A Parallel Processor System
For Nuclear Shell-Model Calculations

Douglas James Berry

Department of Physics and Astronomy

University of Glasgow

Presented for the degree of

Doctor of Philosophy

August 1988

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Acknowledgements

I would like to express my thanks to my supervisor Dr. A.M. MacLeod for the support and assistance given to me over the past six years and also to Prof. R.R. Whitehead for his help and advice with the nuclear physics. I am grateful to Dr. L.M. MacKenzie for his guidance and the useful discussion which have contributed to my work. The actual production of the hardware and the circuit diagrams was performed with great patience and good humour by Ian Smith and Tony Reilly to whom I am indebted, and also to the other members of Room 18 who gave technical support. In particular I am grateful to Ian for doing most of the diagrams for this thesis. I would like to thank the Head of the Department for the use of departmental facilities. Also SERC provided my Research Studentship as well as the grant to fund material spending on the project. Motorola Semiconductors of East Kilbride supplied a number of their products in addition to providing useful information on future product development.

Thanks are also due to my current employer, the Marconi Research Centre in Chelmsford, for the use of their facilities, in particular for the copying of this thesis. Also to my Group Chief for his encouragement over the last two and a half years.

Last, but by no means least, I would like to thank my wife for her understanding and support throughout the prolonged production period of this thesis. For this I am especially grateful.

D.J. Berry

CONTENTS

Abstract

This thesis describes the design and implementation of a dedicated parallel processor system for nuclear shell-model calculations. The purpose of these calculations is to determine nuclear energy eigenvalues by the tridiagonalisation of the nuclear Hamiltonian matrix using the Lanczos method. The Theoretical Nuclear Structure group at Glasgow University's Physics Department would normally perform this type of calculation on a high-performance main-frame computer. However these machines have limitations which restrict the number and scope of the calculations that can be performed.

The Shell Model Processor system consists of a Multiple Microprocessor Unit (MMPU) driven by a highly pipelined dedicated front-end processor. The MMPU has a modular, moderately coupled, MIMD architecture based on autonomous processing modules. The elements within the system communicate via three shared buses. The front-end is responsible for determining the position of non-zero elements within the Hamiltonian matrix. Once the position of an element has been found it is passed to one of the free processing modules within the MMPU. The processing module then determines the value of the matrix element and performs the appropriate arithmetic to accumulate the resultant Lanczos vector. Two such processing modules have been developed. The most recently developed module is based on two MC68000 16/32 bit microprocessors. In addition there are two supervisory processor modules, one of which controls the front-end and also assists it in its function. The other module has privileged system capabilities and is responsible for supervising the system as a whole.

The system has been successfully tested and performance figures are presented. The future expansion of the system to allow it to perform larger calculations is also discussed.

CHAPTER 1

A Review of Parallel Computer Systems

1.0 Introduction

In the 42 years since the introduction of the first electronic digital computer until the present day "supercomputers", arithmetic processing speed has undergone a dramatic increase of over ten million fold. Such an increase has not been achieved solely by the improvements in performance of electronic digital hardware, e.g. the introduction of discrete transistors in 1960, of small-scale integrated circuits in 1965, and of VLSI and VHSIC devices in the 1980s. Rather this increase has been made possible by the marriage of these technological achievements with the introduction of parallel processing techniques at all levels of computer architecture. For example, the Goodyear Aerospace Massively Parallel Processor (MPP) being delivered to NASA is centered around a 128 x 128 (= 16,384) array of bit-serial processing elements (PE), with 8 of these PEs packaged on a single custom VLSI CMOS-SOS chip. Developed primarily to process satellite imagery, it is capable of performing over 6.5 billion additions per second and 1.8 billion multiplications per second on 8-bit integer data. While on 32-bit floating-point numbers it can perform 430 million additions per second and 216 million multiplications per second, [Bat80, HLSM82].

Performance improvements in the last forty years due to technological enhancements alone can be estimated to be a factor of between one and ten thousand, [HJ81]. This would conservatively place the speed up factor due to parallelism at about 1000. Parallelism is now

common place to the extent that it is now embedded even in conventional serial computer architectures; serial in that they execute one instruction at a time, but parallel in that instruction fetch, decode and execution are all pipelined.

However it must be borne in mind that it is the technological advancements that have made much of the parallelism feasible. For example VLSI microprocessors have made multiprocessor systems not only feasible but widely available and the late 1970's and 1980's have seen a proliferation of experimental and commercial multiprocessor systems based on commercially available microprocessors. Indeed the microprocessor manufacturers are very much aware of this and most of the 16 and 32 bit processors have hardware and software features included in their design that facilitate their use in multiprocessor systems. In fact the Inmos Transputer series of microprocessors is designed specifically for multiple processor applications and is described as a "system building block" [BCMW83].

This thesis discusses one such experimental multiprocessor system which is based around commercial microprocessor devices. As an introduction this first chapter will give a brief history of the advances in parallel techniques as well as an overview of multiprocessor configurations and bus structures.

1.1 A History of Parallelism

The first computers to be built which were designed around the classical, serial *von Neumann* architecture were EDSAC (Cambridge, 1949) and EDVAC (Pennsylvannia, 1952). Prior to this the only digital computer built, ENIAC (Pennsylvannia, 1946), did not have a stored program but was wired up for specific computations. Hence any alteration of the program required rewiring [Ro69].

Having the program stored in memory, as with EDSAC and EDVAC, was

obviously much more flexible and is one of the features of the von Neumann architecture. There are five basic units within this architecture, namely;

1/ an input device for reading data and instructions from the outside world into memory,

2/ an output device for sending results and messages to the outside world,

3/ a single memory for storing both program and data,

4/ a single ***Control Unit (CU)*** for interpreting instructions,

5/ and a single ***Arithmetic-Logical Unit (ALU)*** for processing data.

The last two units are collectively referred to as the ***Central Processing Unit (CPU)***.

In the two von Neumann machines mentioned each of the five units operated one at a time. Even their arithmetic was performed in a bit serial manner, with the addition of two numbers requiring one machine cycle per bit. This was due mainly to the fact that their memory consisted of a mercury delay line acting as a shift register, and therefore data was read serially bit by bit with the least significant bit being accessed first. Bit-parallel arithmetic was first used in the experimental IAS machine (Princeton, 1952). This used electrostatic cathode ray tube storage from which 40 bit words could be read in parallel. The first commercial computer to use bit-parallel arithmetic was the IBM 701 introduced in 1953.

The next step in parallelism and the first departure from the von Neumann architecture was the addition of ***data channels***. Up until that point all I/O requests to peripheral equipment e.g. card readers, line printers and drums, had to be processed by the CPU. Even with relatively fast peripherals, such as magnetic tape drives, I/O could cause a major bottleneck in the processing of data. This problem was partly solved by introducing data channels. Data channels had their own separate processing unit and instruction set and also had shared access with the

CPU to the main memory. Once the CPU had started the data channel transferring blocks of data, the CPU could then proceed to operate independently of it, thus allowing concurrency between I/O and computational processes. IBM first introduced such channels in their 709 machine in 1958, and the technique is still used in many modern computers.

The next architectural advance took place shortly afterwards with the Univac Larc (1960) and the IBM Stretch (1961), [Ro69, HB87]. These two machines further departed from the von Neumann structure by introducing ***interleaved memories*** and an ***instruction pipeline***. Interleaved memories, essentially the application of parallelism to the primary memory system, divides the primary memory up into 2 or more independently accessible banks. Thus program words in successive memory banks can be accessed in a pipelined manner, reducing the limitation placed by slow memory technology on the processor cycle time. The instruction pipeline (or lookahead) allowed the current instruction to be executed in parallel with the fetching and decoding of the next few instructions. However neither the Univac Larc nor the IBM Stretch were commercially successful with the Stretch being superseded by the IBM 7094 in 1962.

In the same year Burroughs introduced what can be considered the first multiprocessor system with the introduction of the D-825, [Ba80]. Intended primarily for military applications, it could have up to 4 identical CPUs connected to 16 memory modules via a ***cross-bar switch***. The cross-bar switch was used later in the two procesor Burroughs B-5000 as well as in a number of other multiprocessor systems.

Functional parallelism within the CPU was first introduced, to a limited extent, in the ATLAS computer, [HJ81]. A prototype was first built at the University of Manchester in 1961 under the direction of Professor Kilburn and the computer then went into production with Ferranti in 1963. The ATLAS had magnetic core memory which was divided

into 4 independent, interleaved banks. More important, however, was the introduction of a separate 24-bit adder for address calculations (the B-unit) which worked in parallel with the main 48-bit fixed/floating-point arithmetic unit. An operand address was formed in the B-unit by adding the contents of one or two of the 128 24-bit index registers to a 24-bit address which was contained in the instruction word. The inclusion of these independent functional units along with the use of pipelining allowed four separate phases of instruction execution to be overlapped, namely; instruction fetch, operand address calculation in the B-unit, operand fetch and operation of the 48-bit arithmetic unit.

The ATLAS is also known as the first machine to have a virtual memory system. This gave the user the appearance of having a large (approximately 1 million words) single level primary memory system. In reality the operating system translated memory references to the virtual single level system to a multilevel store consisting of magnetic core, magnetic drum and tapes. Data was transferred between the different levels of the physical storage system in 512 word pages.

The idea of functional parallelism was utilised to a much greater extent in the CDC 6600, introduced in 1964. This machine had a set of 10 dedicated arithmetic functional units for performing multiplication, division, addition, shifting and boolean operations amongst others on 60-bit floating-point numbers. These were controlled by a hardware mechanism which allowed independent instructions to be executed out of sequence without altering the logic of the program yet making most efficient use of the separate functional units. The controller had a "scoreboard" by which it kept track of the availability of the different functional units and registers and thus avoided conflict between the various instructions which were being executed. In addition the CDC 6600 had 32 interleaved memory banks and 10 Peripheral Processors Units (PPU). The PPUs each had their own private memory and executed separate programs while sharing a common arithmetic unit and access to the main

memory on a time-multiplexed basis. The CDC 6600 was replaced in 1969 by the CDC 7600. This was upwardly compatible with the CDC 6600 but replaced the serially organised functional units with fully pipelined ones. The CDC 7600 also had solid state memory devices instead of the magnetic core memory used in the 6600 and had a processor cycle time that was four times faster. The CDC 6600 and 7600 were very popular, powerful machines and many of the ideas found in their architecture were used in later computers.

The chief architect of the CDC machines, Seymour Cray, later left to start his own company, Cray Research Inc., and in 1976 produced the Cray-1, [HB87, KT80]. This follows in the steps of the 7600 but has a processor minor cycle time of 12.5 ns which is twice as fast as that of the 7600. The Cray-1 also includes ***vector processing*** hardware and instructions. That is as well as incorporating hardware for processing data which consists of single numbers (scalars), there is also hardware for processing data which consists of ordered sets of numbers (vectors). The Cray-1 has 12 independent, pipelined functional units with the ability to chain the units together so that intermediate results from one unit can be passed immediately for processing in another unit without reference to primary memory. Three of the functional units are reserved for vector operations (add, shift and logical), while three are shared between scalar and vector 64-bit floating-point operations (add, multiply and reciprocal approximation, there being no divide unit). In support of the vector units there are 8 vector registers, each containing sixty four 64-bit floating-point numbers. The Cray-1, considered a second generation vector processor, has a maximum processing rate of 160 Million floating-point operations per second (MFLOPS) and can achieve rates in excess of 100 MFLOPS for matrix multiplication.

There were two earlier pipelined vector processors, the CDC Star 100 whose design was first conceived around 1964 and the Texas

Instruments Advanced Scientific Computer (TIASC), which started around 1966. Both of these were first delivered around 1973 and both suffered from old technology, e.g. the Star 100 had core memory compared to the Cray's bipolar memory. Consequently neither were as fast as the Cray-1 for either scalar or vector operations. The Star 100 was designed to work at up to 100 MFLOPS but only averaged around 20 MFLOPS while the TIASC, designed to reach 50 MFLOPS, averaged around 40 MFLOPS, [HB87]. The Star 100 was later improved and re-introduced as the Cyber 203, which in turn was improved to become the Cyber 205 (1981).

In the meantime another form of parallel processing had been developing, that is the ***array processor***. Originally conceived by Unger in 1958, his proposal was for a two-dimensional array of Processing Elements (PE) each connected to its four nearest neighbours and all controlled by a common master, [HB87]. Each PE was synchronised to all the other PEs by the master to perform the same function in parallel on their own local data. The proposal was further developed by Slotnick et al in 1962 in their design for the Solomon computer [HJ81]. This was to be a two-dimensional array of 32 x 32 PEs each with its own 128 32-bit word memory and bit-serial arithmetic unit. Every PE would follow the same instruction stream which was supervised by a central control unit. The spatial parallelism of the array processor was a revolution in computer architecture unlike the evolution of the serial processor to the pipelined vector processor. However neither Ungers nor Slotnicks design were ever implemented in full and it wasn't until 1972 that the first array processor, Illiac IV, was built. Originally proposed by Unger for pattern recognition problems, array processors are well suited for certain vector processing applications and grid problems, e.g. matrix problems, Fourier analysis, image processing and weather simulation. However the difficulty in programming array processors and the parallel lockstep operation of the PEs limits their overall range of applications and has restricted them to be special purpose machines.

The original Illiac machine was intended to have four arrays of 64 PEs. Each 8 x 8 array was to have its own CU with its own instruction stream. The PEs would have their own floating-point arithmetic unit and 2048 (2K) 64-bit words of memory and would communicate with their four nearest neighbours. The objective was for a processing rate of up to 1000 MFLOPS working on vector or matrix computations. However the machine which was eventually built, the Illiac IV, only had one of the intended 64 PE arrays and had a peak processing rate of the order of 50 MFLOPS.

Based on the lessons learnt from building the Illiac IV Burroughs went on to design and build the BSP array processor [KT80]. One of the problems with the Illiac IV was the delay involved in transferring data between memories separated by long distances across the array. In the BSP the problem was solved by reducing the number of processors (called arithmetic elements (AE) on the BSP) to 16 and having 17 memory banks connected by an alignment network (a full crossbar switch). This allowed each AE to access every memory without any routing delay and by using 17 memory banks (the next highest prime number above 16) and appropriate mapping algorithms for storing the data memory conflicts are reduced. By pipelining memory accesses with AE processing the BSP was designed to have a maximum processing rate of 50 MFLOPS.

Other array processors have also been developed. For example the ICL Distributed Array Processor (DAP), first produced in 1980, is very similar to the original Solomon design, with a 64 x 64 array of bit-serial PEs connected to four nearest neighbours. Larger arrays of 128 x 128 or even 256 x 256 using 4 PEs per LSI chip have also been proposed for the DAP, [HJ81]. Some multiple array processors, along the lines of the original Illiac design, have been proposed but as yet never built e.g. the MAP and the Phoenix [HB87].

The main architectural elements of parallel processors have now been introduced in essentially chronological order. It is useful to

order these ideas by classifying the various computer organisations into different categories and in the next section two classification schemes are presented.

1.2 Classification of Computer Architecture

A number of different classification schemes have been proposed, each with their own merits and deficiencies, e.g. Flynn's [Fl66], Feng's [HB87] and Shore's [HJ81]. Some other classification schemes are much more detailed and involve descriptive languages of varying complexity by which each individual computer is described. For example PMS (a computer hardware descriptive language intended for any computer system, serial or parallel, [Ba80]), and Hockney and Jesshope's own structural notation [HJ81].

1.2.1 Feng's Taxonomy

Feng classifies a computer according to the degree of parallelism within its architecture. The maximum parallelism degree P is defined as the maximum number of bits that a computer system can process within unit time (usually one processor cycle). P can then be given by the product of the computer word length n and the bit-slice length m. The word length is the number of bits contained in the computer word and the bit-slice length is essentially the number of words being processed in parallel. The pair (n,m) then classifies a given computer architecture according to its degree of parallelism. There are four main categories within this classification :

1/ Word-serial and bit-serial (WSBS) ; $n = m = 1$

One bit is processed at a time in this category e.g. the Minima computer.

2/ Word-parallel and bit-serial (WPBS) ; $n = 1, m > 1$

One bit each from m words are processed in parallel in this

category, sometimes called bit-slice processing. The ICL DAP (m = 4096) and Goodyear MPP (m = 16384) are both WPBS machines.

3/ Word-serial and bit-parallel (WSBP) ; n > 1, m = 1

Conventional serial computers which process one word at a time are placed in this category. An example is the VAX 11/780.

4/ Word-parallel and bit-parallel (WPBP) ; n > 1, m > 1

In this category m n-bit words are processed in parallel. This includes array processors with bit-parallel PE's such as the Illiac IV. It also includes vector processors such as the TIASC, and also multiprocessor systems such as the original Burroughs D-825 and the later Carnegie Mellon University C.mmp system developed in the 1970's.

1.2.2 Flynn's Taxonomy

Flynn's taxonomy classifies a computer into one of four main categories according to the multiplicity of its instruction and data streams. An instruction stream is a sequence of instructions executed by the machine and a data stream is a sequence of data processed by an instruction stream. Flynn's taxonomy appears to be the most popular but is by no means completely definitive and is sometimes augmented by adding subdivisions to the main categories. The four main categories are :

1/ Single Instruction stream/Single Data stream (SISD).

This category represents most serially organised, single processor computers. It includes computers which use pipelining within the CU and ALU, since there is still only one instruction stream operating on one data stream. It even includes computers such as the CDC 7600 which have multiple functional units.

2/ Single Instruction stream/Multiple Data stream (SIMD).

This category primarily includes array processors, such as the Illiac IV and ICL DAP. That is there is a single CU which controls a single instruction stream. The CU broadcasts the instruction to

every PE and the PEs then operate on different sets of data.

3/ Multiple Instruction stream/Single Data stream (MISD).

This category implies that a number of instructions are operating simultaneously on a single data stream. Baer [Ba76] considers that pipeline processors could be included in this category if the consecutive stages are considered separate instructions, however Flynn [Fl66, Fl72] himself gives no positive examples of the architecture.

4/ Multiple Instruction stream/Multiple Data stream (MIMD).

In MIMD architectures several CPUs operate in parallel on different (although not necessarily unconnected) data sets. Multiprocessor systems are therefore classified in this category, e.g. the Cm* system [Fu78].

Flynn's taxonomy is useful in that it clearly distinguishes between certain types of parallel processors, e.g. array processors and multiprocessors, whereas Feng's taxonomy lumps most of the parallel processors in one category, i.e. (WPBP). However it is still a fairly loose definition in that the SISD category includes conventional serial processors, pipelined processors and processors with multiple functional units. The SISD class can even include vector processors, depending on whether a vector is defined as a single data stream or not. The MIMD category is also too broad and most writers further subdivide this for clarity into ***loosely coupled systems*** and ***tightly coupled systems*** (or distributed memory multicomputers and shared memory multiprocessors respectively [Hw87]). The next section will discuss in greater detail the MIMD class of computers.

1.3 Multiple Processor Systems

MIMD processor systems vary extensively in the degree and nature of the coupling and interaction between processors. This coupling determines

the extent to which the various elements in the system share resources and cooperate in performing a task. Thus MIMD systems can be further classified according to the degree of coupling between processors [FK83].

1.3.1 Loosely Coupled Systems

Each processor within a loosely coupled system possesses its own local I/O devices and its own local memory systems which will be large enough to store any programs and data that are being processed. Thus each processor is an autonomous computer module in its own right. Each computer module is connected to a communications net by which it can communicate directly or indirectly to any of the other modules in the system. The modules can be geographically distributed and processes which run on the different modules may communicate with each other by passing messages over the net.

The net, which is usually a high-speed serial link such as Ethernet [MB76], will have a strictly defined transfer protocol with each computer module having its own communications net controller. In this way the net itself is passive and the control, i.e. arbitration and message routing, is distributed throughout the system. The communications net for a loosely coupled system can usually tolerate only a low rate of interaction between tasks, otherwise its performance will be degraded. Loosely coupled systems are also referred to as ***distributed systems*** [HB87].

1.3.2 Tightly Coupled Systems

Processors in such a system communicate with each other via a global primary memory system which they access over an interconnection network. This interconnection network must provide a means of communication between all processors and all memory modules within the system. Individual processors may also have their own small, private memory or cache. I/O devices and any other system resources are generally shared

by the processors, although some devices may be dedicated to specific processors. Each processor is supervised and controlled by a single common operating system. Software/hardware means are provided for synchronising cooperating processes which are being executed on different processors. Since most resources are common and all processors have equal processing power, dynamic load sharing is possible under control of the operating system.

In a tightly coupled system data is passed between processors via the global memory, thus the rate at which interprocess communications can take place is determined by both the bandwidth of the memory system and the bandwidth of the processor-memory switching network. The network must resolve any contentions that arise when two or more processors attempt to access the same memory module. Memory contention is a major limitation on the performance of tightly coupled systems and imposes an upper limit on the number of processors that can usefully be included in a system. Thus the switching network should be designed so as to reduce the number of contentions as much as possible. Any contention which does occur between requests must be arbitrated as quickly as possible and should be invisible to the competing processes.

1.3.3 Moderately Coupled Systems

In between the two extremes of tightly and loosely coupled systems there lies a range of organisations which can be termed moderately coupled systems. These systems are suited to processes where the workload can be partitioned into relatively independent tasks which require only a limited amount of communication between them. In general the processing elements will be self-contained, with their own processor and memory for both data and program. Each element may have its own I/O capabilities or there may be a processor (or processors) which is dedicated to this task. Other processors may be dedicated to specific tasks which are necessary to the overall performance of the process. Interprocessor

communications and communications to global resources are performed over the communications net. In general much of the load sharing is static since some functions are carried out by specific processors. Such moderately coupled systems are also called ***Multiple Task/Multiple Data (MTMD)*** systems [FK83], since they are capable of concurrently executing a number of tasks on different data.

1.3.4 MIMD System Characteristics

A multiprocessor system can at most have a linear increase in performance for increasing the number of processors, i.e. n processors will perform n times faster than one processor. This is the ideal, but in practice the law of diminishing returns will operate so that as more processors are added overall system performance will start to level off before eventually reaching a maximum and then, in some cases, beginning to decrease [Fu78 for some examples with Cm*]. This ***saturation effect*** can be attributed to a number of causes;

Resource contention : as the number of processors increases so will requests to access the global resources, e.g. shared data in global memory and dedicated processors. More and more conflicts will occur as the usage of the resources increases. In some cases the bandwidth of the communications net may ultimately be fully utilised and so processors will spend more time waiting to use the net as well as the resources.

