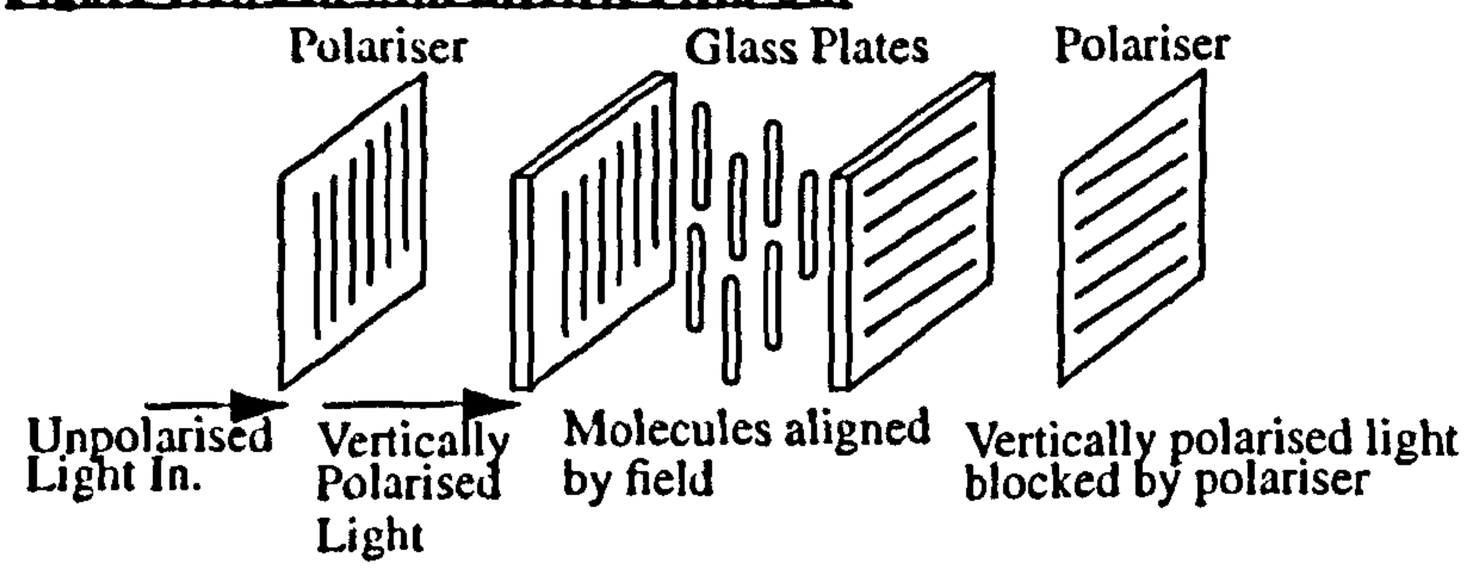


Figure 2: Sega Circuit

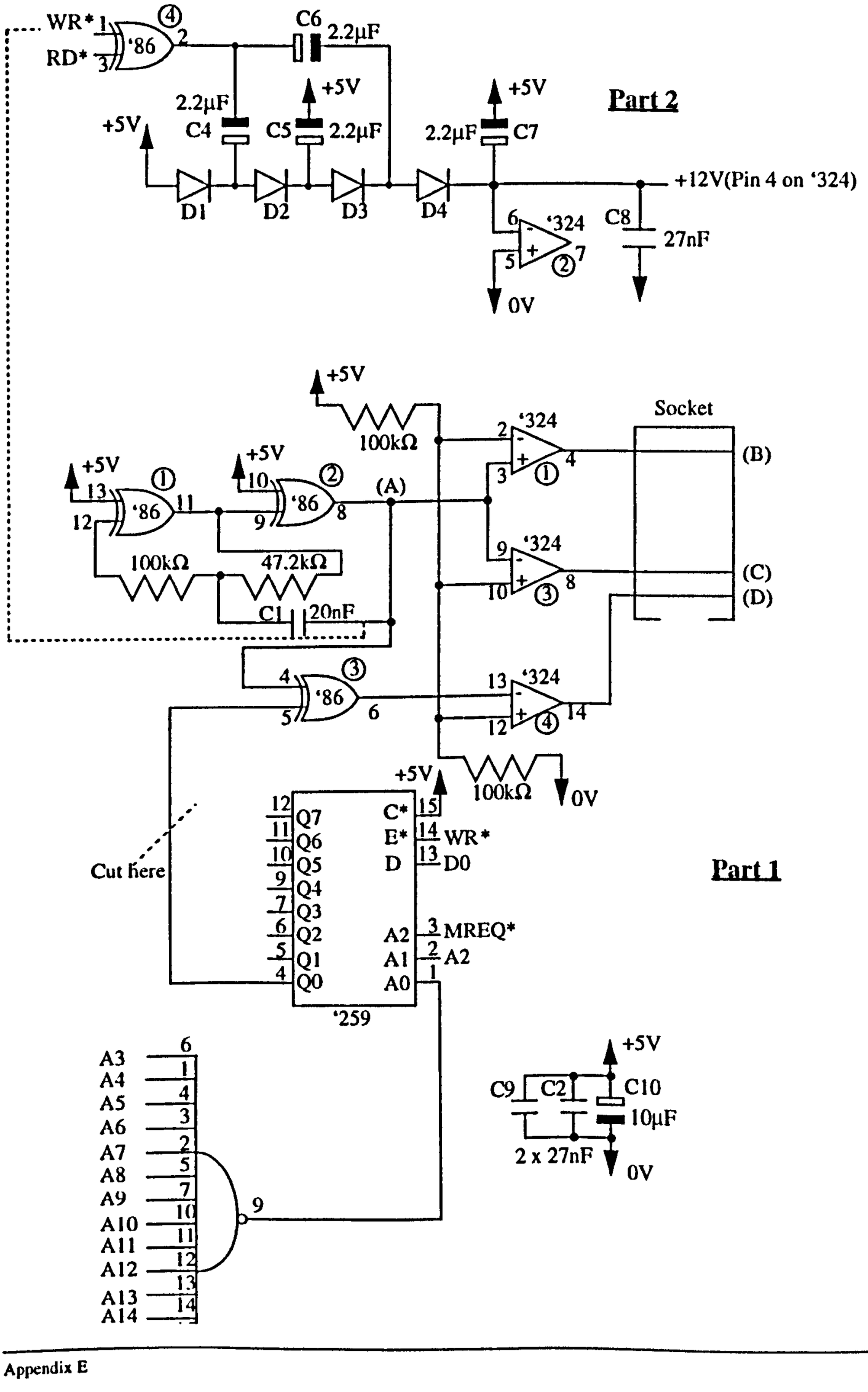


Figure 3(a) ; Waveform at Point A (Fig. 2)

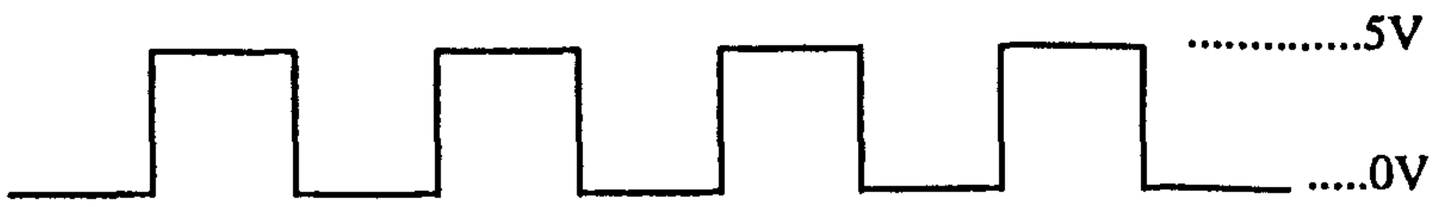


Figure 3(b): Waveforms at Points B, C & D (Fig. 2)