Overheads : a parallel algorithm for a multiprocessor system will inevitably require more steps than a serial algorithm, due to the overheads in managing and scheduling the system. For example certain cooperating tasks may require to be periodically synchronised and thus some processors may have to wait while others catch up.

Input/Output : if there are fewer I/O devices than processor modules then processors may become idle while waiting for input data or while waiting for output requests to be serviced. For example for certain applications on the Illiac IV I/O functions have been measured to

consume up to 60% of the total processing time, [HLSM82].

The onset of saturation will depend on the particular configuration of the multiprocessor system and the actual task it is performing, e.g. whether the process is compute bound or I/O bound. For example, for a compute bound process, e.g. matrix multiplication, the number of computations is larger than the number of I/O operations and so will have improved performance on certain multiprocessor systems than on others.

The advantages of multiprocessor systems over single processor systems cannot simply be measured in terms of performance improvement alone, although this is probably the most important and attractive measure for computer users. However another important factor is cost and even here multiprocessor systems can bring improvements. Traditionally Grosch's law [FK83] suggested that processor performance was proportional to the square of the cost and thus adding extra processors was not an economical means of improving performance. However with the advent of cheap, VLSI microprocessors this is no longer the case and the Cosmic Cube [Se85] is a prime example of this. The Cosmic Cube consists of 64 identical computer modules connected as a hyper-cube, with each module containing a 16-bit Intel 8086 microprocessor and 8087 floating-point coprocessor plus 136K bytes of memory. The system is reported to have one tenth of the processing power of a Cray-1, but with a total manufacturing cost of $80,000 it has only one hundredth of the cost, [FO84].

Another of the advantages of a multiprocessor system lies in their potential for improved reliability due to redundancy. In a redundant system all, or most of, the system elements are duplicated and so in the event of a failure in one of the elements the system can still operate, although perhaps with a reduced performance. Tightly coupled multiprocessor systems are inherently more reliable than moderately or loosely coupled systems, since in such systems there are duplicates of

all processor, memory and I/O modules. Moderately coupled systems where the system elements are not homogeneous in their capabilities are obviously less fault tolerant. However the system software must support fault tolerance as well as the hardware. The system software and hardware must combine to detect any errors as soon after their occurence as possible and the spread of faulty data must then be contained. Diagnostic routines will then determine the extent of the problem and if necessary isolate the faulty module. The system software will then reallocate tasks to the remaining properly functioning modules. The ability to isolate a module while still retaining overall system functionality is also a factor in serviceabilty since this allows the system to operate while repairs are being made to a defective module.

However MIMD systems, and parallel systems in general, have disadvantages as well as advantages. The main problems lie with the software, in the areas of operating systems design, languages and compilers [Pr79, Hw87].

The communications net plays a fundamental role in determining the overall capabilities of an MIMD system. The total useful utilisation of the net is partly determined by the nature of the processor modules and global resources interfaced to it. That is if processors modules are equipped with sufficient local memory to store program code and local data then accesses via the net can be reduced. A number of different methods for interconnecting system elements have been suggested and implemented and the next section is devoted to a brief discussion of these.

1.4 Interconnection Methods

There are a number of important factors to be considered when discussing the merits of any communications net for MIMD systems. Bandwidth, reliability, modularity and cost are some of these factors. These in

turn depend on other considerations, e.g. the number of connections required for each module (be it processor, memory or I/O) interfaced to the net, whether control is centralised or decentralised and whether transfers between modules are direct or indirect (i.e. do some transfers require the cooperation of other modules).

Three of the main structures used for interconnecting processors and global resources are the crossbar switch, multiport resources and the time shared or common bus.

1.4.1 The Crossbar Switch

A crossbar switch provides complete direct connectivity between processors and resources. Essentially there is a separate path from each resource which can be switched to any of the processors. There is therefore never any contention for a communication path but there may still be contention over an individual resource. Thus if there are m resources and n processors then the crossbar requires m x n switches

The important feature of the crossbar switch is that is supports multiple concurrent transfers to all the resources. Only one processor can access a resource at one time, but the switch allows a total of min(m,n) accesses in parallel, if all processors are accessing different resources. Each individual switch must have hardware capable of resolving multiple simultaneous requests to access the same resource, as well as being able to switch the parallel transmission path.

System fault tolerance can be severely compromised by a fault in one of the switches, possibly rendering a processor, resource or both totally isolated. If several switches are integrated on a single chip then fault modes could be even worse. However redundancy within the switch can go a long way to overcoming these problems.

The crossbar switch system has the potential for very high transfer rates. However the complexity of the switches and the numbers required means that the switch as a whole becomes the dominating factor in the

cost of the overall system. The Carnegie Mellon C.mmp system successfully used a crossbar switch to interconnect 16 processors to 16 memory modules, [HB87].

1.4.2 Multiport Resources

With this organisation the switching and arbitration control which is distributed in the crossbar switch matrix is placed at the interfaces of the resources. Thus each processor has access via its own bus to all the memory and I/O modules. Contention for access to a single resource can still occur and must be resolved essentially by the resource itself. Cost considerations again make multiporting unsuitable when connecting many processors and resources.

1.4.3 Time Shared Buses

This is the simplest method of interconnecting the processors and resources of an MIMD system. The processors have direct access via the bus to each of the resources. Transfers can be controlled totally by the bus interfaces of the processors and resources and hence the bus is often totally passive and thus extremely simple. However with a single bus there can be no concurrency in transfers since only one access to one resource can take place at a time. As a result of this there must be some means of arbitration between competing requests to use the bus. This will be performed in hardware to reduce delays and can arbitrate requests on either a fixed or dynamic priority scheme.

The total bandwidth of the bus is determined by the transfer rate of the processors and the time taken to resolve competing requests. However it is quite feasible that in order to increase the total bandwidth of a large system that it be divided into clusters of processors and resources with each cluster having its own shared bus. Clusters themselves can then be connected via intercluster buses. This is the method used in the Carnegie-Mellon Cm* multiprocessor [Fu78] and

on Fastbus (IEEE P960) where clusters are called segments [Gu84,FAST83]. Each processor can still access each resource in the system, although not necessarily in the same amount of time, and the presence of multiple buses allows accesses within clusters to be performed concurrently thus increasing the total system bandwidth.

Alternatively system bandwidth can be increased by incorporating multiple, dedicated, parallel buses. Each processor would have a dedicated interface to each of the buses, with each bus being interfaced only to a certain type of resource, e.g. a bus dedicated to I/O devices and another to global memory. This method is better suited to systems where the communications load is reasonably well balanced between the different types of resources, otherwise one bus could become the system bottleneck long before any of the others.

As has been said any processor which wants to use the bus must first receive permission in order to avoid a conflict. There are a number of mechanisms for resolving the bus request/arbitration problem. One solution is to have individual request and bus grant signals from each potential ***bus master*** to the ***arbiter***. Thus each master has his own private two-way connection to the arbiter. However this has the disadvantage of requiring two lines on the bus for each potential bus master. It does have the advantage though of speed, simplicity and great flexibility in that it allows the arbiter to use any method in allocating priority to multiple requests.

Another solution is the use of ***daisy chaining***. This method assigns a unique static priority to the requesting devices which is dependent on their physical position relative to the bus arbiter. With this method all devices request the bus from the arbiter via a common (wire-OR) ***bus request line*** and bus ownership is signalled by a ***bus busy line***. When a request is signalled to the arbiter it issues a ***bus grant signal*** down the bus grant daisy chain, as long as the bus is not currently being used. Each requester has two separate lines for the bus grant; a bus

grant input and a bus grant output. When the arbiter issues the bus grant it is passed on to the first module on the daisy chain. If that module is not currently requesting the bus then it will propagate the grant on to the next module on the chain, via its bus grant out line. The first module which receives the bus grant and which is actively requesting the bus will block the propagation of the bus grant down the daisy chain. This module will then assert the bus busy signal and negate its bus request. When the arbiter then sees the bus busy line being asserted it will rescind the bus grant signal.

The new bus master can hold the bus and perform as many bus accesses as it wishes until it decides to negate the bus busy signal. VME bus (or IEEE P1014) uses this type of bus arbitration and allows the current bus master two options on when to release the bus, [Fi85, VME82]. The first is ***release-when-done (RWD)*** which allows the current bus master to keep the bus only to perform a single or block transfer and then to release it. This option is useful where multiple masters require approximately equal bus usage and where transfers are mostly done on a cycle by cycle basis. The second option, ***release-on-request (ROR)***, allows the master to hold the bus as long as it wishes even if it is not actually using the bus. However the current master must release the bus when a bus request is issued by another master. This latter option is most useful in situations where the majority of masters have low bus usage and where the bus transfer rate of a few masters must be maximised. Giving these masters the ability to hold on to the bus so that they do not need to re-arbitrate for every usage will obviously increases their throughput.

When the bus master finishes with the bus, which it signals by negating the bus busy, the arbiter must recommence the arbitration process if there are outstanding requests to use the bus. The arbiter does this by sending the bus grant down the daisy chain again. It can thus be seen that the nearer a requester is to the arbiter on the bus

grant daisy chain then the more likely it is to receive the bus grant signal first and thus the higher its priority in the arbitration process.

Other bus arbitration techniques are possible, such as dividing the bus bandwidth into fixed length time slots that are sequentially offered to each master in rotation. However all the arbitration mechanisms mentioned so far require a centralised arbitration controller. This obviously reduces fault-tolerance since a failure in such a critical component would cause the whole system to fail, unless there was a redundant controller which could be switched in.

However a system of arbitration has recently been introduced on buses such as Fastbus and Futurebus which uses ***distributed arbitration control***, [Gu84, Ta84]. In such systems there are no critical centralised components required for the arbitration but instead each potential bus master has all that is necessary to determine whether it can or cannot assume control of the bus.

With distributed control each potential master is assigned a unique n-bit arbitration number. The bus contains n lines to which the requesters apply their number, via open collector drivers, at the start of an arbitration cycle. Each requester then monitors the lines and if it sees a logic 1 (the lower voltage level) on a line to which it is driving a logic 0 then it ceases to apply all bits of lower significance. After a delay to allow the bus to settle down the bus lines will carry the highest arbitration number among the competitors. The requester which recognises that the number remaining on the bus is its own then knows that it has gained control of the bus.

However this scheme would impose a disadvantage on requesters with low arbitration numbers unless an additional fairness constraint is imposed. On Futurebus (IEEE P896) the fairness constraint means that once a module has finished with the bus it cannot request the bus again until there are no other requests to be serviced, [Ta84]. However some

modules by their nature may have more urgent needs for the bus. Futurebus takes account of this and allows these ***priority modules*** to request the bus whenever they want. Such modules will also have the most significant bit of there arbitration number equal 1, while ***fairness modules*** will have this bit equal 0, giving priority modules an additional advantage in gaining the bus.

With distributed control the current bus master is responsible for initiating the next bus arbitration procedure. It can do this even before it has finished its bus usage, thus allowing the arbitration time to be pipelined with bus transfers. Arbitration time can thus be lost and need not therefore impose a limit on bus bandwidth. The winner of the new arbitration contest must then monitor the bus to wait for the current bus master to finish before it assume bus control.

No central clocks or control circuits are required for Futurebus. Instead 3 dedicated, wire-OR, control lines ensure that all operations concerned with the transfer of the bus are synchronised. Arbitration is thus a completely decentralised operation. Fault tolerance can be additionally enhanced by having a parity bit on the bus for the arbitration number. All potential masters can then check that the state of this parity bit is correct before a new bus master takes control. As a possible additional check at the end of the arbitration contest all losers can test that their arbitration number is less than the number on the bus. Any errors that are found will prevent the hand over of the bus and the current master then restarts the process.

In general, bus systems can be highly modular, allowing an almost unlimited number of processors to be attached, e.g. as with Fastbus. Even simple, more general purpose single bus systems are modular, although usually only up to some upper limit. This upper limit may be determined by the physical limit of the number of slots on a bus backplane. Or it may be determined by the total bus bandwidth available, which itself it technology dependent. Most buses require no alterations

or additions in order to add other processors or resources. In fact new processors, while they must obviously still conform to the bus arbitration and transfer protocols, can make use of faster interfaces and thus achieve higher transfer rates, if the transfer protocol is asynchronous.

The bus itself can be totally passive allowing bus systems to be comparatively cheap and simple. The only dedicated bus hardware lies in the processor and resource interfaces and any controllers that may be required. Additionally all bus transfers are direct thus removing the need for cooperation amongst other processors. Global broadcast transfers are also possible, where one processor sends data to all or some of the processors and resources.

Bus systems have become very popular in modern computing systems, mainly as a result of their simplicity and flexibility. With the introduction of decentralised control they can also be highly reliable. Their main disadvantage has always been their speed but with use of new technology including the development of special bus driver circuits they can be very fast, e.g. Fastbus claims 30 MHz transfer rates giving 120 Mbytes/sec capability.

1.5 Conclusion

Digital hardware technology is still advancing at much the same rate as it has over the last 25 years. Research into x-ray and e-beam lithography as well as improved processing techniques have achieved sub-micron features to the extent that IBM have announced that they have chips "ready for production" with features smaller then 0.5 microns [Electronic Times, May 1988]. However the improvement in speed that reduced feature size brings serves to highlight other problems such as suitable inter-connection techniques and packaging technology.

Eventually fundamental limits in mos and silicon bipolar technology

will be reached, [SM84]. However other currently more rare technologies still have a lot of scope for development. GaAs devices, for example, are five times faster then ECL devices and have other advantages too, but difficulties in lower density and lower yield still must be overcome. GaAs technology has advanced to an extent that a number of discrete functions are now commercially available (the Cray 3 uses mostly GaAs, [Hw87]). However it may be that some of the phenomena which impose size limitations on conventional semiconductor devices could be exploited to produce a new generation of much more efficient devices, in the form of ***the quantumn effect devices*** [Bat88].

A radically new technology is emerging in the form of ***optical computing devices*** using photons instead of electrons. Optical gates and processing elements have been built in a number of labs and a few computers have been proposed, [Wi87].

The exploitation of concurrency also continues to grow, with an increasing number of commercially available parallel and multiprocessor systems. New architectures continue to appear, with a key area of current research being ***neural networks*** [Wi87]. These attempt to model the parallel operation of neurons in the brain with massively parallel collections of relatively simple processors. Applications include Artificial Intelligence and image recognition. Machines have been built, with up to 64000 such cells, and have produced encouraging results [Hi84, RT88].

The remainder of this thesis is a description of a particular application of parallel processing techniques in the field of nuclear physics research. Using current microprocessor technology and applying some of the techniques just discussed a high performance special purpose parallel processor has been built. As such it is a prime example of the reduced size and cost as well the increased performance and flexibility that is now available utilising VLSI technology and parallel processing techniques.

CHAPTER 2

The Shell Model Processor System

2.0 Introduction

Special purpose, or dedicated, processor systems are becoming increasingly prevalent in scientific research due to the ease with which such systems can now be put together using VLSI components. Areas such as aerodynamics, fluid dynamics, Monte Carlo simulations and image processing can require a very high arithmetic processing bandwidth as well as an equally high data transfer bandwidth. General purpose computer systems will not usually achieve their maximum efficiency when applied to such problems. However dedicated machines can have a much higher performance than general purpose computers since their architecture can be optimised to reflect the structure of the problem, so that generality is traded for performance, [PRT85].

When designing a dedicated system the form of the calculation and the architecture of the machine must be as closely matched as possible. For example the application of parallel processing within the architecture can be optimised to mirror any parallelism within the computation. Thus when designing the machine sufficient processing elements should be included to be able to handle the computational load. Equally important is an efficient means of interconnecting the processing elements to each other as well as to the data storage devices so that the necessary data can be moved and processed as required. However full system optimisation will not only influence the architecture of the machine but also the algorithm and form of the

computation.

One example of a machine dedicated to theoretical physics calculations is the previously mentioned Cosmic Cube (Section 1.3.4). The main motivation for the system was Monte Carlo studies of lattice gauge theories. However the message-passing architecture of the system is flexible enough to be applied to the whole class of problems involving multidimensional arrays of interrelated data, such as are found in statistical mechanics and field theory. The Cube has been successfully programmed for a number of different applications with some performing up to 10 times faster than a VAX 11/780 [Se85], thus demonstrating the performance which can be obtained from well designed special purpose processor systems.

The field of nuclear physics theory is another area of research where computationally intensive problems arise. In particular the calculation of the nuclear energy levels which arise out of the theory of the nuclear shell model is of much interest. In the following sections of this chapter we will discuss the nuclear shell model problem and introduce the *Shell Model Processor (SMP)* system. The SMP is a high performance, parallel processor system developed at the Department of Physics at Glasgow University for the purpose of performing such nuclear structure calculations [MBMW85, MMB87].

2.1 The Nuclear Shell Model

It is well known that the nucleus exhibits a behaviour with respect to "magic numbers" of nucleons that is similar to that of atoms which have closed electron shells. For example the rapid change in nucleon binding energy at the nuclear magic numbers is similar to the change in electron separation energy in the atom. It therefore seemed logical for the early nuclear theorists to attempt to develop a shell model of the nucleus based on the quantum-mechanical procedures which had been so

successfully used to develop the atomic shell model. However the early attempts at predicting closed shells through the operation of the Pauli exclusion principle were only able to produce the first three empirically observed nuclear magic numbers. It was not until the later addition of spin-orbit coupling to the theory that the full list of magic numbers was produced.

Although superficially similar the atomic and nuclear systems are physically very different. Electron motion in the atom is governed mainly by the Coulomb force between individual electrons and the central nucleus. The force between individual electrons produces only a small perturbation from this main effect. However the essence of the nuclear shell model is that each nucleon moves under the combined influence of all the other nucleons. The major assumption is that the total effect of the other nucleons can be represented by a potential well having a large negative value at the centre of the nucleus and rising to zero at the surface. Various shapes for the potential have been suggested, ranging from the simple rectangular well, through the three-dimensional harmonic oscillator to the Woods-Saxon potential.

In practice however the single particle spherically symmetrical potential is a simplification since there is evidence of a pairing or two-body interaction within the nucleus [ER74]. The two-body interaction represents a departure from the average single particle potential and arises when a nucleon is close to another nucleon with which it can interact uninhibited by the Pauli exclusion principle. For example two nucleons with different values of m_j collide and after the collision enter states such that the total m_j is unchanged, thus conserving angular momentum. The nuclear force therefore has a two-body nature and the Hamiltonian thus takes the form;

$$H = \frac{-\hbar^2 \nabla^2}{2m} + \sum_{i<j} V(i,j) \qquad (2.1)$$

2.2 The Slater Determinant Representation

In attempting to determine the nuclear energy levels it is usually assumed that only one major shell is actively involved. The problem is therefore to set up the Hamiltonian matrix and then to diagonalise it to obtain the eigenvalues. The eigenvectors are also required in order to calculate the transition rates and expectation values for various measurable quantities. Traditionally the basis states were specified using group theory and were coupled to good J and T quantum numbers. However the need to handle the angular momentum algebra computationally greatly inhibited progress.

It was for this reason that the nuclear theorists at Glasgow University gave up the angular momentum coupled representations and instead used uncoupled antisymmetric product wave functions, i.e. ***Slater determinants***, and an occupation number representation, [Wh72, MBMW85]. A Slater determinant is given by

$$\Phi_{a_1 a_2 \ldots a_n}(\underline{r}_1 \ldots \underline{r}_n) = \frac{1}{(n!)^{1/2}} \begin{vmatrix} \phi_{a_1}(\underline{r}_1) & \ldots & \phi_{a_1}(\underline{r}_n) \\ \vdots & & \vdots \\ \phi_{a_n}(\underline{r}_1) & \ldots & \phi_{a_n}(\underline{r}_n) \end{vmatrix}$$

where $\phi_{a_i}(\underline{r}_j)$ is the wave function for the jth particle in the ith state, for some arbitrary ordering. A Slater determinant can then be written in the occupation number formalism using the creation and annihilation operators, a^+ and a respectively. A typical determinant then becomes

$$a^+_A a^+_B \ldots a^+_N |0\rangle \qquad (2.2)$$

where a^+_i is the creation operator for orbital i, and A, B, etc are the indices of the occupied orbitals with A < B < .. etc. Such states have definite values for the total z-component of angular momentum and total z-component of isospin but no definite total angular momentum or

isospin. This representation is known as the ***m-scheme***, [WWCM77].

Under the m-scheme it is appropriate to use an occupation number representation for the Hamiltonian, so that;

$$H = \sum_{ik} H^{(1)}_{ik}\, a^{+}_{i}\, a_{k} + 1/4 \sum_{ijkl} H^{(2)}_{ijkl}\, a^{+}_{i}\, a^{+}_{j}\, a_{l}\, a_{k} \qquad (2.3)$$

where $H^{(1)}_{ik}$ and $H^{(2)}_{ijkl}$ are the one and two-body Hamiltonian matrix elements respectively. A simplification can be achieved by combining the two terms, so that the Hamiltonian can be treated as a purely two-body operator, [WWCM77], thus;

$$H = \sum_{\substack{ijkl \\ i<j \\ k<l}} \left[\frac{1}{n-1} H^{(1)}_{ik}\, \delta_{jl} + H^{(2)}_{ijkl} \right] a^{+}_{i}\, a^{+}_{j}\, a_{l}\, a_{k} \qquad (2.4)$$

so that the Hamiltonian is now explicitly dependent on n, the number of nucleons. Thus the Hamiltonian can be written as;

$$H = \sum_{\substack{ijkl \\ i<j \\ k<l}} \hat{H}_{ijkl}\, a^{+}_{i}\, a^{+}_{j}\, a_{l}\, a_{k} \qquad (2.5)$$

To diagonalise the Hamiltonian, H, it is necessary to have a form for the actual matrix elements. These are given as follows, [Mac83]:
Let H be the two-body Hamiltonian as given in 2.5 and L be a basis list of Slater determinants for a system of n nucleons. Let $|i\rangle$ and $|f\rangle$ be two states, both members of L, such that;

$$|i\rangle = |\alpha_1 \ldots \alpha_n\rangle$$
$$|f\rangle = |\beta_1 \ldots \beta_n\rangle$$

where $\alpha_1, \ldots \alpha_n, \beta_1, \ldots \beta_n$ are the indices of the occupied single particle orbitals. The matrix elements are then;

1/ If $|i\rangle = |f\rangle$, i.e. $\alpha_i = \beta_i$ for i = 1 to n, then

$$\langle f|\, H\, |i\rangle = \sum_{wxyz} \langle f|\, \hat{H}_{wxyz}\, a^{+}_{w}\, a^{+}_{x}\, a_{z}\, a_{y}\, |i\rangle$$

$$= \sum_{wxyz} \langle \alpha_1 \ldots \alpha_n|\, \hat{H}_{wxyz}\, a^{+}_{w}\, a^{+}_{x}\, a_{z}\, a_{y}\, |\beta_1 \ldots \beta_n\rangle$$

$$= \sum_{1=i<j}^{n} \hat{H}_{\alpha_i \alpha_j \alpha_i \alpha_j} \qquad (2.6)$$

2/ If $\{\alpha_1 \ldots \alpha_n\} + \{\beta_1 \ldots \beta_n\} = \{\alpha_i, \beta_j\}$, i.e. there is only one different occupied orbital between $|i\rangle$ and $|f\rangle$, then

$$\langle f| H |i\rangle = \sum_{\substack{k=1 \\ k\neq i,j}}^{n} \hat{H}_{\alpha_k \beta_j \alpha_k \alpha_i} (-1)^p \qquad (2.7a)$$

where

$$p = \sum_{s=\alpha_k+1}^{\beta_j-1} n_s + \sum_{s=\alpha_k+1}^{\alpha_i-1} n_s \qquad (2.7b)$$

where $n_s = 0$ if the orbital with index s is empty,

$= 1$ if the orbital with index s is occupied,

for the Slater determinant $a_{\alpha_k}^{\dagger} a_{\alpha_i} |i\rangle$.