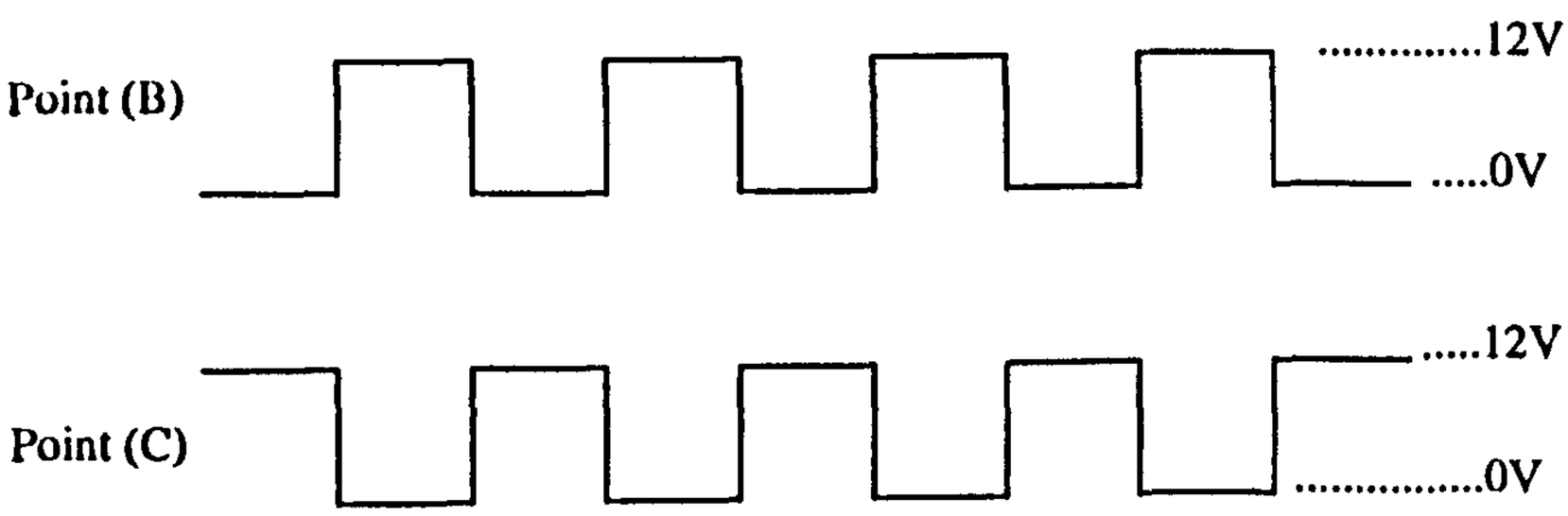


Figure 4: PC Circuit

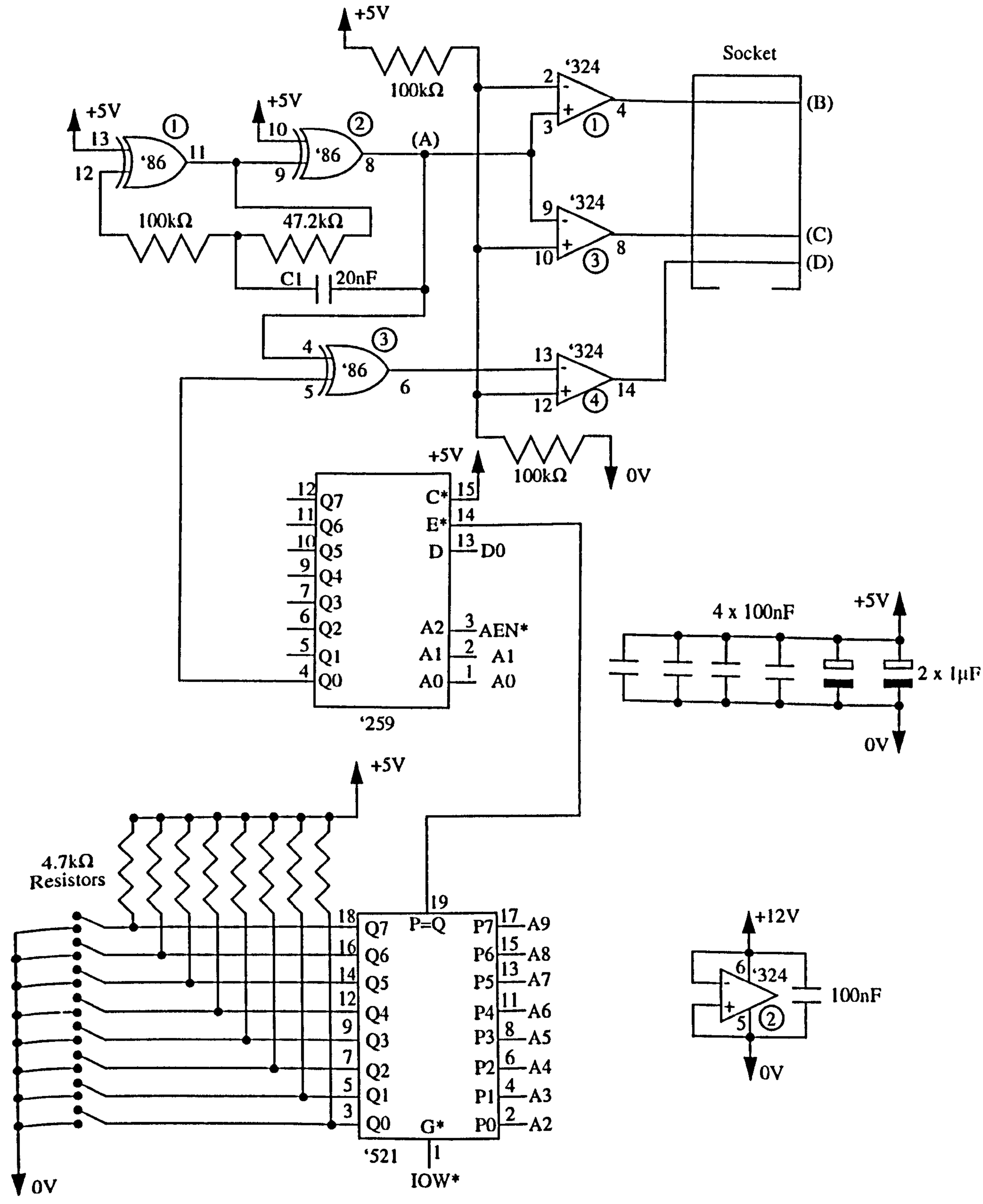


Figure 5: Alternative Address Decoding

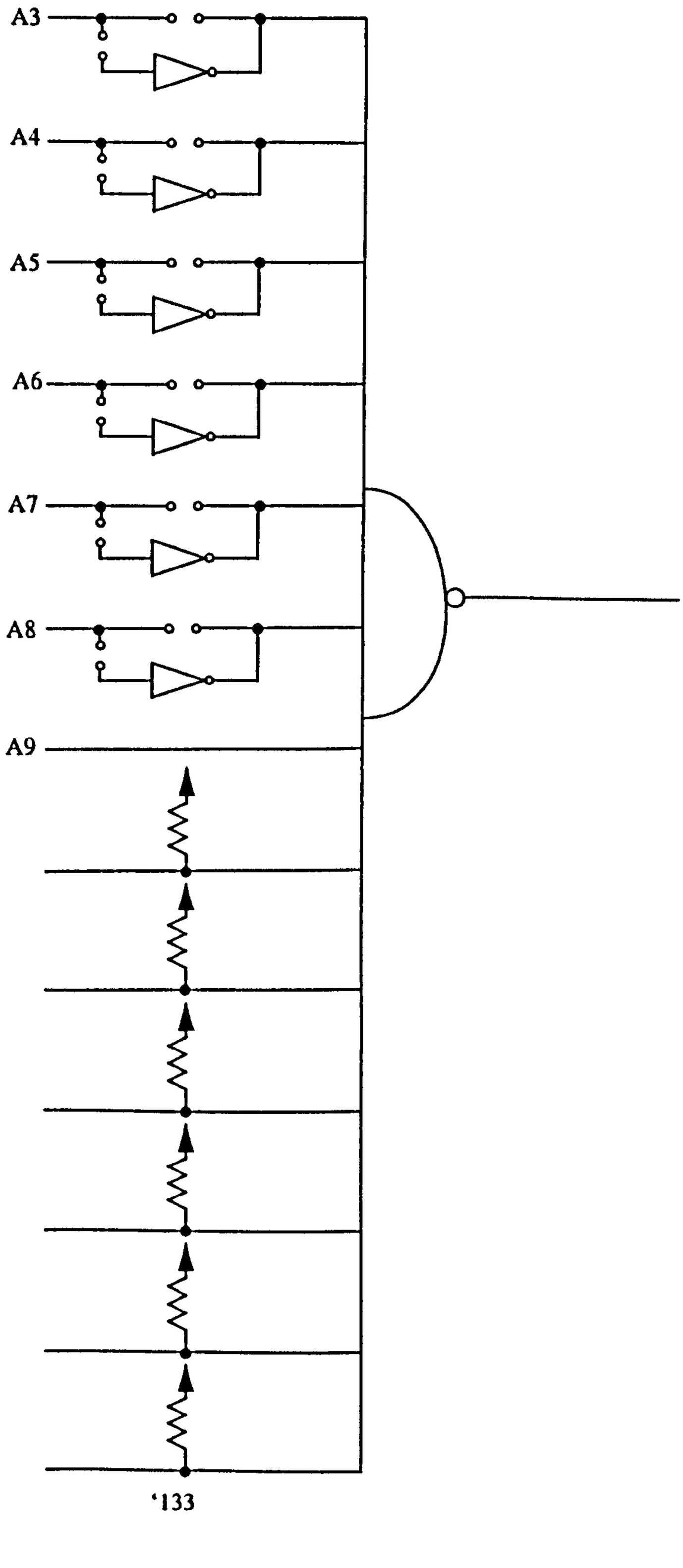


Figure 6: 8086 Assembler Code

```
                PUBLIC    _swchgl
_swchgl         PROC FAR
                PUSH      BP
                MOV       BP,SP          ; Set up stack frame
                MOV       CH,03H         ; Set plane 1 value
                MOV       CL,OOH         ; Set plane O value

test_vb :       MOV       DX,310H        ; Load status port address
vb_is_1a :      IN        AL,DX          ; Read status register
                TEST      AL,02H         ; Test if on
                JZ        vb_is_1a       ; Jump back if not 1

vb_is_Ob :      IN        AL,DX          ; Read status register
                TEST      AL,02H         ; Test if off