3/ If $\{\alpha_1 \ldots \alpha_n\} + \{\beta_1 \ldots \beta_n\} = \{\alpha_i, \alpha_j, \beta_k, \beta_l\}$, i.e. there are two different occupied orbitals between $|i\rangle$ and $|f\rangle$, then

$$\langle f| H |i\rangle = \hat{H}_{\beta_k \beta_l \alpha_j \alpha_i} (-1)^p \qquad (2.8a)$$

where

$$p = \sum_{s=\alpha_i+1}^{\alpha_j-1} n_s + \sum_{s=\beta_k+1}^{\beta_l-1} n_s \qquad (2.8b)$$

for the Slater determinant $a_{\alpha_j}^{\dagger} a_{\alpha_i} |i\rangle$.

4/ If there are more than two occupied orbitals different between $|i\rangle$ and $|f\rangle$ then

$$\langle f| H |i\rangle = 0 \qquad (2.9)$$

(N.B. that for any two sets A and B, $A+B = (A-B) \cup (B-A)$)

Thus making use of the occupation number representation there is now a simple mapping by which SDs can be efficiently represented and manipulated within a computer. That is to assign each possible single

particle orbit to a different bit position in a computer word. An occupied orbital is then represented by 1 in the relevant bit of the word, while an unoccupied orbital is represented by a 0. For example in the 2s-1d shell there are 24 single particle orbits, thus requiring only a 24-bit word to represent each SD. Thus using this method SDs and the form of the Hamiltonian itself can be generated by using bit manipulation and logic operations.

The basis space for a nucleus in shell model calculations is potentially very large. For example in a calculation for ^{28}Si ($m = 0$) with 12 active nucleons in the 2s-1d shell, there are 93710 states (the maximum for the sd shell) giving almost 10^{10} elements in the matrix. However only 20 to 30 of the eigenstates produced are actually compared with experimentally determined values. Therefore a diagonalisation method which produces all the eigenstates will generate mostly unwanted information. Central to the method developed at Glasgow University, along with the m-scheme representation, is the use of the ***Lanczos algorithm*** for the iterative tri-diagonalisation of the nuclear Hamiltonian. Using this algorithm only as many of the lower eigenstates as are wanted are produced with the minimum of additional unwanted information.

2.3 The Lanczos Method

The task of determining the eigenvalues and eigenvectors for a real symmetric matrix is generally performed using the Householder tri-diagonalisation method. However for shell-model work its major drawback is that it requires the full tri-diagonalisation process to be completed before any of the eigenvalues can be obtained. The Lanczos method [FM77] is, at least in theory, almost ideal for finding the extreme eigenvalues of a large sparse symmetric matrix [Pa72]. The two methods are equivalent and will produce the same results. However the Lanczos method

is an iterative scheme which will produce the upper left-hand k x k submatrix after only k iterations. The eigenvalues of this k x k matrix converge rapidly to very accurate approximations of the extreme eigenvalues of the full matrix as k increases, [WWCM77]. This remains true even when k is much less than the dimension of the matrix. Therefore in shell-model work the lowest energy levels, which are the most useful, can be obtained after only 50 to 100 iterations, regardless of the size of the basis space.

The Lanczos method works as follows; let A be a real, symmetric, n x n matrix and v_1 an arbitrary, n x 1 vector, such that $v_1^+ v_1 = 1$ (where the + denotes the transpose). New vectors are then generated by iteration;

$$
\begin{aligned}
Av_1 &= \alpha_1 v_1 + \beta_1 v_2 \\
Av_2 &= \beta_1 v_1 + \alpha_2 v_2 + \beta_2 v_3 \\
Av_3 &= \beta_2 v_2 + \alpha_3 v_3 + \beta_3 v_4 \\
&\;\;\vdots \\
Av_n &= \beta_{n-1} v_{n-1} + \alpha_n v_n
\end{aligned}
$$

such that the v_i are all orthogonal with respect to each other and are all normalised. The process terminates automatically after n iterations since there can only be n mutually orthogonal vectors for the space and therefore v_{n+1} must be 0.

The Lanczos vectors v_1 to v_n then form an orthonormal basis in which A takes the tri-diagonal form

$$
\begin{bmatrix}
\alpha_1 & \beta_1 & & & \\
\beta_1 & \alpha_2 & \beta_2 & & \\
 & \beta_2 & \alpha_3 & \beta_3 & \\
 & & \ddots & \ddots & \ddots \\
 & & & \beta_{n-1} & \alpha_n
\end{bmatrix}
$$

The coefficients are determined as follows;

$$
\begin{aligned}
\alpha_i &= v_i^+ A v_i \\
\beta_{i-1} &= v_{i-1}^+ A v_i
\end{aligned}
$$

$$w_{i+1} = \beta_i v_{i+1} = Av_i - \beta_{i-1} v_{i-1} - \alpha_i v_i$$

$$\beta_i = (w^+_{i+1} w_{i+1})^{1/2}$$

If there are degenerate eigenvalues then the Lanczos method will terminate in less than n iterations and only one eigenvector and eigenvalue from the degenerate set will be obtained, [WWCM77]. However degenerate eigenvalues rarely arise in shell model work, but the problem can be overcome by using a new initial vector.

Unfortunately in practice the Lanczos method is not as ideal as at first it seems. This is due to arithmetic processing inaccuracies which lead to a loss of orthogonality in the Lanczos vectors and which stops the process from actually terminating. The remedy is to re-orthogonalise the current vector, v_i, with all the previous ones, as follows;

$$x_i = w_i - \sum_{j=1}^{i-1} v^+_j w_i v_j \qquad (2.10a)$$

$$v_i = \frac{x_i}{(x^+_i x_i)^{1/2}} \qquad (2.10b)$$

It is this which makes the Lanczos method less attractive than at first appears. Indeed if the full matrix were to be diagonalised it would be much less efficient than the Householder method. However if less than n/4 iterations are sufficient, which is exactly the case with shell model work, then the Lanczos method has the computational advantage over the Householder method in terms of storage requirements and speed.

2.4 SMP System Introduction

We have so far described the nature and extent of the nuclear physics problem and a method for its solution. However the original Glasgow Program for determining the nuclear energy eigenvalues has a number of limitations. These restrictions are a result of the type of computers that the Glasgow Program is implemented on, which because of the

magnitude of the shell-model calculation must be very large, high performance mainframe installations. Access to these computers is both limited and expensive, thereby reducing the number and scope of the calculations that can be performed.

The number of single particle orbitals in any calculation is limited by the type of computer used, being equal to the number of bits in the computer word, allowing up to 59 orbitals on a CDC 7600 machine but only up to 32 on an IBM 370 series computer. It is possible to store each Slater determinant in more than one word, as has sometimes been done, but this reduces the efficiency of the process and therefore still imposes a limitation.

The amount of primary memory available on a mainframe also further acts to limit the scope of the calculations since the Glasgow Program requires that the complete Slater Determinant basis list be stored during runtime [WWCM77]. The amount of space required to store this list increases rapidly with any increase in the number of active orbitals and in a 128 orbital system would require too much space even on today's mainframes.

Out of a desire therefore to overcome these limitations and so to increase the number and scope of shell-model calculations which the Glasgow Nuclear Structure Group could perform, the following aims were drawn up:

1/ That a dedicated computer system should be designed and built in order to carry out nuclear structure calculations.

2/ The initial computer should be a prototype system, able to deal with up to 32 single particle states.

3/ The computer should be totally accessible to the nuclear physicist and have a low construction and running cost.

4/ The performance should be comparable to that of an IBM 360/195.

5/ That this prototype system should act as a testbed for a later machine with four times the capability, i.e. it should be able to

deal with up to 128 single particle states.

Having gone so far as to decide to design a dedicated shell-model processor, the question must be asked, what method should it use and what should its nature be so that it is not restricted by the same limitations as the current mainframes? i.e. should it be a simple one processor SISD machine, or perhaps a SIMD array processor. An answer to this question lies in the nature of the shell-model calculation, examination of which shows that it divides into two logical stages;

1/ to generate the basis list of Slater determinants for the nucleus and then to perform the annihilation and creation operations on the list to determine the positions of the non-zero matrix elements within the Hamiltonian matrix,

2/ to multiply the Hamiltonian matrix by the Lanczos vector and so to accumulate a resultant vector.

This second stage further subdivides into a large number of independent, non-identical tasks. Namely the determination of the magnitude and sign of the matrix element from the annihilation and creation operators (found in the first stage) and then its multiplication by the appropriate Lanczos vector element and accumulation into a resultant vector element. These tasks are non-identical not just in the fact that they operate on different data, but also in that they will follow different paths to determine the Hamiltonian entry according to one of equations 2.6-2.8. Since the tasks are independent it is possible that any number of them could be carried out in parallel. Normally matrix multiplication is well suited to array processor architectures. However since in this instance the matrix is irregularly sparse this is not the case. In fact this second stage of the calculation is ideally suited to a multi-processor configuration.

It is only the first stage of the calculation that actually manipulates the SDs and so only it need have the capability of handling 32 bit words (or 128 bit words in the expanded system). It is therefore

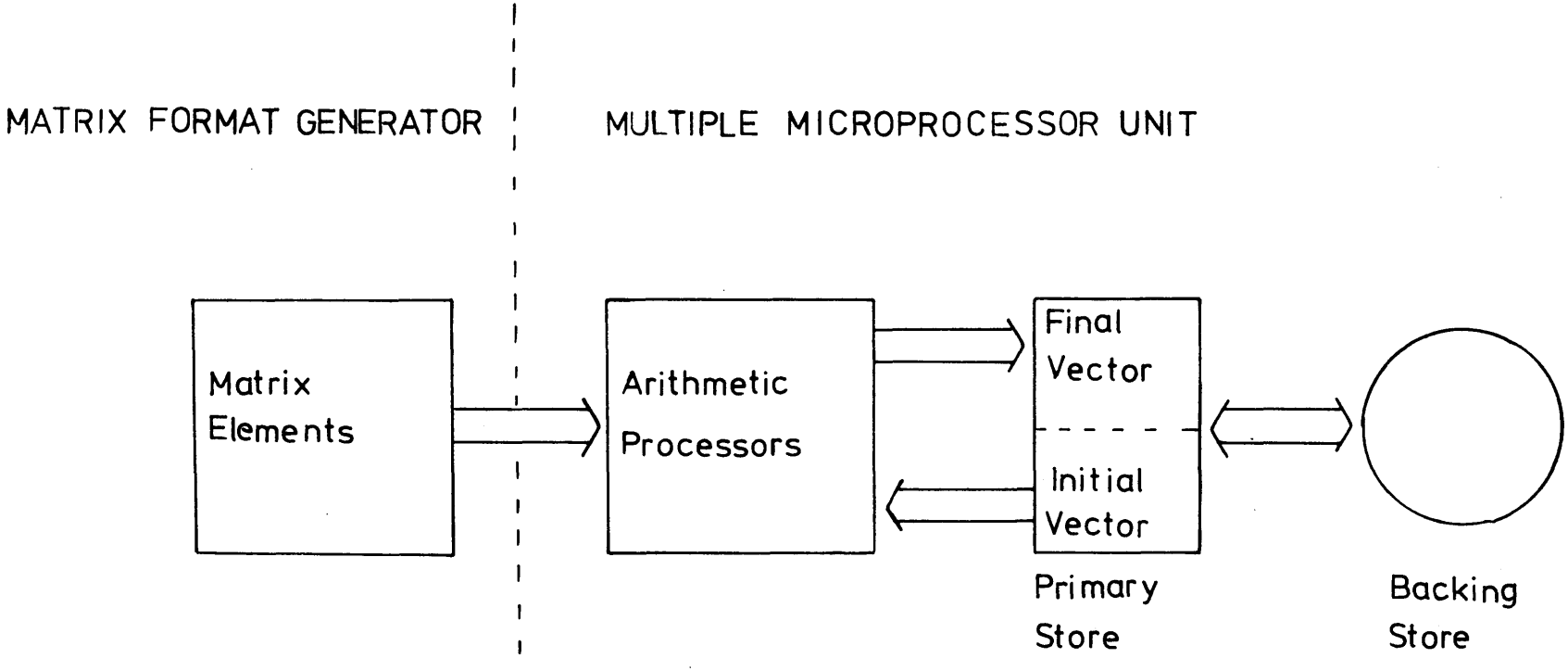


Figure 2.1 SMP LOGICAL STRUCTURE

possible that this stage be a dedicated hard-wired unit.

These aims and objectives were drawn up a number of years ago by Dr. A.M. MacLeod and Prof. R.R. Whitehead in the Dept. of Natural Philosophy at Glasgow University. The prototype Shell-Model Processor is now almost complete with only one component of the system still to be added. The SMP system is operational without this element, allowing one full iteration on a basis size of up to 13,000 elements. A number of test iterations have been successfully run, thus proving the integrity of the system and fulfilling the original aims. The initial feasibility studies and some design and prototyping work was carried out by Dr. L.M. MacKenzie as the work for his Ph.D. My work has been largely concerned with the later design and testing of both hardware and software in order to integrate and commission the system as a whole. What follows therefore is mainly a description and discussion of the SMP system both in terms of its hardware and software.

2.5 A Global View

As has already been said the shell-model calculation divides into two logical parts, with this division being reflected in the two major functional sub-systems of the SMP (figure 2.1). The ***Matrix Format Generator (MFG)*** has the responsibility of determining the position of non-zero elements within the Hamiltonian, i.e. it must determine the row and column index for each non-zero element as well as its creation and annihilation operators. The second sub-system, the ***Multiple Microprocessor Unit (MMPU)***, then uses the information determined by the MFG in order to identify the magnitude and sign of the Hamiltonian matrix elements and then perform the arithmetic to produce a new vector from the current Lanczos vector. The MMPU must hold all the previous Lanczos vectors in order to be able to perform the re-orthogonalisation which is necessary after each iteration (section 2.3). The MMPU is a

modular, moderately coupled MIMD system based on autonomous processing elements and is thus able to process a number of matrix elements in parallel.

The two SMP sub-systems can themselves be further subdivided into a number of functionally separate units (figure 2.2). A ***communications subnet*** is also be defined so that the two main sub-systems, and the units within them, can communicate with each other. We will now describe the detail present within the two subsystems and the subnet.

2.5.1 The Matrix Format Generator

1/ The Primary Generator: In the SMP system, the SD basis list does not need to be stored, thus overcoming the needs to have vast amounts of primary memory for its storage. Instead the MFG generates the basis list during each iteration. This task is carried out by the ***Primary Generator (PG)***. The PG is basically a single board computer based on the Motorola MC68000 (8 MHz) microprocessor and 128K bytes of local dynamic RAM. Its task of generating the basis list is performed using several data tables built prior to runtime and stored in local RAM. The PG also acts as a supervisor and controller to the rest of the MFG, ensuring its proper initialisation and performing runtime maintenance and control.

2/ The Secondary Generator: Once a state within the basis list has been produced by the PG we have, with the state, identified a column within the Hamiltonian matrix. This state, called a ***prime state*** $|e_n\rangle$, is then passed to the ***Secondary Generator (SG)*** which in response has the task of generating sections of the basis list, i.e. sections of the matrix column, where non-zero elements may exist. The states so produced, called ***secondary states*** $|e_m\rangle$, form pairs of states with the prime state $(|e_n\rangle, |e_m\rangle)$ and each pair must then be tested to determine whether it defines a non-zero matrix element.

The SG has effectively to regenerate parts of the basis list for

each member in the basis list and this obviously has the capacity for being a very large task. To cope with this workload the SG is a dedicated, hard-wired logic module which does not run a control program and is constructed using ***emmitter coupled logic (ECL)***. The SG is at present clocked at 112 MHz and is capable of producing a peak rate of approximately 8.6 million secondary states per second (i.e. one every 13 clock cycles). For ^{27}Al m=5/2, which has a basis list of 64,299 states, the SG must produce approximately 1.666×10^{9} secondary states per iteration, which it can do in 3.28 mins, effectively placing an upper limit on the performance of the SMP as a whole.

3/ <u>The Pair Filter</u>: As a result of the method of operation of the SG (which will be explained later) many of the secondary states it produces will not actually combine with the prime state to produce a non-zero matrix element. The task of filtering out these redundant secondary states is given to the ***Pair Filter (PF)***. For each valid secondary state the PF finds, i.e. one which is two particles or less different from the prime state $|e_n\rangle$, the PF must also generate the indices of the annihilation operators (a_k, a_l) and creation operators (a_i^+, a_j^+) such that $|e_n\rangle = a_k a_l a_i^+ a_j^+ |e_m\rangle$. These operator indices (k,l,i,j) determine the magnitude of a non-zero matrix element within the Hamiltonian matrix and must be passed to the MMPU to be processed. Obviously the performance of the PF must match that of the SG and so the PF is also a dedicated hard-wired module constructed using ECL.

4/ <u>The MFG Buffer</u>: The rate of output from the PF will vary considerably and will only rarely reach the same peak rate as the SG due to the fact that most of the secondary states are filtered out. In order to even out the rate of output of valid secondary states by the PF and to reduce the occurrence of the MFG being held up while it waits for its output to be consumed by the MMPU, a ***first-in-first-out (FIFO)***

buffer stores the PF output.

Each output word, called a ***Task Setup Word (TSW)***, in the MFG Buffer contains the necessary set-up parameters for the MMPU to identify the matrix element magnitude and also which element in the final vector is to be updated. To this end the TSW must contain the index of the secondary state (m), the annihilation and creation operator indices (up to 4 of these), and information regarding which of eqns 2.6-2.8 should be used. When the MMPU is able to receive a new set of parameters to start another job, then the TSW at the top of the buffer is read out thus providing an extra empty space at the bottom of the buffer. It is only when the buffer becomes full that the MFG must halt its operation and remain idle until a new space becomes available. The read/write control circuitry for the buffer is also constructed using ECL.

In order for the MFG to achieve maximum throughput it is designed as a parallel processor, with all 4 of its sub-units completely pipelined with one another. In particular the SG, PF and MFG buffer all have the same major cycle time in which they process a state.

2.5.2 The Multiple Microprocessor Unit

1/ **The Microcomputer Modules** : these are the modules which must read the set-up parameters from the MFG Buffer and perform the matrix times vector arithmetic. When a ***Microcomputer Module (MCM)*** reads a TSW it must determine the magnitude and sign of the matrix element using the information imparted by the annihilation and creation operators and the job type bits. The index m, also included in the TSW, gives the index of the final vector element, V_{fm}, whose new value is to be calculated as follows:

$$V_{fm} = V_{in} \times H_{mn} + V_{fm} \qquad (2.11)$$

The index n of the initial vector element V_{in} is the index of the prime state being considered by the MFG and therefore remains static

for varying lengths of time. For this reason n is not included in the TSW but instead is passed directly to the MCMs by the MFG each time the prime state changes.

The MCMs must therefore have their own native intelligence capable of evaluating one of equations 2.6-2.8 and performing the floating point arithmetic. Their ability to carry out this task is extremely important since it is the speed of the individual MCMs which will determine the performance of the MMPU. To fulfill their purpose the MCMs are therefore high performance single board computers.

2/ Central Memory: as has been said, all the MCMs require access to the initial and final vectors during an iteration. Since the storage space required is large, up to 800K bytes for the biggest sd shell nucleus, it is much more efficient to store these vectors centrally, which is the purpose of ***Central Memory (CM)***. As each MCM starts a new task it will read the required initial and final vector elements from CM and at the end of the task will write the updated final vector element back to CM.

Included in the CM subsystem will be a high capacity backing store which is intended primarily to store the Lanczos vectors. After each iteration the new Lanczos vector will be orthogonalised with respect to all the previous vectors held in the store and then copied into the store itself.

3/ Supervisor Module : it is this module's responsibility to monitor the system during runtime and also to ensure the correct initialisation of all the parts of the SMP system. The ***Supervisor Module (SM)*** also acts as the interface to the outside world, e.g. via terminals, printers, disks, etc.

2.5.3 Communications Subnet

1/ Input Bus: *I-bus* is the dedicated highway between the MFG Buffer and the MCMs, along which the TSWs are read. As such it is fairly simple

single address, uni-directional bus, but must have a high transfer rate in order to keep up with the required flow of TSWs to the MMPU.

2/ Central Memory Access Bus: *CMA-bus* is the means by which the MCMs perform read and write cycles to CM to access the vector elements. As such it is more complicated than I-bus but requires the same performance capabilities.

3/ Communications Bus: *C-bus* is the main system highway for communications between components of the MMPU and the MFG. It is a general purpose multiprocessor bus.

2.5.4 SMP Modes of Operation

Having thus described the tasks of the MFG and MMPU we can now draw attention to an important fact that allows us to almost half the workload of the MFG and also, but to a much lesser extent, reduce the workload of the MMPU. This is simply the fact that the Hamiltonian is symmetric i.e.

$$\langle e_m | H | e_n \rangle = \langle e_n | H | e_m \rangle \qquad \text{for all m and n.}$$

Therefore once the MFG has identified two states $|e_n\rangle$ and $|e_m\rangle$ such that $H_{mn} \neq 0$ the MMPU can then perform two jobs, i.e. instead of the MMPU just evaluating

$$V_{fn} = V_{im} \times H_{mn} + V_{fn} \qquad (2.12a)$$

it can also evaluate

$$V_{fm} = V_{in} \times H_{mn} + V_{fm} \qquad (2.12b)$$

using the same H_{mn}. Thus the MFG need only search half of the matrix for non-zero elements, i.e. in every column it need only search up to the diagonal element, and in turn the MMPU has only half the number of jobs to process.

However for each task the MMPU has to process there is now twice the arithmetic workload and twice the number of vector elements to fetch although there is still only one matrix element value to be determined. Thus if the MFG is operated in this way, called *H-mode* as opposed to

W-mode when the whole matrix is generated, the workload is significantly shifted off the MFG. H-mode is therefore particularly useful in situations where the MFG is the system bottleneck.

2.6 Conclusions

Although the system has been named the Shell-Model Processor it should not be seen as a rigidly dedicated system useful only for nuclear structure calculations, since this is far from the case. For a start this type of calculation, i.e. matrix generation and diagonalisation, is common in many other branches of science. However far more than this the SMP has the flexibility to be applied to many problems which have a degree of parallelism and which could utilise the processing power of the MMPU. The MMPU itself, since it is based on multiple, high-performance single board computers, can be viewed as a general purpose moderately coupled multiprocessor system and is therefore useful in many other types of calculations. Even if the MFG could not be used in these problems, the I-bus is of a sufficiently general nature that it could be used to connect the MMPU to some other input device e.g. a high speed disk or pre-processor.

We have in this chapter given an overview of both the nuclear structure problem and the prototype SMP as a means for its solution. The following chapters will be devoted to a more detailed description and discussion of the system. Particular attention will be given to the MFG, the multiple MCMs and the communications subnet since they are the most important sections of the system in terms of their workload and performance. The details of the Supervisor module will also be given as well as the plans for the Central Memory, this being the only part of the system not yet implemented.

CHAPTER 3

The Matrix Format Generator

3.0 Introduction

The function of the Matrix Format Generator (MFG) has already been described (Sec. 2.5.1) as well as its internal high-level structure. We will now give further details of the MFG, describing the algorithm it uses and its implementation in terms of both hardware and software.

3.1 Basis List Representation and Partitioning

Having chosen to use a Slater Determinant representation for the basis states the most simple and (for manipulation purposes) efficient method of representing them is, as we have said, to have one bit in the computer word representing one single-particle orbital. Thus for the 24 orbitals of the sd shell only 24 bits in a computer word are required, giving 8 spare bits in the current MFG which is a 32-bit machine.

The Shell-Model Processor system further subdivides this 32-bit word such that bits 0-15 (i.e. the least significant 16 bits) are reserved for neutron orbits and bits 16-31 are reserved for proton orbits. Within these two half-words the orbital assignment is completely arbitrary, for example figure 3.1 shows a possible assignment (note that any particular assignment is called an *SD representation*). Thus given the number of protons (Np), the number of neutrons (Nn) and the z component of the total angular momentum (M_j) for the sd shell of the nucleus and an appropriate representation we can generate a list of 32-

Bit number	l	j	m_j	nucleon
31			0	unused
30			0	unused
29	2	5/2	5/2	proton
28	2	5/2	3/2	proton
27	2	3/2	3/2	proton
26	2	3/2	-3/2	proton
25	2	5/2	-3/2	proton
24	2	5/2	-5/2	proton
23			0	unused
22			0	unused
21	2	5/2	1/2	proton
20	2	3/2	1/2	proton
19	0	1/2	1/2	proton
18	0	1/2	-1/2	proton
17	2	3/2	-1/2	proton
16	2	5/2	-1/2	proton
15			0	unused
14			0	unused
13	2	5/2	5/2	neutron
12	2	5/2	3/2	neutron
11	2	3/2	3/2	neutron
10	2	3/2	-3/2	neutron
9	2	5/2	-3/2	neutron
8	2	5/2	-5/2	neutron
7			0	unused
6			0	unused
5	2	5/2	1/2	neutron
4	2	3/2	1/2	neutron
3	0	1/2	1/2	neutron
2	0	1/2	-1/2	neutron
1	2	3/2	-1/2	neutron
0	2	5/2	-1/2	neutron

Figure 3.1 Example SMP Orbital Assignment

bit numbers which represent the Slater Determinant (SD) basis list for the nucleus. These 32-bit numbers are called ***SD-words***.

To make the generation of the basis simpler and so ease the task of the PG and SG we partition up the basis list and define an order on it. It should be noted that from this point on the method used by the SMP system to generate the basis states and Hamiltonian entries starts to differ significantly from the original method of the Glasgow Shell-Model Program [WWCM77].

First the SD word is sub-divided up into 4 8-bit sub-words which we call ***SD-bytes***;

SD-byte 0 comprises bits 31 - 24

SD-byte 1 comprises bits 23 - 16

SD-byte 2 comprises bits 15 - 8

SD-byte 3 comprises bits 7 - 0

SD bytes 0 and 1 are proton bytes and named P1 and P2 respectively while SD bytes 3 and 4 are neutron bytes and named N1 and N2 respectively. For simplicity we define an integral ***M-value***, M_i, for each bit i, such that;

$$\begin{aligned} M_i &= 2m_{ji} && \text{for each used bit,} \\ &= 0 && \text{for an unused bit.} \end{aligned} \qquad i = 0..31 \qquad (3.1)$$

The total M-value for an SD is then defined as;

$$M = \sum_{i=0}^{31} M_i \delta_i \qquad (3.2)$$

where : $\delta_i = 0$ for an unoccupied orbital,

$= 1$ for an occupied orbital.

We also define $n_i(A)$ and $m_i(A)$ where

$n_i(A)$ = total number of occupied orbitals (set bits) in byte i of SD word A,

and $m_i(A)$ = the sum of the individual M-values for the occupied orbitals in byte i of SD word A.

We also denote the basis for a given nuclei with Np protons, Nn neutrons, total M-value M and under representation R, as;

B-R(Np,Nn,M)

A basis list can now be partitioned up into what are defined as ***N-partitions*** and denoted;

$$[\ n(P1)\ |\ n(P2)\ |\ n(N1)\ |\ n(N2)\ ]$$

such that for all SD-words A in the N-partition;

$$n_0(A) = n(P1),\quad n_1(A) = n(P2),\quad \text{etc.} \qquad (3.3)$$

and where $n(P1) + n(P2) = Np$ and $n(N1) + n(N2) = Nn$.

Thus all states in B-R(Np,Nn,M) can be placed in one, and only one, N-partition and so the basis is completely and uniquely subdivided by these partitions.

Each N-partition can now be subdivided by defining an ***M-partition***, denoted;

$$\begin{bmatrix} n(P1) & n(P2) & n(N1) & n(N2) \\ m(P1) & m(P2) & m(N1) & m(N2) \end{bmatrix}$$

such that for all SD-words A in the M-partition;

$$m_0(A) = m(P1) \quad \text{and} \quad n_0(A) = n(P1),\quad \text{etc} \qquad (3.4)$$

and where $m(P1) + m(P2) + m(N1) + m(N2) = M$.

Each state can thus be placed in one and only one M-partition and so the N-partitions are uniquely subdivided.

Using the N and M-partitions an order can now be imposed on the states within any basis B-R(Np,Nn,M). First the N-partitions are ordered;

Let $N_1 = [\ n(P1)\ |\ n(P2)\ |\ n(N1)\ |\ n(N2)\ ]$

and $N_2 = [\ n(P1)"|\ n(P2)"|\ n(N1)"|\ n(N2)"]$

be two arbitrary N-partitions within a basis. Then we define

$$N_1 < N_2 \iff (1)\ \ n(P2) < n(P2)" \quad \text{or}$$
$$(2)\ \ (\ n(P2) = n(P2)"\ )\ \text{and}\ n(N2) < n(N2)" \qquad (3.5)$$

(Note that if $n(P2) = n(P2)"$ then $n(P1) = n(P1)"$).

We can thus say that an N-partition N_1 "is less than" another N-partition N_2 if the above is true for N_1 and N_2.

An order can now be imposed on the M-partitions, such that if M_1 and M_2 are two arbitrary M-partitions within a basis, and if M_1 and M_2 belong to different N-partitions, N_1 and N_2 respectively, then we define

$$M_1 < M_2 \iff N_1 < N_2$$

Np = 3
Nn = 3
m = 0

N-partitions

	[	3	0	3	0	]	---- Initial N-partition
*	[	3	0	2	1	]	
	[	3	0	1	2	]	
	[	3	0	0	3	]	
*	[	2	1	3	0	]	
*	[	2	1	2	1	]	
*	[	2	1	1	2	]	
	[	2	1	0	3	]	
*	[	1	2	3	0	]	
*	[	1	2	2	1	]	
*	[	1	2	1	2	]	
*	[	1	2	0	3	]	
*	[	0	3	3	0	]	
*	[	0	3	2	1	]	
*	[	0	3	1	2	]	
	[	0	3	0	3	]	--- Final N-partition

* denotes N-partition connected to [1 , 2 , 2 , 1]

Figure 3.2 Example N-partitions

Np = 3
Nn = 3
m = 0

M-partitions

[	-5 , -2 , 6 , 1	]	---	Initial M-partition
[	-5 , -2 , 8 , -1	]		
[	-5 , 0 , 6 , -1	]		
[	-5 , 2 , 2 , 1	]		
[	-3 , -2 , 6 , -1	]		
[	-3 , 0 , 2 , 1	]		
[	-3 , 2 , 0 , 1	]		
[	-3 , 2 , 2 , -1	]		
[	3 , -2 , -2 , 1	]		
[	3 , -2 , 0 , -1	]		
[	3 , 0 , -2 , -1	]		
[	3 , 2 , -6 , 1	]		
[	5 , -2 , -2 , -1	]		
[	5 , 0 , -6 , 1	]		
[	5 , 2 , -8 , 1	]		
[	5 , 2 , -6 , -1	]	---	Final M-partition

Figure 3.3 M-partitions for N-partition [1,2,2,1]

Np = 3
Nn = 3
m = 0

02	18	22	01
02	18	22	02
02	18	22	04
02	18	24	01
02	18	24	02
02	18	24	04
02	28	22	01
02	28	22	02
02	28	22	04
02	28	24	01
02	28	24	02
02	28	24	04
02	30	22	01
02	30	22	02
02	30	22	04
02	30	24	01
02	30	24	02
02	30	24	04
04	18	22	01
04	18	22	02
04	18	22	04
04	18	24	01
04	18	24	02
04	18	24	04
04	28	22	01
04	28	22	02
04	28	22	04
04	28	24	01
04	28	24	02
04	28	24	04
04	30	22	01
04	30	22	02
04	30	22	04
04	30	24	01
04	30	24	02
04	30	24	04

Figure 3.4a SD-words in M-partition [-3,2,2,-1]

02	18	22	01
04	28	24	02
	30		04

Figure 3.4b SD-chains for seed 02 18 22 01

If however M_1 and M_2 belong to the same N-partition such that,

$$M_1 = \begin{bmatrix} n(P1) & n(P2) & n(N1) & n(N2) \\ m(P1) & m(P2) & m(N1) & m(N2) \end{bmatrix} \quad \text{and}$$

$$M_2 = \begin{bmatrix} n(P1) & n(P2) & n(N1) & n(N2) \\ m(P1)" & m(P2)" & m(N1)" & m(N2)" \end{bmatrix}$$

then $M_1 < M_2 \iff$ (1) $m(P1) < m(P1)"$ or

(2) $(m(P1) = m(P1)")$ and $m(P2) < m(P2)"$ or

(3) $(m(P1) = m(P1)")$ and $(m(P2) = m(P2)")$

and $m(N1) < m(N1)"$ (3.6)

We can now say that an M-partition M_1 "is less than" another M-partition M_2 if the above is true for M_1 and M_2.

Thus for any two arbitrary states S1 and S2 within a basis, where S1 and S2 belong to different N-partitions N_1 and N_2 respectively, then;

$$S1 < S2 \iff N_1 < N_2$$

Similarly if S1 and S2 both belong to the same N-partition but different M-partitions, M_1 and M_2 respectively, then;

$$S1 < S2 \iff M_1 < M_2$$

If S1 and S2 are within the same M-partition then they are simply ordered according to normal numerical ordering.

Thus using these definitions all states within a basis can be ordered. It is this partitioning and ordering that the PG uses to produce all the SD-words for a given nucleus.

As an example of what has just been described figure 3.2 shows all the N-partitions (note that the definition of connected N-partitions will be given later) for the $^{30}_{15}P$ m=0 nucleus under the representation given in fig. 3.1. The N-partitions are given in order, with the lowest, under the definition given in 3.5, shown at the top. Figure 3.3 shows, in order, all the M-partitions contained in the [1, 2, 2, 1] N-partition. Finally figure 3.4a shows all the SD-words, (in hexadecimal), within the [-3, 2, 2, -1] M-partition.

3.2 Secondary Generator Methods

As has been said, for every state, $|e_n\rangle$, that the PG produces, a column within the Hamiltonian is defined. This column of the matrix must then be searched in order to find all the other states, $|e_m\rangle$, such that $\langle e_m|H|e_n\rangle \neq 0$. The task of searching the column to find non-zero matrix elements involves generating the basis list and then comparing each state with the prime state $|e_n\rangle$. If a state is two or less particles different from the prime state then a non-zero matrix element has been found. The ordered basis of SD-words must therefore be generated and searched for each state in the basis, although as has already been stated, in H-mode only the states up to the current prime state, i.e. the diagonal element, are compared.

The task of generating the basis list for each prime state is performed, in hardware, by the SG. The SG is not a completely autonomous piece of hardware, that is it will not generate the complete basis of SD words unaided. However the SG will independently generate, in order, all the SD-words belonging to an M-partition in response to being sent the initial SD-word for that partition. The task of driving the SG by sending these initial states, called ***seed states***, is part of the function of the PG.

In addition to H-mode there is, fortunately, another means whereby the SG need only produce certain sections of the basis for searching, thereby reducing the number of states it must generate. This is due to the fact that for each prime state there exist certain sections of the basis which cannot possibly contain any states which contribute non-zero matrix elements. The sections of the basis which are generated and searched for a given prime state $|e_n\rangle$, are those N-partitions

$$N^* = [\, n(P1)^* \,|\, n(P2)^* \,|\, n(N1)^* \,|\, n(N2)^* \,]$$

such that

$$| n(P1) - n(P1)^* | + | n(P2) - n(P2)^* | + | n(N1) - n(N1)^* | + | n(N2) - n(N2)^* | <= 4 \quad (3.7)$$

where the prime state $|e_n>$ belongs to the N-partition

$$N = [\ n(P1)\ |\ n(P2)\ |\ n(N1)\ |\ n(N2)\ ]$$

If equation 3.7 is not true for a particular N-partition N^* relative to N then all the states in N^* must have more than 4 differences relative to all the states in N, i.e. more than 2 creations and 2 annihilations. It can be seen then that if equation 3.7 is true for one of the states which belongs to N then it is true for all states in N. We say that two N-partitions are ***connected*** if they are related by eqn. 3.7. Thus for all the prime states which belong to a given N-partition, N, the SG need only search those N-partitions which are connected to N.

As has been said it is the PG's task to send the seed states to the SG. A table of these seed states is built by the PG every time the new prime state belongs to a different N-partition. This table will contain the initial SD-word of each M-partition within all the N-partitions which are connected to the N-partition which the prime state belongs to.

As an example figure 3.2 identifies all those N-partitions which are connected to the [1, 2, 2, 1] partition. In H-mode, of course, the SG need only search those N-partitions up to and including the one in which the current prime state resides, since only half the matrix is being searched. In figure 3.4a the first word shown (= 02 18 22 01) is the seed state for that particular M-partition and the remaining 35 words are those which the SG must produce in response to being sent it.

It can be seen that each of the individual SD-bytes in all the states in fig. 3.4a take on only a few different values. These different values are shown for each SD-byte in figure 3.4b, with each column corresponding to the SD-byte above it. Each of the four different sequences of numbers in the four columns of fig. 3.4b is called an ***SD-byte chain***. Each SD-byte chain is a list of the values, in numerical order, that each SD-byte can assume in a particular M-partition, under

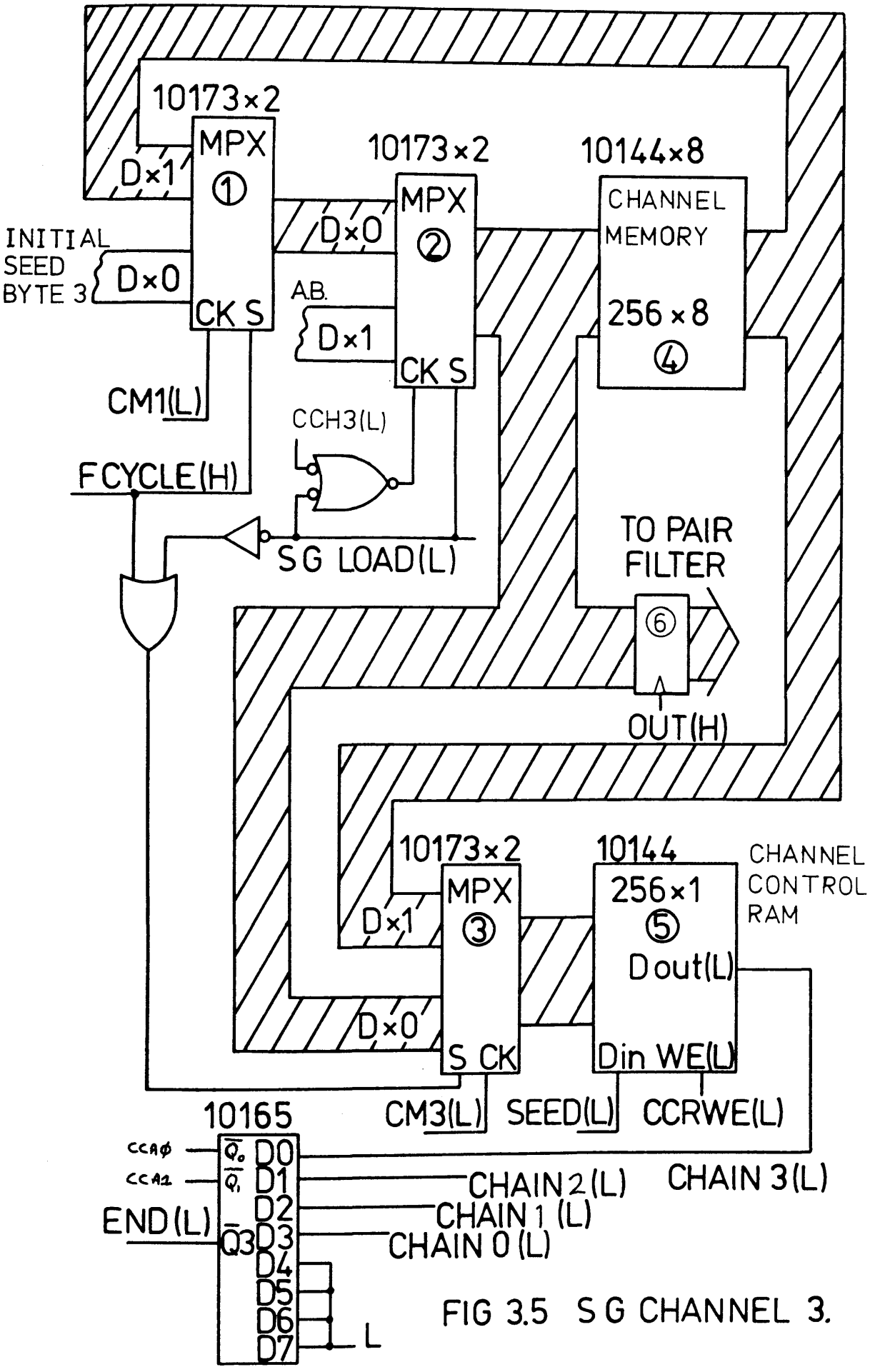


FIG 3.5 S G CHANNEL 3.

the constraints of constant n_i and m_i imposed by the partitioning.

To produce the states of an M-partition the SG is built as four separate byte-wide channels, ***SG channels 0 to 3***, corresponding to the four SD-bytes that make a SD-word. Associated with each channel is a block of 256 x 8 RAM, the ***channel memory***, which stores the SD-byte chains for that channel. When a seed state is sent down to the SG each SD-byte in the seed is used to address the appropriate channel memory.

Figure 3.5 shows the hardware for channel 3 (corresponding to SD-byte 3) although there is little difference for any of the other channels. During the first cycle of the SG the appropriate byte of the seed word enters the SG via the Dx0 input of multiplexer (1) and is latched into the output register of multiplexer (2). The signal FCYCLE(H) is only active during the first cycle of the SG and so only then will the multiplexers (1) and (3) use the Dx0 inputs (note the (H) suffix on the signal name denotes that it is active high, while an (L) suffix denotes an active low signal).

The output of (2) addresses the channel memory and is also the output of the SG to the Pair Filter via the register (6). The byte which is read out of each of the four channel memories is the next element in each of the SD-byte chains. The output of each of the channel memories is fed back round and latched first onto the output of (1) and then onto the output of (2). This next byte in the SD-byte chain now addresses the channel memory and the output it produces is the next member of the chain, and so on.

After the first cycle of the SG only the least significant channel, i.e. channel 3, has its multiplexer (2) clocked round. Therefore the output of the top 3 most significant channels stays the same, initially equal to the bytes of the seed state, while the lowest channel is clocked through the elements in its SD-byte chain.

When the last element in the chain for channel 3 addresses the channel memory it produces the first element at its output. It is only

when this byte is clocked round to the output of the SG, i.e. the output of (2), that the next most significant channel, channel 2, has its multiplexer (2) clocked round so that the next byte in its chain is then presented at its output. Channel 3 now has the first byte in its SD-byte chain at its output, while channel two has the second byte in its chain at its output. When both channels 3 and 2 reach the end of their chains channel 1 is then clocked round and so on. In this way the SG acts like a 4 byte counter, with each of the bytes only taking on a limited number of values, i.e. the elements of their respective SD-byte chains. The SG thus produces in numerical order all the SD-words present in an M-partition, in response to being sent a seed state.

The contents of the channel memories are thus organised as closed self-addressing chains. For example taking byte 3 of the example given in figure 3.4b, the contents of location $01 ($ signifying a hexadecimal value) would be $02, the contents of location $02 would be $04 and the contents of location $04 would be $01.