                JNZ       vb_is_Ob       ; Jump back if not

vb_is_1b :      IN        AL,DX          ; Read status register
                TEST      AL,02H         ; Test if on
                JZ        vb_is_1b       ; Jump back if not 1

vb_is_Oa :      IN        AL,DX          ; Read status register
                TEST      AL,02H         ; Test if off
                JNZ       vb_is_Oa       ; Jump back if not O

                MOV       AL,CL          ; Get value to output in AL

                MOV       DX,300H        ; Load page port address
                OUT       DX,AL          ; Switch display page

                MOV       DX,150H        ; Load glasses port address\
                OUT       DX,AL          ; Switch glasses

                XCHG      CL,CH          ; Swap page number in CL

                PUSH      CX             ; Save CH, and CL
                MOV       AX,3           ; Load mouse function code
                INT       33H            ; Call mouse interrupt
                POP       CX             ; Restore CH, and CL
                CMP       BX,O           ; Check the button status
                JE        test_vb        ; Exit if one is down

test_bu :       MOV       AX,3           ; Load mouse function code
                INT       33H            ; Call mouse interrupt
                CMP       BX,O           ; Check the button status
                JNE       test bu        ; Exit when all up

                POP       BP             ; Restore previous stack frame

                RET

_swchgl         ENDP

OVL8 TEXT       ENDS

                END
```

Figure 7: Status Register Polling