The task of recognising when a chain has come to an end is performed by the 256 x 1 ***channel control RAM*** (5). Initially this RAM will contain all ones, but when a new seed state is latched into the SG then a zero is written in to the RAM at the location addressed by initial seed SD-byte. As each byte in the chain is read out of the channel memory, it addresses the channel control memory. Thus as the different bytes of the chain address the memory only when the first element in the chain addresses it will it output a zero. This is then the signal that the channel has reached the end of its chain. When all channels reach the end of their chains then the M-partition has been exhausted. The control memory then has a one written back into it, overwriting the zero, and a new seed is requested.

The channel memories are initialised at the start of SMP system processing by the PG. The SGLOAD(L) signal is driven low by the PG thus switching the multiplexer (2) over to its Dx1 inputs which are connected

to the PGs address bus (A.B. fig. 3.5). The data inputs and the data outputs of the channel memories are connected to the PG's data bus so that it can initialise, and verify, their contents. The contents of the channel control RAM are automatically initialised to all ones when the channel memories are written to.

The SG must also keep track of the index number of the the SD-words it produces so that the MCMs can identify the appropriate vector element which is to be used. To this end the SG has a 20-bit counter, called the ***Secondary Index Counter (SIC)***, which is clocked up each time the SG produces a new state. However as has been said the SG does not produce all the SD-words in the basis but only those belonging to connected N-partitions. For this reason the SIC must have the capability to be initialised at the start of each new N-partition that the SG produces, since the N-partitions which the SG produces will not in general be contiguous. The PG has the task of initialising the SIC and must therefore maintain a table, called ***NUMTB***, of the index numbers of the initial states in all the N-partitions. When the PG prepares a new table of seed states it must also prepare a table of initial indices selected from NUMTB. This ***initial number table (INT)*** will contain the indices of the initial SD-words in each of the N-partitions connected to the currently active partition.

We have now given a more complete description of the task of the SG and of the methods it uses to fulfill this task. Section 3.5 will go into greater detail and discuss its hardware implementation. However first the Pair Filter and MFG buffer must be described more fully.

3.3 Pair Filter Operation

Once a secondary SD-word has been generated by the SG it is passed directly to the Pair Filter. There it is compared to the prime state $|e_n\rangle$ to determine whether it is two particles or less different. First

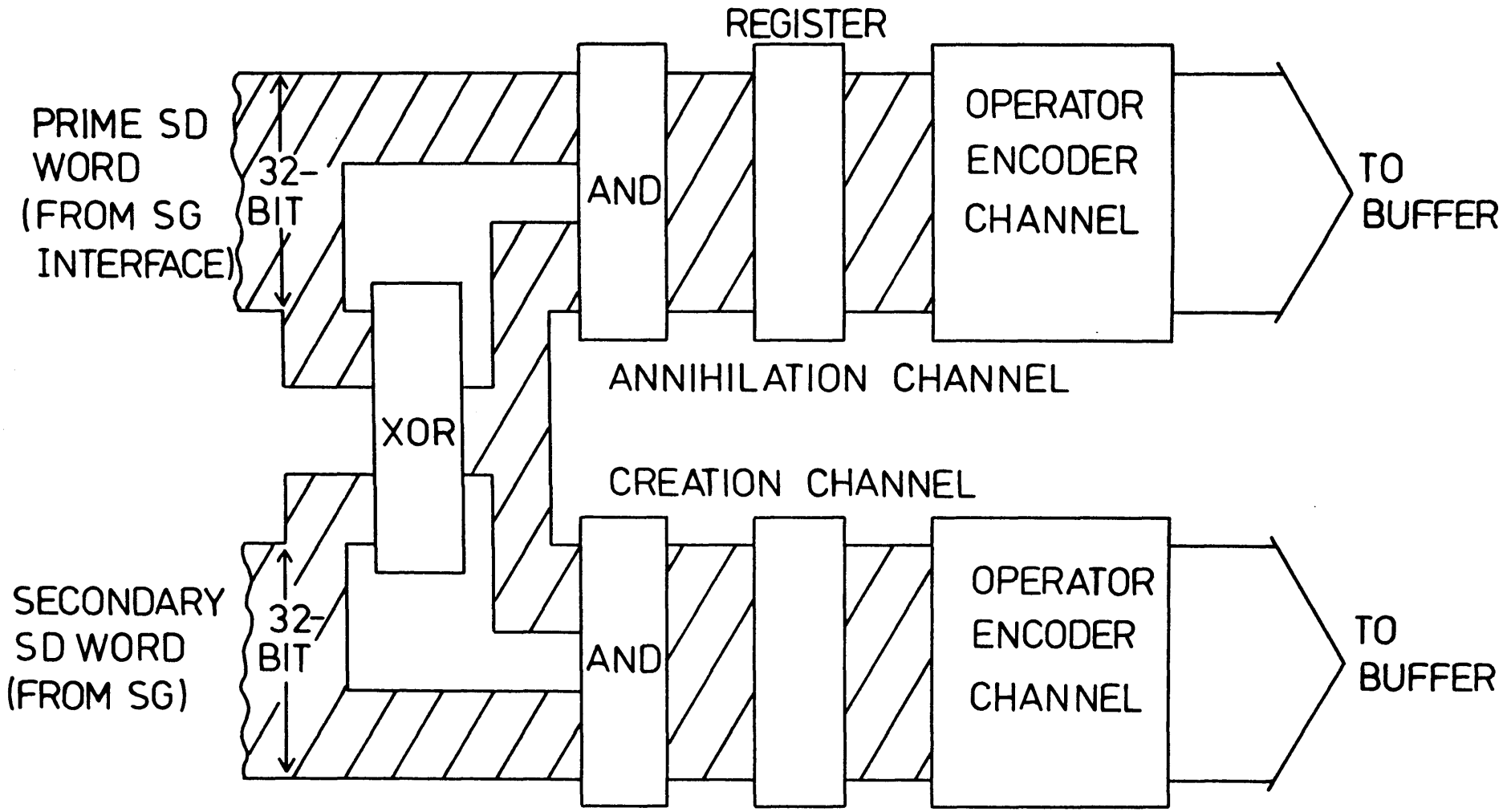


FIG. 3.6 PAIR FILTER

the orbitals which differ between the two states must be identified. This is performed by logically exclusive-ORing the two SD-words that represent $|e_n\rangle$ and $|e_m\rangle$, figure 3.6. The resultant 32-bit word will have ones only in those positions which differed, thus marking out those orbitals in which a particle was either created or destroyed. To then determine those particles in $|e_n\rangle$ which have been destroyed the output of the XOR array is logically ANDed with $|e_n\rangle$. The particles which have been created in the prime state are determined by logically ANDing the state $|e_m\rangle$ with the output of the XOR array. The two resultant 32-bit words are then latched into registers feeding separate ***operator encoder channels (OEC)***.

The output of each OEC is a 5-bit word giving the index of the least significant set (i.e. high) bit stored in the input registers. These output words, the index of an annihilation/creation operator depending on the channel, are latched into two 5-bit registers. The index of the next least significant set bit on the input registers of the OECs is then determined and latched at the output. If after this it transpires that there is another set bit on both the input registers then there must have been more than 2 particles difference between the two input SD-words, therefore $\langle e_m|H|e_n\rangle = 0$. If however there are no more bits left then the four operator indices are written into the MFG buffer.

The operation of the PF is completely pipelined with that of the SG. That is, as the SG is in the process of producing a new state, the PF is processing the last state the SG produced. The SG and PF thus have the same ***major cycle***, i.e. the time taken to process a state. The major cycle time for the SG and PF is currently 13 clock cycles.

3.4 MFG Buffer Operation

The Task Setup Word (TSW) written into the Buffer for each state passed

by the PF has three subwords contained within it. These are made up as follows;

Subword 1: a 20-bit word consisting of the four 5-bit operators produced by the PF,

Subword 2: the 20-bit output of the SIC which gives the index, m, of the secondary state, $|e_m\rangle$.

Subword 3: a 2-bit code to identify whether the TSW word refers to a 0, 1 or 2-job. This code is also produced by the PF.

The operation of reading and writing to the buffer is pipelined with the operation of the SG and PF. To ensure as far as possible the uninterrupted operation of the SG and PF they must not be delayed by buffer read/write operations. Therefore although only a few states are actually passed by the PF the buffer must still have the capability of performing a write operation on every major cycle of the SG and PF. The write cycle time of the buffer must therefore be at most 13 clock periods. However read requests from the MCMs to the buffer, which are completely asynchronous to the MFG operation, must also be fitted into this cycle so that the SG and PF are not held up. To this end the 13 clock period major cycle of the MFG is split into two subcycles for the buffer; one for buffer read operations and the other for buffer writes. The buffer must therefore synchronise any read request to its read subcycle. On some occasions however the SG and PF will be halted, e.g. if the buffer is full, in which case the reads can take place at anytime.

The buffer must keep a track of how many locations within it are used at any time and from this provide signals to indicate whether it is empty or full. These signals are then used to stop any more reads from the buffer or to halt the SG and PF from producing any more states.

3.5 MFG Hardware Implementation

The MFG was first run successfully as a complete unit towards the end of 1983, at a clock rate of 50 MHz. It has now, after a major revision of its timing control and a number of other changes to the design, been uprated to run at 112 MHz. This section will detail the updated MFG hardware, as well as identify those sections of the hardware which currently impose the upper limit om its clock speed.

The SG, PF and buffer read/write control logic are all built with Motorola *10K series ECL* gates. This high speed family of logic devices has typical gate propagation delays of 2 ns, rise and fall times of 3.5 ns and offers a wide range of SSI devices and functions [MECL86]. In some key areas of the timing circuit Motorola 10KH ECL devices were used. The 10KH series is fully compatible with the 10K family but has an improved performance, e.g. providing typical propagation delay of 1 ns for the same power consumption (typically 25 mW per gate). 10KH devices also provide improved noise margin and reduced parasitic capacitance on inputs allowing faster rise and fall times.

With the fast edge speeds and low propagation delays of ECL devices path lengths can approach the wavelength of the signals. Thus any line which is improperly terminated will produce reflections causing serious distortions in the waveform [Ch86]. As a result of this, transmission line practices must be used, requiring each line to be properly terminated at its end with a load approximately equal to the characteristic impedance of the line [MECL83]. This practice is facilitated by the open emitter output used on all 10K devices. This also allows "wire-ORing" of outputs, i.e. the ability to produce an OR function between a number of outputs simply by connecting them directly together.

A full power (equal -5.2 V for 10K ECL) and ground plane on the circuit board also helps to reduce the impedance of signal lines and so

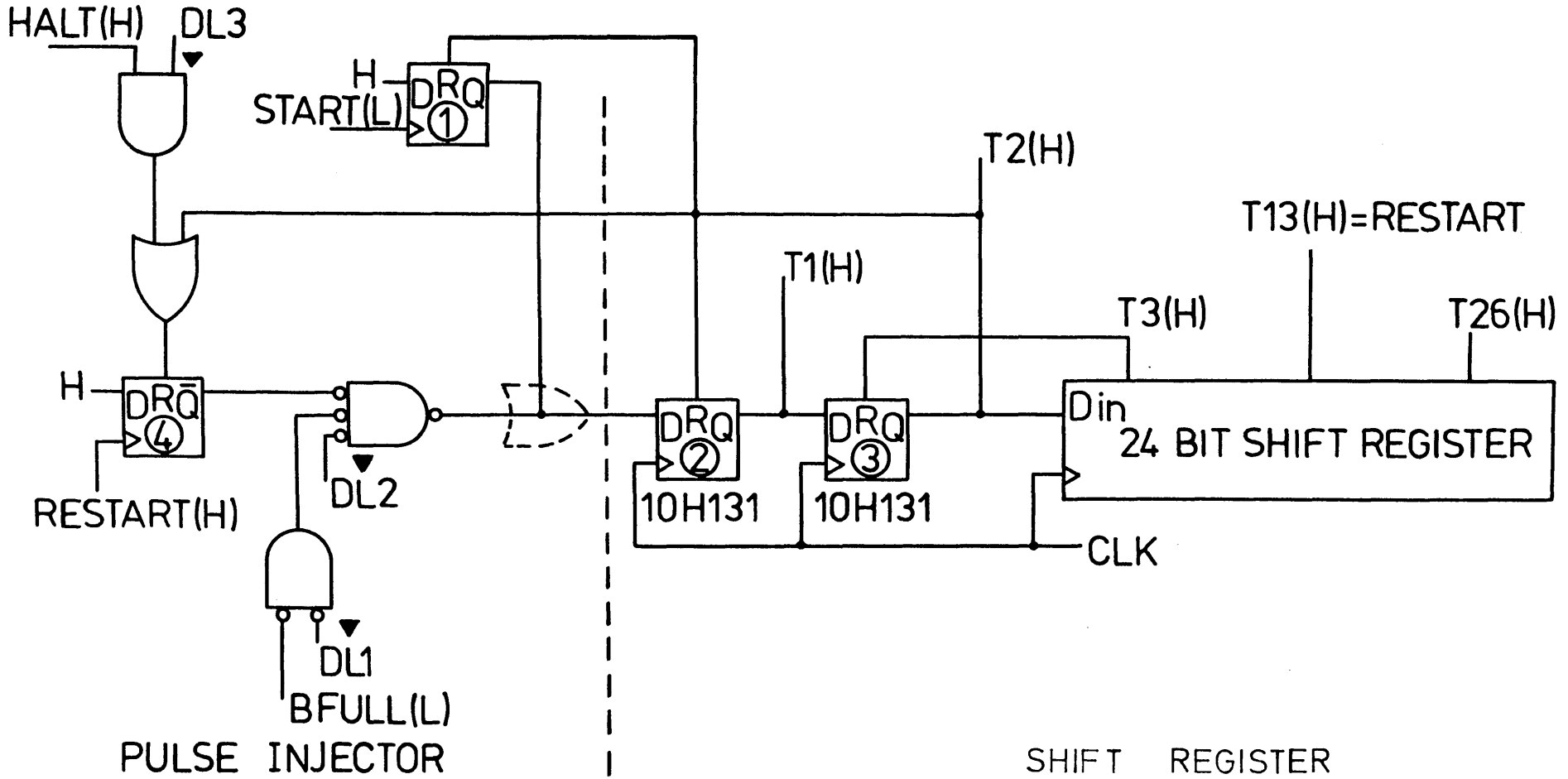


FIG. 3.7 TIMING AND CONTROL UNIT

minimise cross-talk between signals [MECL83]. The circuit boards used, as well as providing a full power and ground plane, also provided positions for the terminating resistor networks. Single-in-line (SIL) resistor packs were used, providing seven 100 ohm resistors connected to a common terminal. This was connected to a -2.0 V supply to provide an active pull down termination.

3.5.1 Timing and Control Unit

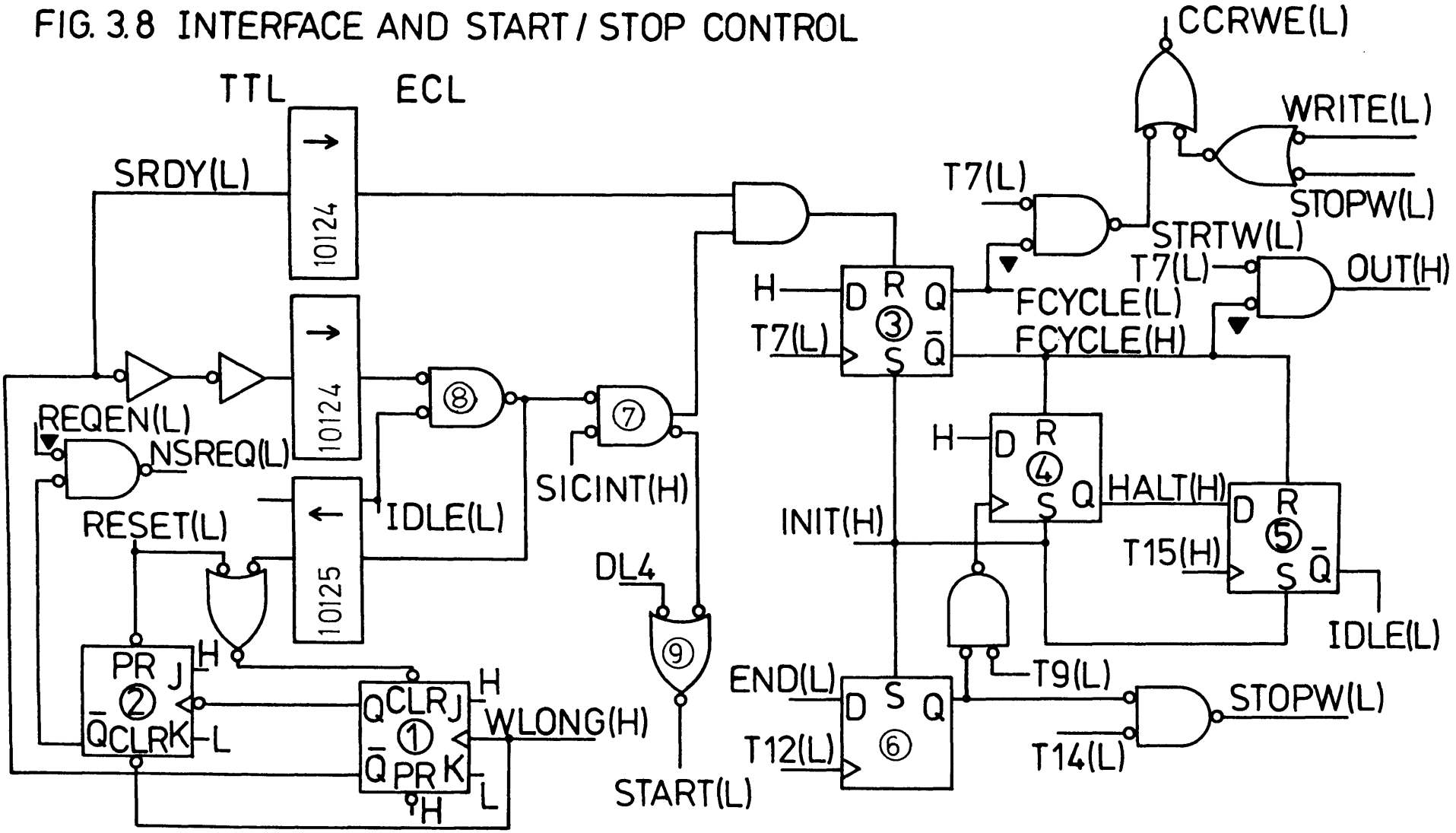
The ***timing and control unit (TCU)*** provides the main timing and control signals for all the major units within the MFG. It also controls the synchronisation of the three stages within the pipeline, i.e. the SG, PF and buffer.

The TCU can be separated into two functional subsections (fig 3.7); the pulse injector and the 26-bit serial-in-parallel-out shift register. A pulse is injected into the shift register by the D input of (2) being high on a positive edge of the clock. This happens in two ways;

1/ **START:** This active low signal will inject a pulse into the TCU on its back edge. START(L) is only activated when the SG commences processing a new seed state. Thus if the SG is idle and waiting for a new seed state then START(L) will only be activated when the PG sends one. Alternatively if the SG finishes processing a seed state and a new seed is already waiting then START(L) will be activated immediately.

2/ **RESTART:** When the shift register is triggered, a pulse one clock cycle long will travel along it causing each output to go high for one clock period, starting at T1 and ending at T26. When the pulse reaches T13 another pulse will be injected into the shift register. This therefore generates the 13 clock period major cycle of the MFG system. Thus on most occasions there will be two pulses travelling through the shift register, separated by 13 clock cycles. Exceptions to this are on the first cycle after the TCU has been started and on

FIG. 3.8 INTERFACE AND START / STOP CONTROL



the last cycle before it becomes idle.

There are two conditions that can stop a new pulse being injected in:

i) The SG finishing an M-partition: this condition is signalled by the HALT(H) line. Obviously if this happens then the SG must be brought to a halt but the rest of the MFG pipeline must be allowed to empty before they are halted. In this case RESTART is barred from injecting a pulse into the shift register and instead must wait for START to be activated, signifying the arrival of a new seed state. However timing signals must continue to the PF and buffer and so the shift register is allowed to continue on, generating pulses up to T26. It is from the latter half of the shift register that the PF and buffer receive most of their timing signals.

ii) The buffer becoming full: this condition is signalled by the BFULL(L) line. When this occurs RESTART is blocked from injecting new pulses in. Only when a read is executed from the buffer and BFULL(L) is thus negated is a new pulse allowed into the shift register. As in i) above the MFG pipeline is allowed to empty. Since this could mean another request to write to the buffer if the PF passes the state it is processing, then BFULL is actually made a pre-emptive signal. That is BFULL is activated when there are still 16 positions left in the buffer. This is more than enough room to store any states allowed through by the PF while the pipeline is being emptied.

Note that the two flip-flops, (2) and (3), at the input to the shift-register serve to synchronise the the input pulse to the clock since both START(L) and BFULL(L) are asynchronous to the system clock.

The lines DL1, DL2 and DL3 are all debug lines controlled by the PG software and used during MFG testing. Their operation will be explained later in section 3.5.10.

3.5.2 SG Interface and Start/Stop Control

The interface between the SG and PG, figure 3.8, is the means whereby

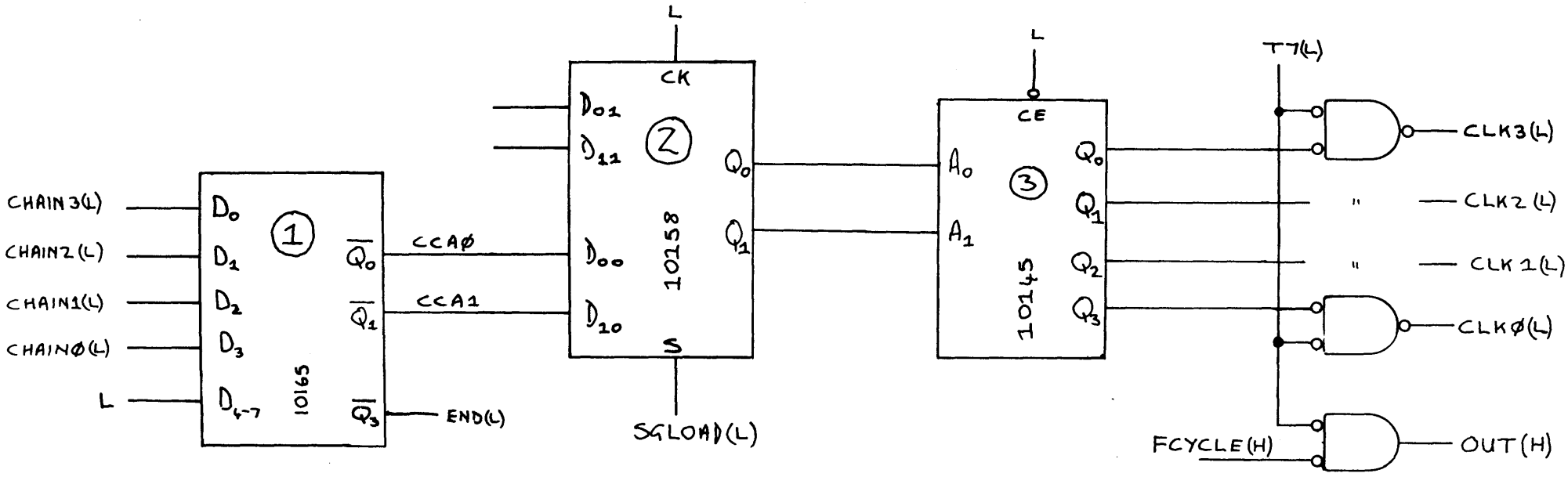
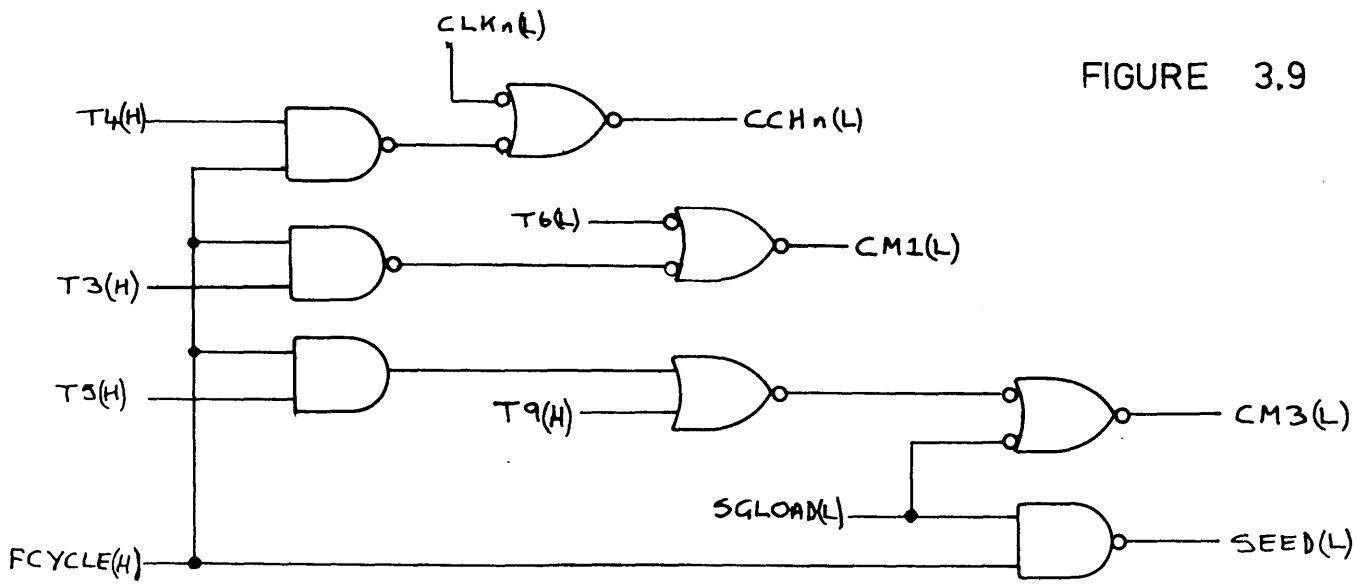


FIGURE 3.9 CHANNEL CLOCKING CONTROL



the SG signals to the PG that it requires a new seed state and whereby the PG then transfers new seed states to the SG.

When the SG has exhausted an M-partition it will activate END(L) (6) causing IDLE(L) (5) to be clocked low thus signalling to the PG that the SG is indeed idle. If the next seed state is available, signalled by SRDY(L), then the SG can start again, otherwise it must wait.

When the PG writes a new seed to the interface then the WLONG(H) (1) signal is activated. This in turn triggers START(L) and also resets (3) indicating, via FCYCLE, that the first cycle of the SG processing an M-partition is in progress. When the SG does start again it signals to the PG, via NSREQ(L), that a new seed is now required. Thus the supply of seed states to the SG by the PG is pipelined with the SG's activity.

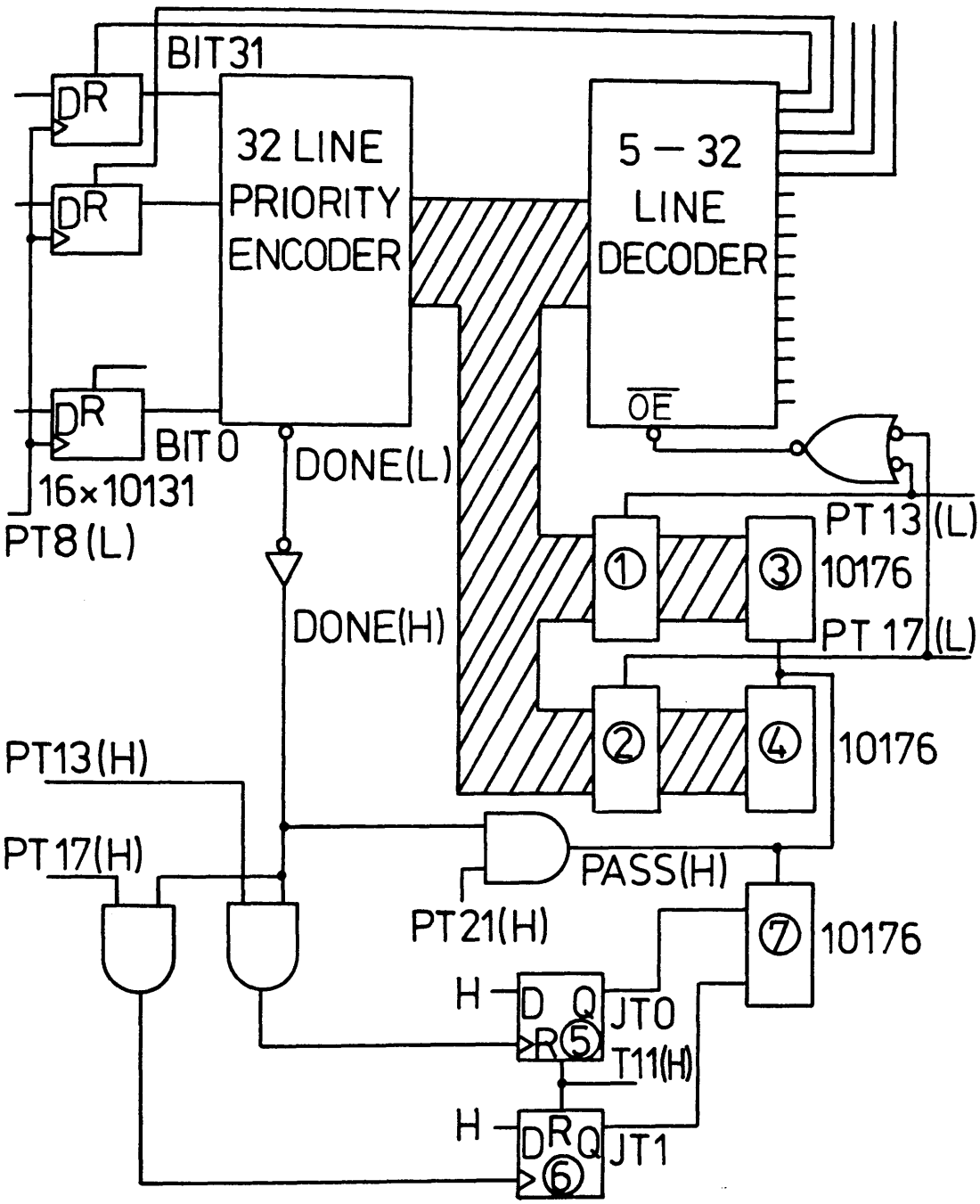
The first cycle of the SG is not used to produce any new states but only to take in the new seed state and initialise the appropriate locations in the Channel Control RAMs. The write pulse to the RAMs, CCRWE(L), is generated on the first cycle of a new seed by STRTW(L) and on the last cycle by STOPW(L).

The WRITE, INIT and RESET lines are initialisation control signals driven by the PG at the start of SMP processing. They are used, among other things, to ensure the correct state of various flip-flops in the MFG control system and to set all bits in the SG channel control RAMs to ones.

3.5.3 Channel Clocking and Control

As has been said the output of the multiplexer (2), fig. 3.5, of the nth channel is only clocked if all the channels of lower significance, i.e. channels n+1 to 3, are also at the end of their chains. The control for the clocking of the multiplexers in each of the four SG channels is shown in figure 3.9.

The decision as to which channels are clocked round is implemented by the priority encoder (1) and the 16x4 RAM (the ***channel clocking***



JT0	JT1	
1	1	0-JOB
1	0	1-JOB
0	0	2-JOB
0	1	(NOT ALLOWED)

PAIR FILTER OPERATOR
ENCODER CHANNEL.
FIG. 3.10

memory (CCM)) (3). The only function of the multiplexer (2) is to allow the PG to initialise the CCM at the start of processing. The inputs to the priority encoder (1) are the outputs of the 4 channel control RAMs ((5) figure 3.5), with the output of channel 3 connected to the highest priority input. The 2-bit output word of (1) is the index of the highest priority input that is high. This output is then used to address the CCM, only the lowest 4 locations of which are used. The CCM is preprogrammed such that the bit pattern which is read out will enable only the appropriate channel to be clocked.

3.5.4 The Pair Filter

Figure 3.10 shows one of the two PF OECs while figure 3.11 details the logic to control its timing. The timing of the PF has been completely revised to allow it to operate at 112 MHz. This has meant increasing the time between the 4 PF timing pulses, so that the OEC now completely utilises the 13 clock period major cycle of the SG. Previously it had only used 8 clock periods to perform its function.

The first timing pulse to the PF, PT8(L), clocks the output from the XOR/AND arrays into two 32-bit registers. Each bit of these registers can be individually reset to a low. The DONE(L) output of the priority encoder signals that all of the input bits are low. Therefore if this output is activated before PT13 then there could have been no set bits in the registers. This indicates that the secondary state and the prime state were identical and that a 0-job has been identified. If DONE(L) is not activated by this point then the output of the priority encoder, which gives the index of the most significant set bit on the input register, is clocked into the 5-bit register (1), (note that bit zero of the input register has highest priority).

PT13(L) is also used to enable the output of the 5-line-in, 32-line-out decoder. This output is used to reset the highest priority set bit in the input register. If DONE(L) is now activated before PT17 then

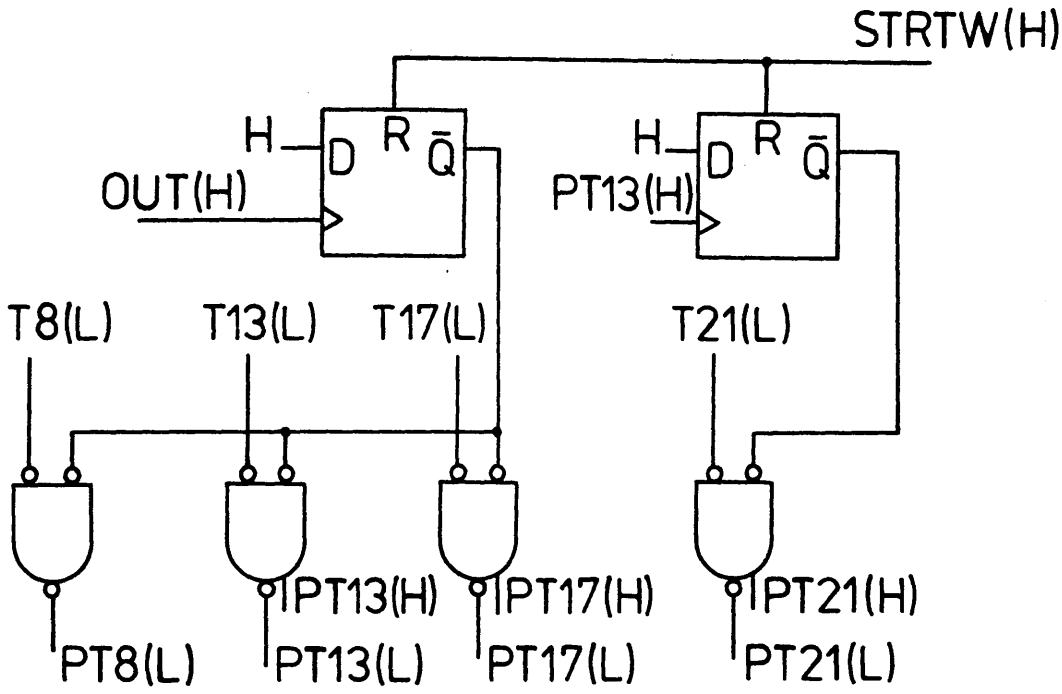


FIG. 3.11 PAIR FILTER TIMING CONTROL.

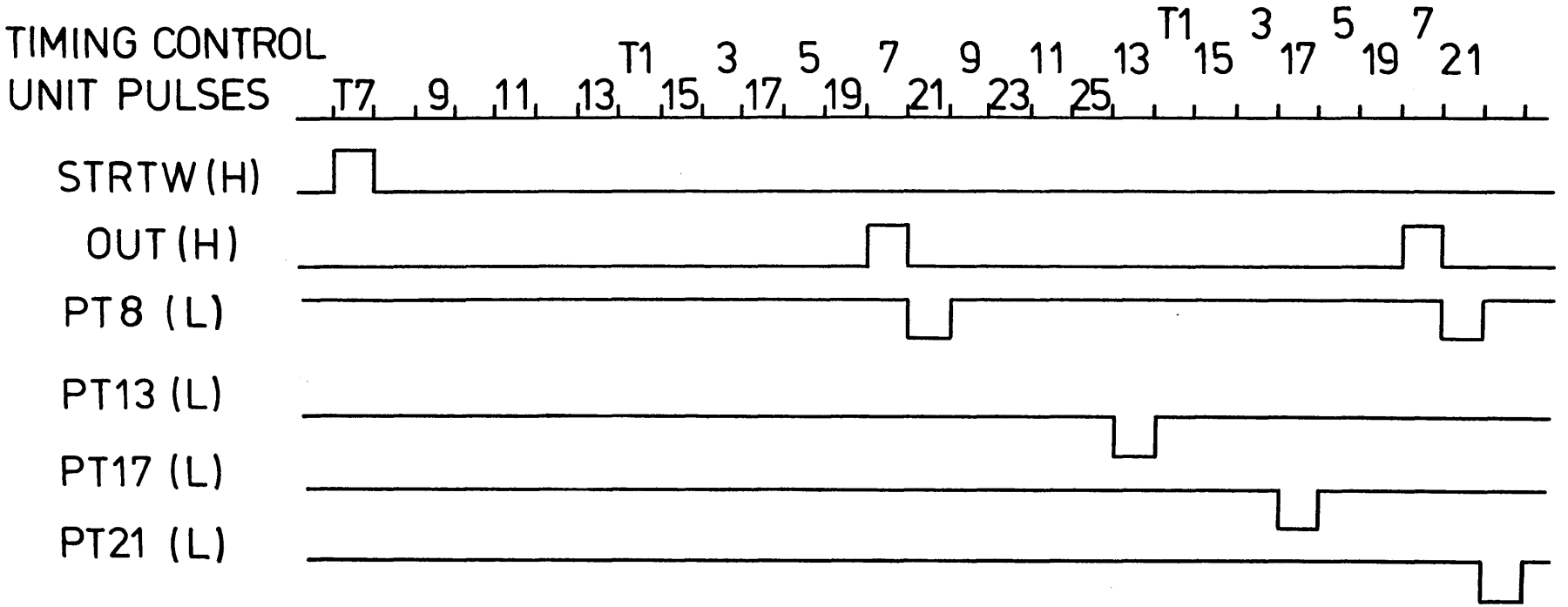


FIG. 3.12 PAIR FILTER TIMING PULSES

there was obviously only 1 set bit in the input word. Thus there was only one particle different between the secondary state and the prime state and so a 1-job has been identified.

If DONE(L) is not activated before PT17 then the above process is repeated for the next highest priority set bit. If after this bit has been encoded and cleared DONE(L) is activated before PT21 then a 2-job has been identified. Otherwise if DONE(L) is not active by PT21 then there must have been more than 2 particles different between the two states and so the secondary state is not passed. In the 0, 1 and 2-job cases the encoded annihilation and creation operators present in the latches (1) and (2) are transferred into latches (3) and (4) by PASS(H). A write enable pulse for the buffer is also generated by PASS(H).

The two flip-flops (5) and (6) generate the job-type bits JT0 and JT1, which also form part of the data word written into the buffer. These two bits are encoded as shown in fig. 3.10.

Figure 3.11 shows the PF timing control circuit. Figure 3.12 details the timing relationship between the different clocks for the PF. The timing pulses to the PF are disabled during the the first cycle of the SG processing an M-partition, since the SG does not produce a state for the PF in this cycle. The STRTW(H) clock, which is generated only on the first cycle of a new seed (fig. 3.8), is used to disable the PF timing clocks. The OUT(H) clock, which is generated on every cycle of the SG except on the first one, is then used to enable the first three clocks to the PF (PT8, PT13 and PT17). PT13 is used to enable the last clock, PT21. This difference is caused by the fact that PT8 and PT21 will actually occur at the same time since they are 13 clock periods apart. Therefore on the first cycle of the PF at the start of a new seed, PT21 must only be enabled after PT8 in order to avoid spurious clock pulses to the PF which could potentially cause unwanted write pulses to the buffer.

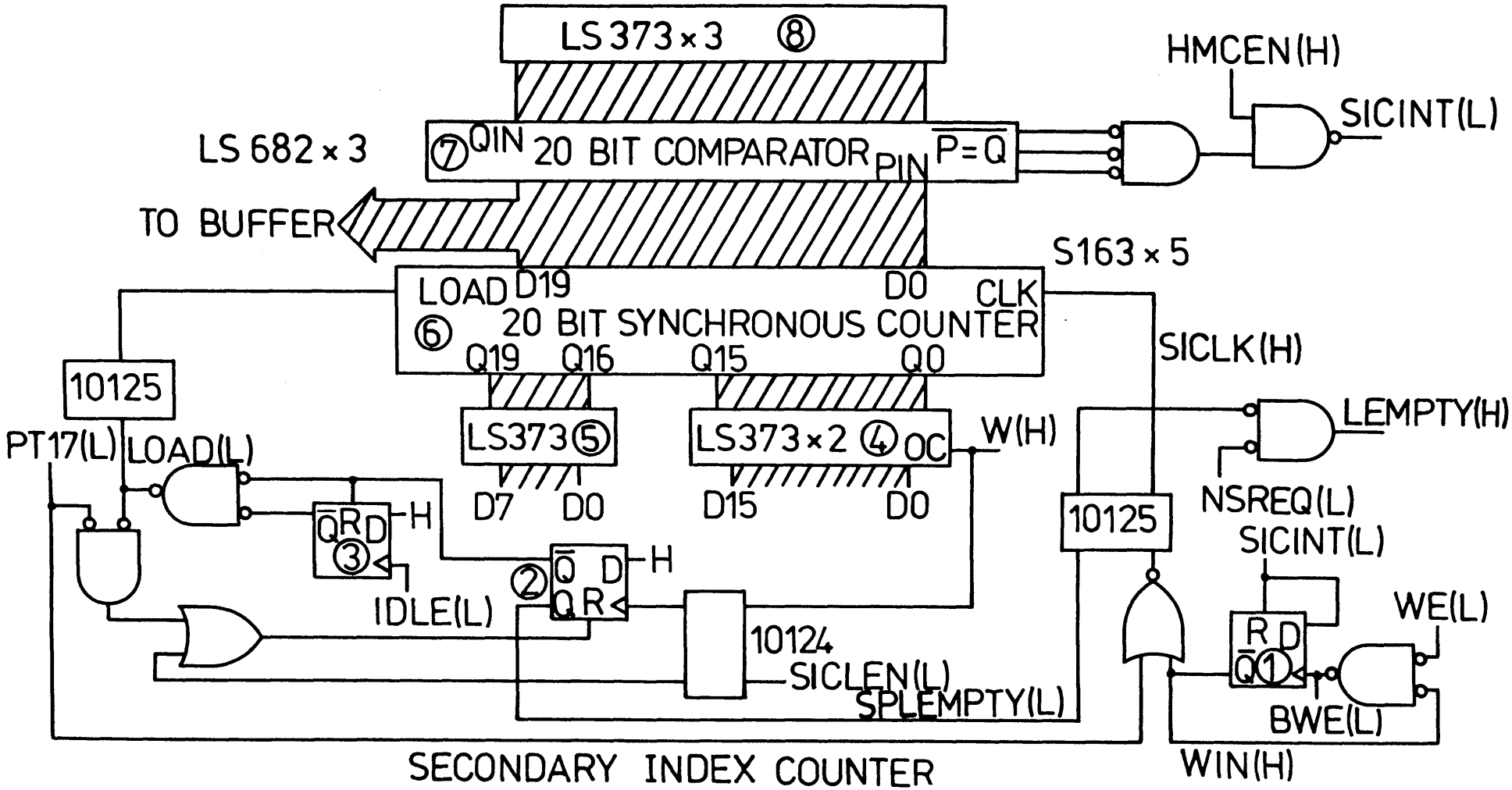


FIG. 3.13 SECONDARY INDEX COUNTER AND H-MODE COMPARATORS

3.5.5 Secondary Index Counter and H-mode Comparator

Since the SG only searches certain N-partitions belonging to a prime state then the PG must preload the SIC with the index of the initial state of every new N-partition processed. The inputs to preload the SIC are fed by the latches, (4, 5) figure 3.13, which can be written to by the PG.

Once the SG starts processing the last seed state of an N-partition the PG must write the initial index number of the next N-partition to the SIC preload latches. The PG can tell when the SG has started processing a seed by testing that NSREQ(L) (fig. 3.8) is active. When the PG writes the initial value to the SIC the flip-flop (2) is clocked, signalling that the SIC preload latches are full. Only then does the PG write the first seed of the new N-partition to the SG interface.

When the SG finishes the old N-partition and starts processing the new seed for the new N-partition IDLE(L) will be driven low and high again, (see fig. 3.8 for the circuit which generates IDLE) thus activating the LOAD(L) signal. The SIC is then synchronously preloaded by the first SICLK pulse. The SICLK pulse, which clocks up the SIC and also preloads it, is generated with one of the PF timing pulses, PT17, since the SIC must only be advanced when a new state has been clocked out of the SG.

It is quite possible that an N-partition contains only one M-partition. In such a situation there would only be one seed state for the PG to send down to the SG before requiring to reload the SIC. However it is feasible the SIC has not yet been preloaded with the previous value, even although the SG has started to process the only seed state. This could occur if the last state produced by the SG on the last seed was written into the buffer causing it to go full. Under these conditions the BFULL signal would not stop the SG from starting to process the new seed, it would instead only stop the SG after its first cycle (fig. 3.7). Consequently the PF timing would not yet have been

enabled (fig. 3.11) and so the SIC would not have been preloaded. Therefore the PG must always check SPLEMPTY(L) (fig. 3.13) to determine if the SIC preload latches are empty before writing to them. Since the PG must also check that the SG has started processing the previous seed, tested via NSREQ(L), before writing to the preload latches, a composite signal, LEMPTY(L), is formed. This signal is active only when both the above conditions are true.

While processing in H-mode the SG should only produce states up to and including the diagonal element. The secondary state for the diagonal element will have the same index as the prime state. Thus when the diagonal element has been produced, being identified by its index number, the PG and MFG buffer must be notified. The PG needs to know so that it can abort loading down the seed table for the current prime and move onto the next prime state. The buffer must also know so that any more states in the current M-partition which are passed by the PF will not be written into the buffer. No writes are then allowed into the buffer until the SG has started processing the new prime state.

The output word of the SIC is fed into a 20-bit hardware comparator, (7) figure 3.13. The other input to the comparator is fed by a 20-bit latch (8). This latch is loaded by the PG at the start of processing on each new prime state with the index of the prime state. When the index of the secondary state equals the prime state index then the SICINT(L) signal is activated. This signal interrupts the PG processor and is also sent to the SG and buffer. If the PG is not in H-mode then the HMCEN(H) signal will be inactive thus permanently disabling the SICINT line.

The diagonal element will always be passed by the PF and so the back edge of the write signal, WE(L), which the diagonal element generates is used to produce the write inhibit signal, WIN(H). The WIN(H) signal is used to block any more clock pulses to the SIC as well as disabling writes to the buffer. In this way the SICINT(L) signal

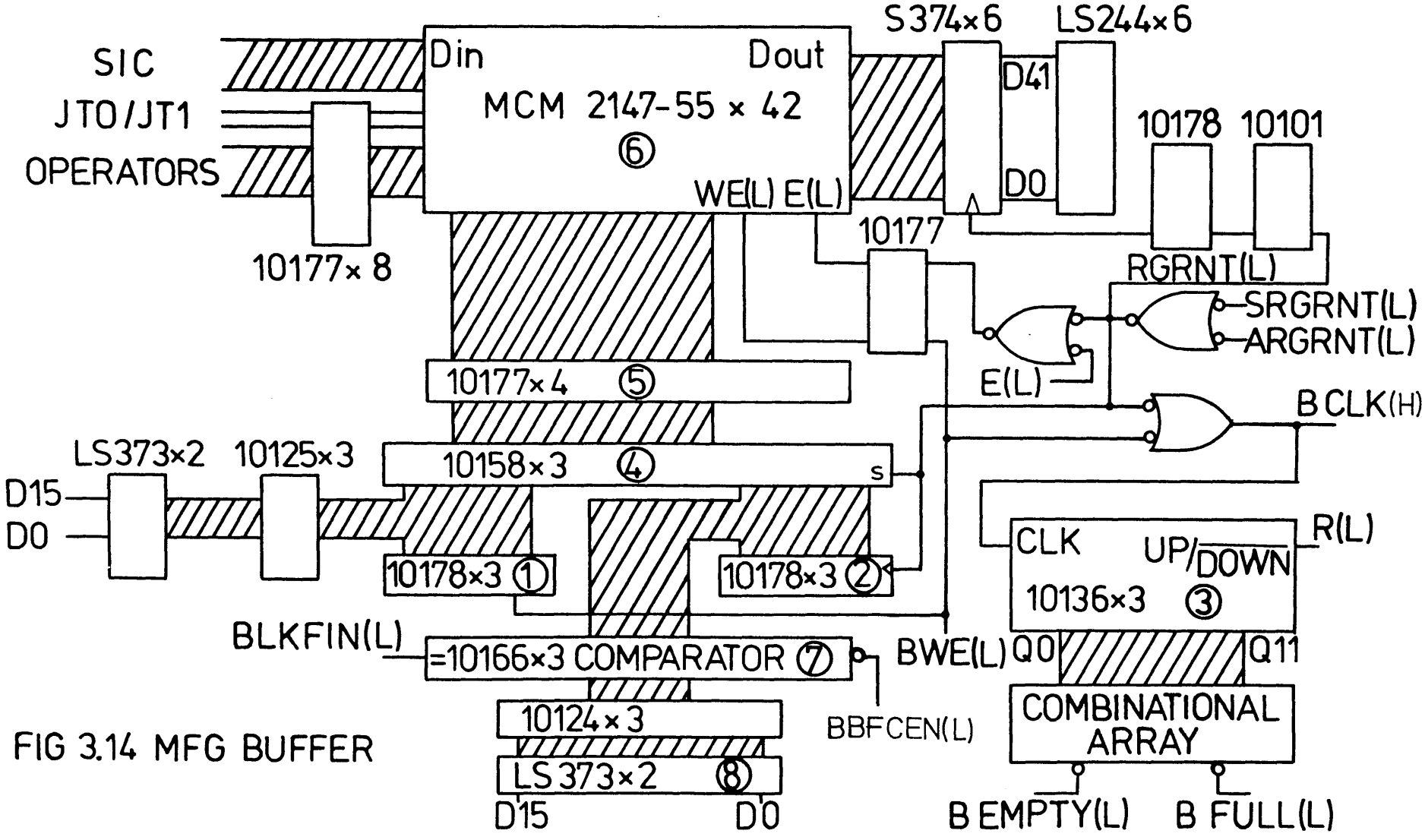


FIG 3.14 MFG BUFFER

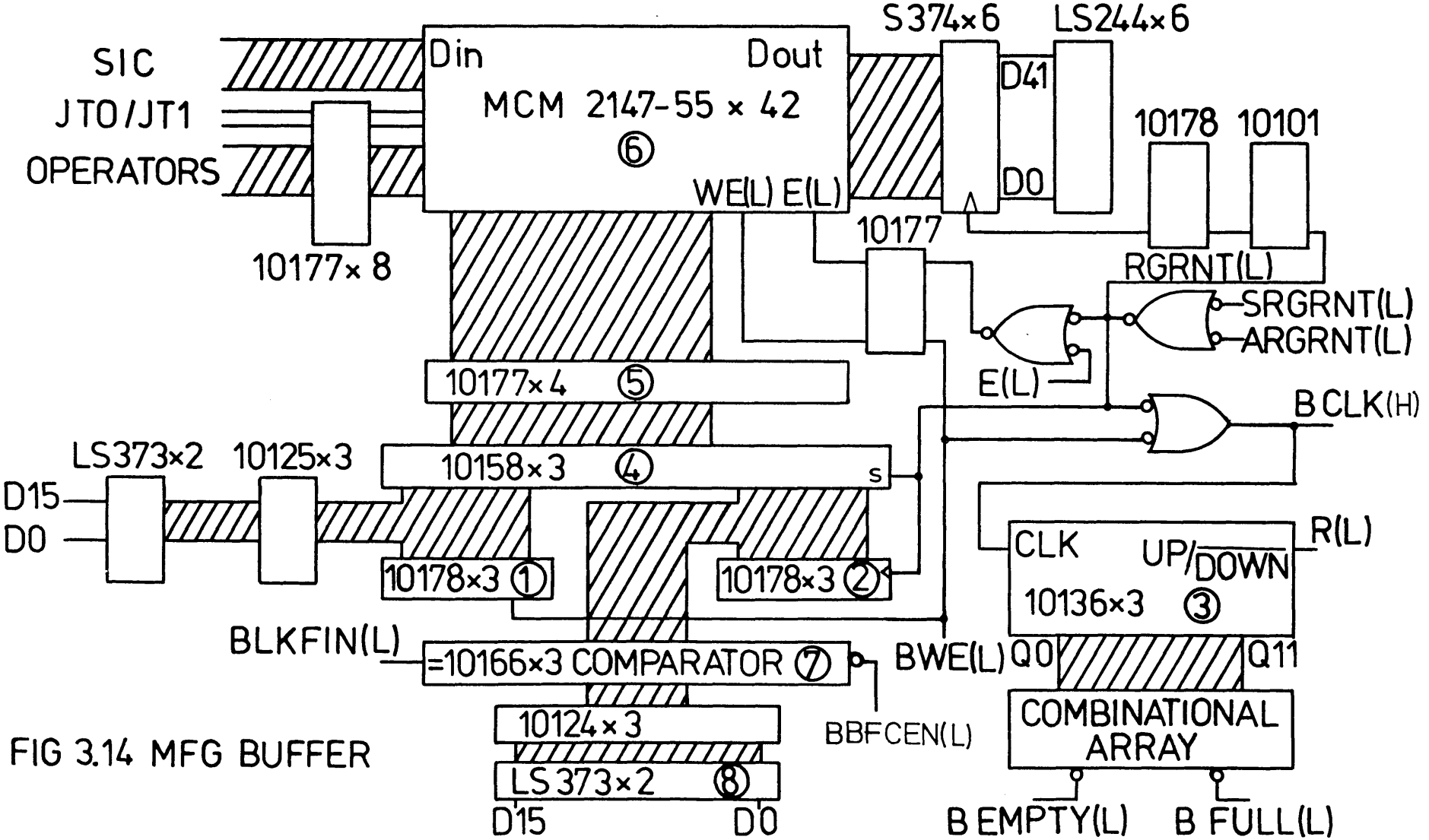


FIG 3.14 MFG BUFFER

remains active until the PG has received and processed the interrupt, when it will initialise the latches (8) with the new prime state index.

The SICINT signal cannot be used to abort the SG/PF from processing an M-partition since this would leave a position in the channel control RAMs (fig. 3.5) with a zero written in it. Therefore the SICINT signal is only used to block any more writes to the buffer after the diagonal element has been written in. As a result the PG must wait until the SG finishes an M-partition as normal before it can go on to process a new prime state. However when the SG finishes it is possible that the SG interface still has a seed from the old prime state ready to be processed by the SG/PF. In order that the SG should not take and process this seed and so waste time, SICINT is used to block any new START pulses, (7) fig. 3.8.

There is the danger that a race will occur between SICINT and IDLE causing a glitch out of (7). SICINT will safely block IDLE as long as it reaches (7) before IDLE reaches (8), thus ensuring no glitches out of (7). This will always happen since the SIC is clocked 11 clock cycles before IDLE (5) thus giving SICINT enough time (in worst case conditions) to reach (7) first. However there is one exceptional condition when SICINT will not be able to block IDLE, but which still ensures no glitches out of (7). That is when SICINT is caused by the last state produced in an M-partition, in which case IDLE is clocked 2 clock cycles before the SIC. This will unfortunately mean that the SG will waste time processing a seed.

3.5.6 The MFG Buffer Implementation

A schematic of the buffer and its control is given in figure 3.14. The requirement that the buffer must be capable of handling both a read and write cycle within the 13 clock period major cycle of the MFG (= 116 ns at 112 MHz) necessitates the use of fast static RAMs. The 55ns cycle time of the Motorola MCM2147 4K x 1 memories only just allows this to be

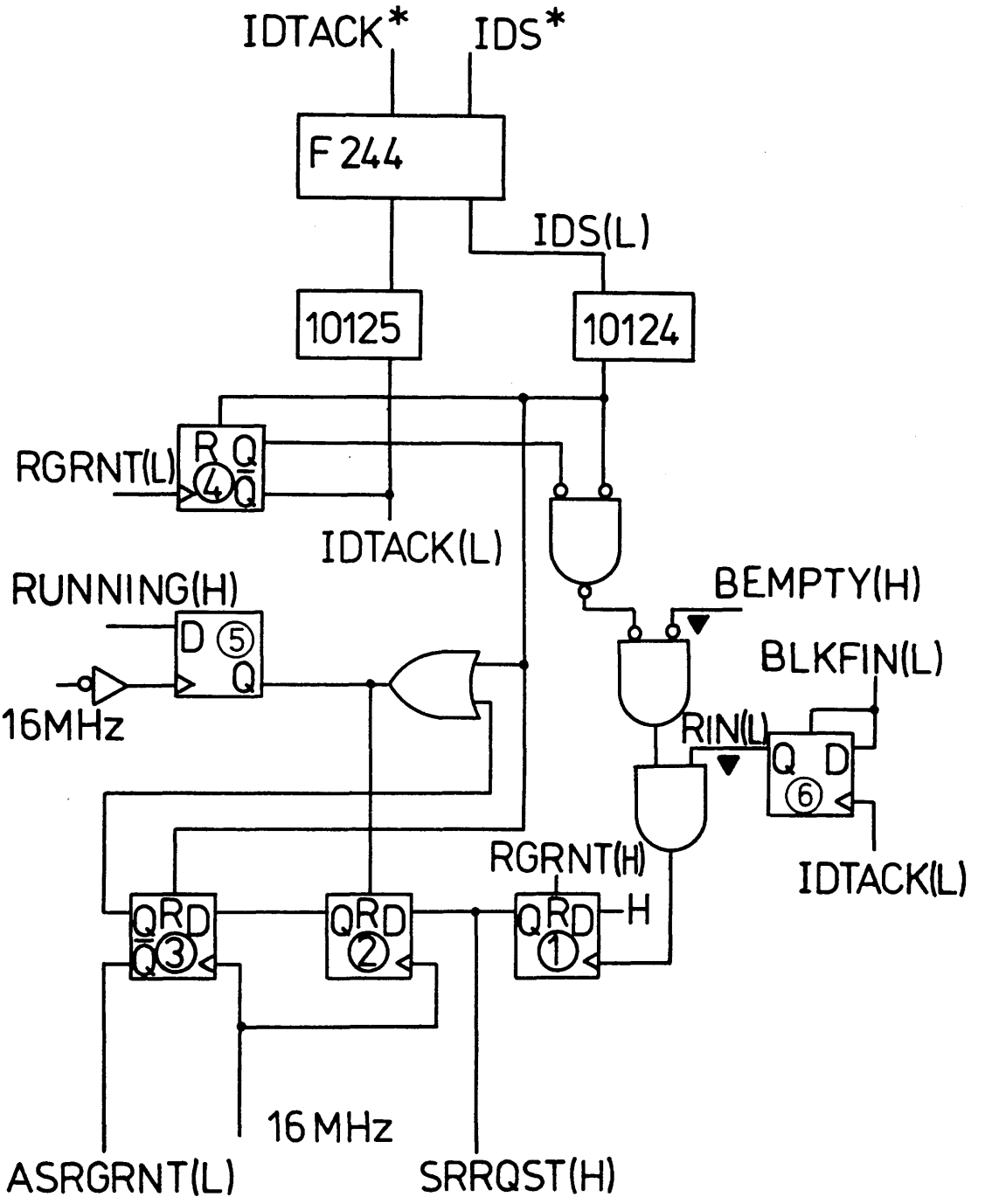
achieved. At the time these memories were one of the main limiting factors in increasing the clock speed of the MFG.

There are three sets of 12-bit counters within the buffer subsystem (1, 2 and 3). (1) and (2), ***the buffer write address counter (BWAC)*** and ***buffer read address counter (BRAC)***, generate the write and read addresses respectively for the buffer and only count up. (3) is the ***buffer word counter (BWC)*** and holds the number of used positions within the buffer. The BWC will count up or down depending on the state of the read signal, R(L). The output of the BWC is used to generate the buffer full and buffer empty signals, BFULL(L) and BEMPTY(L) respectively. This is done by means of a combinatorial AND/OR array.

The multiplexer (4) outputs either the read or write address to the memories depending on the state of the read grant signal RGRNT(L). Thus the output of the multiplexer will default to the write address and only change when a read access is actually being performed. Note that the R(L) signal changes on <u>every</u> major cycle of the MFG, splitting it up into a write and read phase. The RGRNT(L) signal on the other hand is active only when a read is actually taking place.

As has already been noted the parameters within the TSWs which are stored in the buffer do not contain all the data required by the MCMs for each task. That is the MCMs must also know the prime state SD-word and its index. These parameters change very infrequently and only need to be sent to the MCMs when they start processing a new prime state. To achieve this the PG must know when the last TSW for a prime is read out of the buffer.

To this end the PG must read the BWAC, (1), when the SG finishes processing a prime state and before it starts processing a new prime. At this point the BWAC will contain the address of the next position to be written to in the buffer. The PG then writes this address into the register (8) which feeds a 12-bit comparator (7), the ***Buffer Block Finished Comparator (BBFC)***. When the MCMs read the last TSW from the

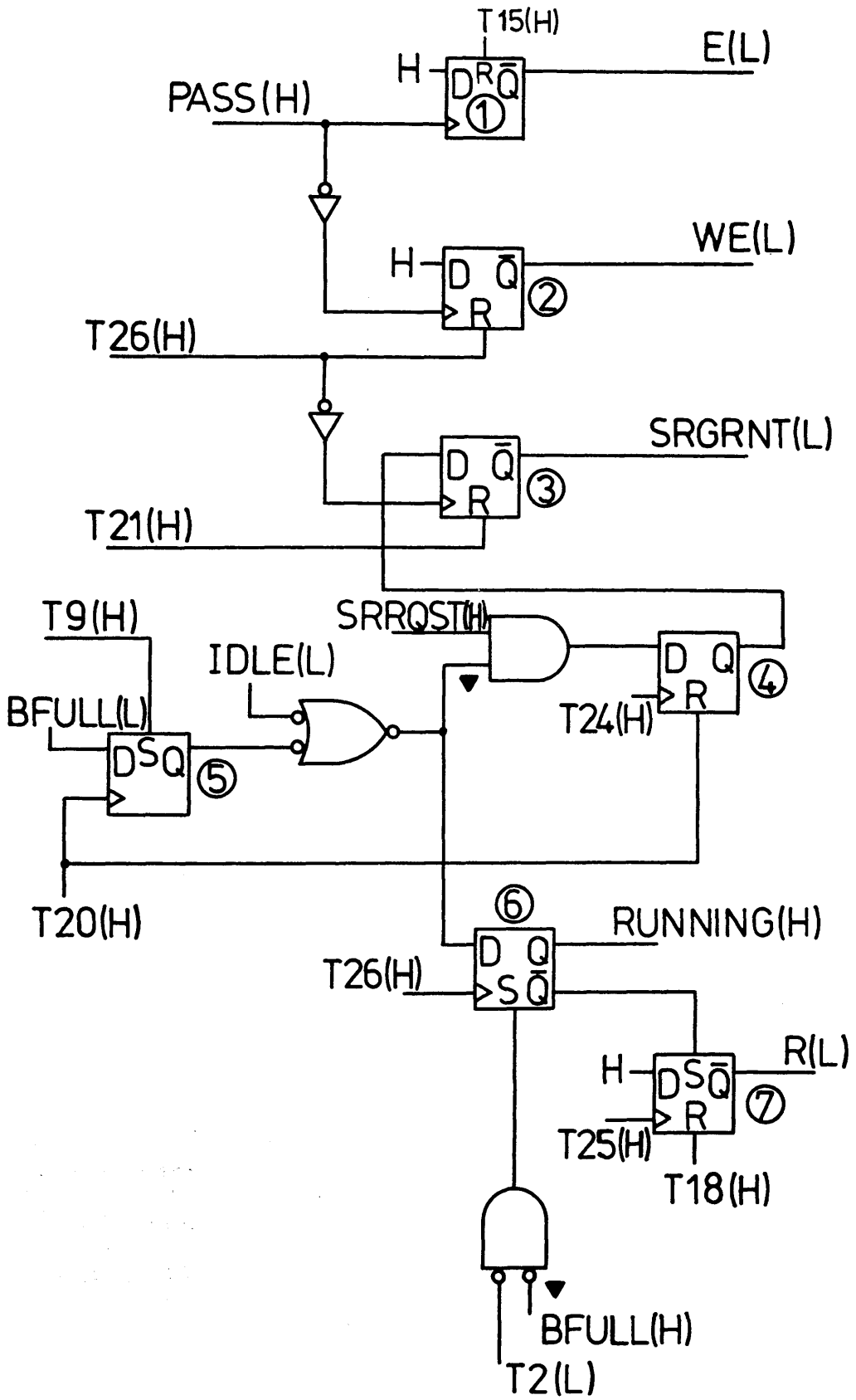


MFG BUFFER READ CONTROL

FIG. 3.15

FIG. 3.16

WRITE AND SYNCHRONOUS READ CONTROL



buffer relating to the old prime, the BRAC, (2), will then equal the contents of the register (8), at which point the BBFC will activate the BLKFIN(L) signal. This signal is then used to interrupt the PG, which then broadcasts the new prime state information to the MCMs. BLKFIN(L) is also used to generate a read inhibit signal which stops the MCMs performing any more reads from the buffer. This is done until the PG has successfully informed all the MCMs of the new details.

3.5.7 MFG Buffer Read Control

All reads to the MFG buffer are performed along I-bus and are controlled by two signals; the data strobe IDS^* and the data transfer acknowledge $IDTACK^*$ (note that the * denotes an active low signal on the bus). A read from the buffer is only initiated when the date strobe IDS^* is activated, figure 3.15. This will latch in a read request on the flip-flop (1), unless either the buffer is empty, BEMPTY(H) active, or the read inhibit from the BBFC is active, RIN(L). If either of these signals is active then a read request will be delayed until it is removed.

Once a request has been latched in it can produce either a synchronous or asynchronous read cycle;

1/ **Asynchronous cycle**: this type of read cycle will only happen if the SG and PF have been stopped, either by a buffer full condition or when the SG is waiting for a new seed. If this happens then RUNNING(H) is brought low by T26, the last timing pulse of the TCU, (see (6) of figure 3.16 for circuit). RUNNING(H) is then brought high again on the second pulse T2 of the first cycle immediately after the SG/PF restarts. RUNNING(H) is synchronised with the inverted 16 MHz clock by (5), fig. 3.15, and then used to hold (2) reset. Thus only if the SG/PF have stopped, RUNNING(H) low, will an asynchronous grant, ASGRNT, be generated lasting 62.5 ns.

2/ **Synchronous cycle**: If RUNNING is active then the synchronous read request signal, SRRQST from (1) of fig. 3.15, will generate a

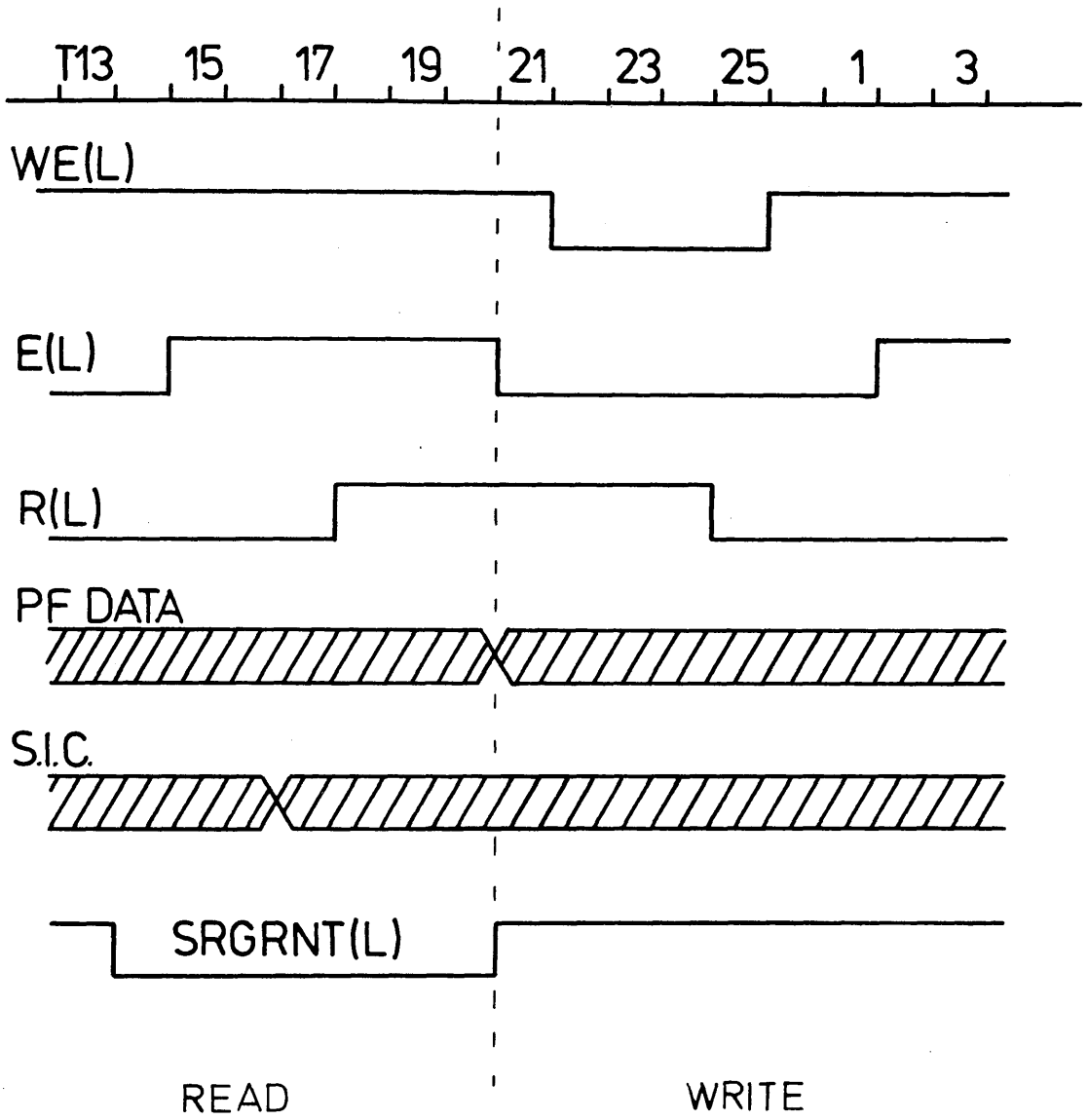


FIG. 3.17 MFG BUFFER TIMING

synchronous read grant signal, SRGRNT, via (3) and (4) of fig. 3.16. This SRRQST(L) signal is completely asynchronous with the MFG system at this point and so is synchronised to the MFG clock by the two flip-flops (3) and (4). It is also synchronised to the MFG buffer read phase by (3). Note that the read phase starts at the beginning of T25 (7) with R(L) going low, but the synchronous read grant does not start until two clock cycles later at the end of T26. This allows time for the R(L) signal to place the BWC, (3) fig. 3.14, into the count down mode before it is clocked by the SRGRNT signal.

It is possible that a read request is first initiated when RUNNING(H) is low and gets as far as bringing the output of (2) high, fig. 3.15, only for RUNNING(H) then to go high again. In this case the asynchronous request would be aborted and then treated as a synchronous request. However if an asynchronous request gets as far as driving ASRGRNT(L) and then RUNNING(H) goes high there is no danger of the request also being treated as synchronous, since the RGRNT(H) signal will clear the read request on (1).

3.5.8 MFG Buffer Write Control

The read and write subcycles of the buffer are split so that a synchronous read is performed in 7 clock cycles (= 62.5 ns at 112 MHz) leaving 6 clock cycles (= 53.5 ns) for a write. Figure 3.16 shows the circuitry to control the write cycle. The PASS(H) signal comes from the PF OEC circuitry, fig. 3.10, and signals that a state has been passed by the PF and so must be written in to the buffer. The E(L) and WE(L) signals are the chip enable and write enable signals respectively for the MFG buffer memories during a write cycle.

Figure 3.17 details the timing for the buffer synchronous read and write cycles. This was also completely revised to accommodate the changes made to the PF timing. The R(L) signal is only used on the BWC, (3) fig. 3.14, to determine whether they count up or down. Since these

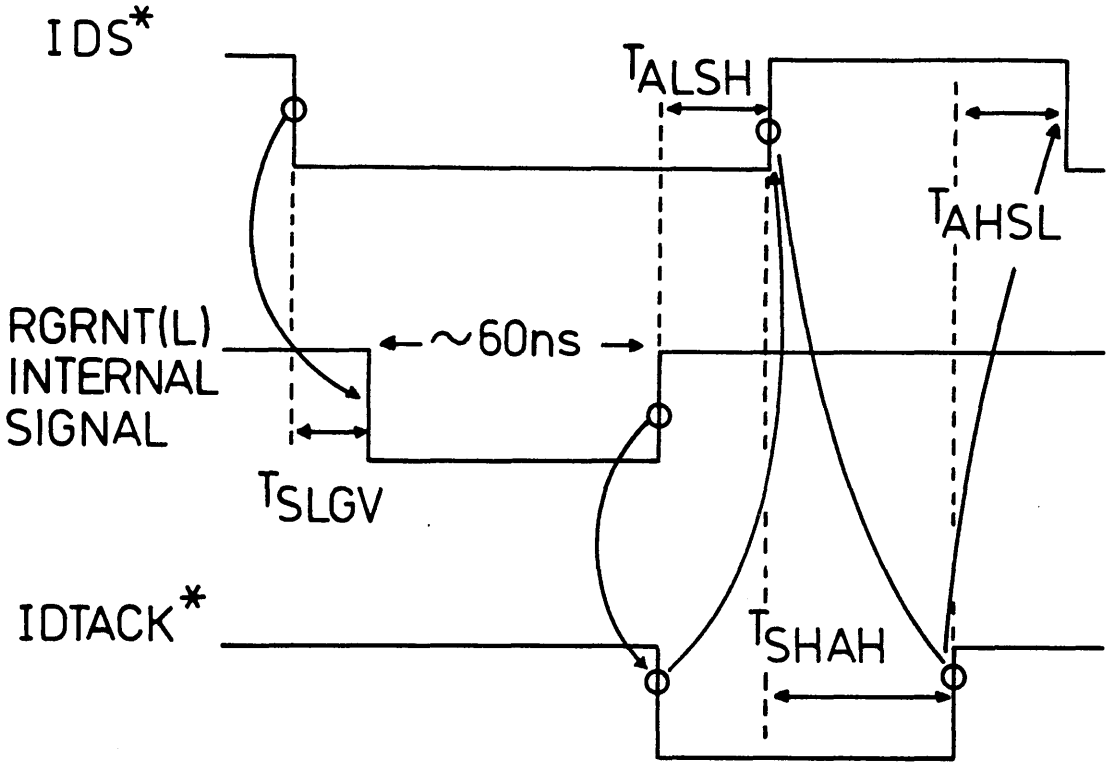


FIG. 3.18 CURRENT I-BUS READ CYCLE.

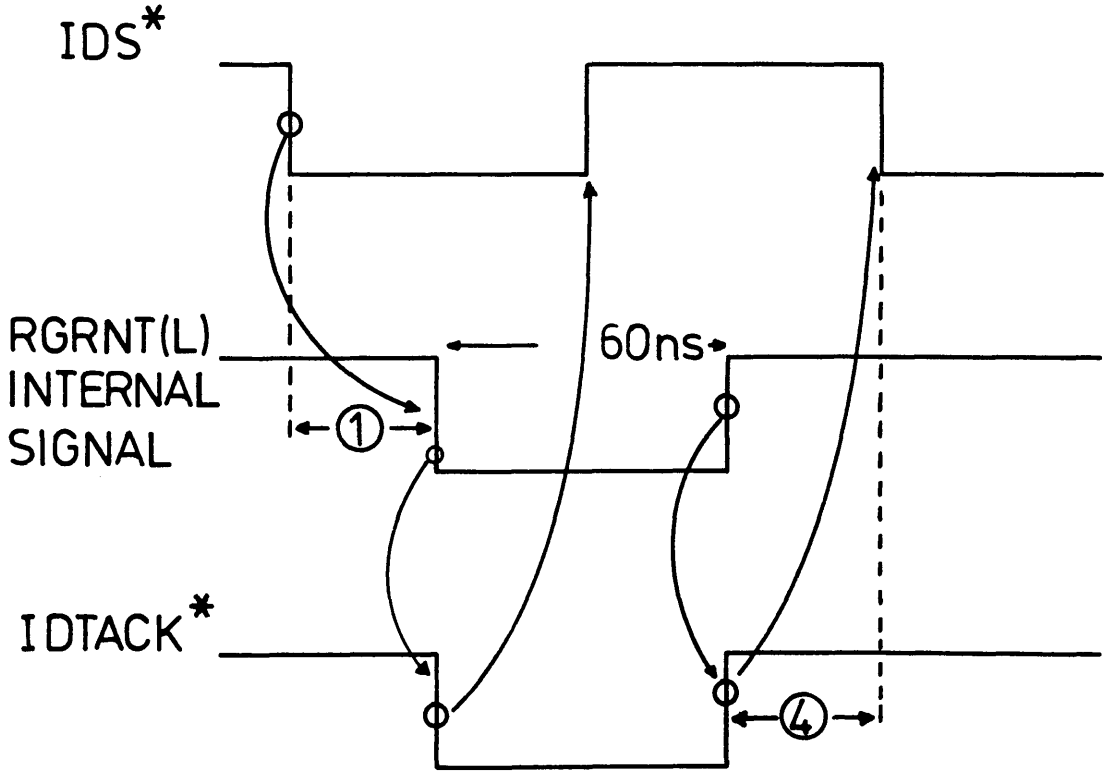


FIG. 3.19 PROPOSED I-BUS READ CYCLE.

			Min	Max	Measured
1	a/ T_{slgv}	(synch)	43.8	77.4	--
	b/	(synch)	159.8	193.4	--
	c/	(asynch)	75.1	96.3	--
	d/	(asynch)	137.6	158.8	--
2/	T_{alsh}		17.5	50.9	25
3/	T_{shah}		8.8	29.3	25
4/	T_{ahsl}		10.9	31.1	15

Table 3.1 I-bus cycle timings

counters are clocked at the start of any read/write cycle then the R(L) signal is changed ahead of the write cycle to give them sufficient setup time.

3.5.9 I-Bus Data Transfer Protocol

I-bus is a dedicated, unidirectional, asynchronous bus capable of supporting only one bus slave, the MFG buffer, and multiple bus masters, the MCMs. The I-bus signal lines fall into two subsets; the ***arbitration bus*** and ***data transfer bus (DTB)***. The arbitration bus requires only four lines; a common bus request line (IBR*), a bus busy line (IBBSY*), a daisy chained bus grant line (IBG*) and a bus grant return line (IBGRTN*). The operation of the bus arbitration protocol will be explained later in Section 4.3. All that need be noted at present is that the arbitration for the next bus master is pipelined with the bus transfer of the current bus master. This pipelining allows minimal delay to be incurred when handing over bus mastership.

The DTB consists of up to 64 data lines of which 42 are used at present. There are only two DTB control lines, IDS* and IDTACK*. These two lines form a simple handshake between the bus master and MFG buffer. Figure 3.18 details the current protocol for the I-bus data transfer, while table 3.1 gives the associated timings. The IDS* line is driven by the current bus master and signals a read request to the buffer as well as indicating to other potential masters that a bus cycle is currently in operation. IDTACK* is the MFG buffers response when the data is valid at its output. In response to the buffer asserting IDTACK* the current bus master will negate his IDS* and latch in the data after a short delay to guarantee set-up times. Only when the buffer has negated IDTACK* does the next bus master assume control by driving IDS*.

Since a read has to be synchronised with the read subcycle of the buffer for both asynchronous (fig. 3.15) and synchronous reads (fig. 3.16), it is more than likely that there will be a delay before this

happens. Time 1a in table 3.1 is the best case delay for T_{slgv} during a synchronous read, i.e. is when the request arrives just in time to be clocked into (4) fig. 3.16. The worst case delay for a synchronous read is where the request just misses the clock and has to wait the full 13 clock period cycle of the MFG before being granted, 1b in table 3.1. 1c and 1d in table 3.1 give the best and worst case timings for the delay imposed on requests being synchronised with the 16 MHz clock during asynchronous reads. Using fig. 3.18 and the figures given in table 3.1 we can arrive at the following bus cycle times for synchronous buffer accesses (allowing 60 ns for memory access);

a) Peak cycle time (worst case),

i.e using 1a for T_{slgv}, and using worst case delays

$= T_{slgv} + 60 + T_{clsh} + T_{shch} + T_{chsl}$ (all max. timings)

$= 77.4 + 60 + 50.9 + 29.3 + 31.1$

$= 248.7$ ns cycle time

$= 4.02$ MHz transfer rate.

b) Peak cycle time (best case),

i.e using 1a for T_{slgv}, and using best case delays

$= 43.8 + 60 + 17.5 + 8.8 + 10.9$

$= 141$ ns cycle time

$= 7.09$ MHz transfer rate.

c) Average cycle time (worst case),

i.e using average of 1a and 1b for T_{slgv}, and using worst case delays

$= 135.4 + 60 + 50.9 + 29.3 + 31.1$

$= 306.7$ ns cycle time

$= 3.26$ MHz transfer rate.

d) Average cycle time (best case),

i.e using average of 1a and 1b for T_{slgv}, and using best case delays

$= 101.8 + 60 + 17.5 + 8.8 + 10.9$

$= 199$ ns cycle time

$= 5.03$ MHz transfer rate.

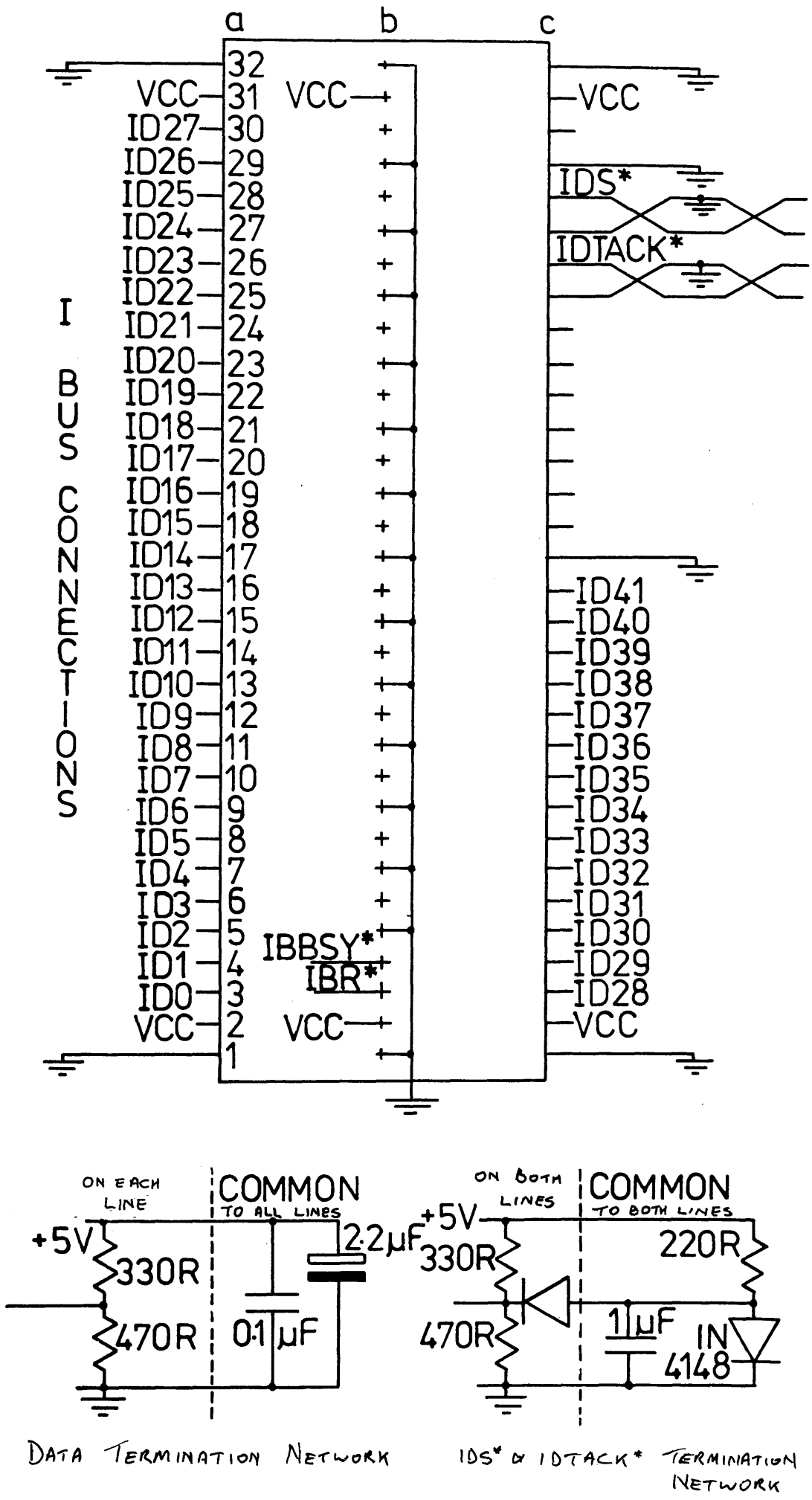


FIG. 3.20 I-BUS CONNECTIONS AND TERMINATIONS.

An average data rate of between 3.26 MHz and 5.03 MHz can therefore be expected on I-bus. While it is possible that the data rate could peak at up to between 4.02 MHz and 7.09 MHz.

However examination of fig. 3.18 shows that time is wasted at the end of each bus cycle in the way IDS* and IDTACK* are removed. Since the TSW is valid at the output of the MFG buffer on the rising edge of the buffer RGRNT(L) signal and stays valid until the rising edge of the next RGRNT(L) pulse it is possible to change the date transfer protocol to that shown in fig. 3.19. As before the current bus master removes his IDS* signal when IDTACK* is driven low and the next bus master is only allowed to assume control when the IDTACK* is negated. However with this method IDTACK* simply follows the internal RGRNT(L) signal and when it is negated it signals to the current bus master that the data is ready. The bus master then latches in the data after a short delay as before. With this protocol the T_{alsh} delay would be buried in the 60 ns memory access time and the T_{shah} time would be lost altogether. The above figures for bus bandwidth would thus become:

a) 168.5 ns = 5.93 MHz,

b) 114.7 ns = 8.71 MHz,

c) 226.5 ns = 4.42 MHz,

d) 172.7 ns = 5.79 MHz.

I-bus bandwidth could therefore be expected to increase to average between 4.42 MHz and 5.79 MHz.

I-bus is physically implemented on a commercially available 21 slot, 96-line multi-layer backplane. 12 of the lines on the backplane are reserved for power and ground lines since they are connected to the power and ground plane of the backplane. The remaining 84 signal tracks on the backplane are layed out with a ground line on either side, thus reducing the impedance of the track and so reducing signal crosstalk. The layout of the signals on the edge connector is shown in figure 3.20. The IBG* and IBGRTN* signals are not physically present on the I-bus

backplane since they require a daisy-chained line which was not available on the backplane used. Instead they physically reside on the C-bus backplane, which being a VME-bus has four daisy-chained signal lines.

The termination circuits used for the data and clock lines are shown on fig. 3.20. These custom-made termination circuits are placed at either end of the backplane on all the data and clock lines. The active pull-up, pull-down termination has an impedance of 194 ohms. This approximately matches the impedance of the signal lines on the backplane and so reduces signal reflection from either end of the backplane. The diodes on the termination for the clock lines help to limit undershoot and so reduce ringing.

3.5.10 MFG Testing and Debugging

During testing and debugging of the MFG hardware it was possible to override some of the normal functions within the MFG using a number of dedicated debug control lines. All these debug control lines are driven by the PG and are under software control. However during some of the early stages of testing they were controlled by switches. While testing circuitry within the MFG signals were monitored via a 2 channel 100 MHz oscilliscope.

For example it is possible to block the BFULL signal from stopping the TCU, via DL1 fig. 3.7, thus allowing uninterrupted operation of the SG and PF. This will of course mean that data is overwritten in the buffer. However it is a very useful facility when signals are being examined within the SG, PF and indeed the buffer write circuitry but when the data in the buffer is not required.

There is also a facility to start the TCU via DL4, fig. 3.8, and thus to initiate cycles in the SG/PF under software control. Using DL2, fig. 3.7, it is possible to block the RESTART signal so that the TCU does not have a pulse injected into it every 13 clock cycles. Thus using

DL2 and DL4 it is possible to run "single shot" full speed cycles within the SG/PF, i.e. only one pulse is allowed to travel through the shift register, thus allowing each major cycle to be initiated under user control. This facility proved useful for single stepping through SG and PF operation and then checking their output as well as checking the state of various key registers and flip-flops after each cycle.

It is also possible using DL3, fig. 3.7, to inhibit the HALT line from stopping the SG at the end of a seed. Thus a single seed is processed repeatedly without PG servicing. This is useful where signals within the MFG are being examined with a 'scope, and so continuous and synchronous operation is required.

The SIC H-mode comparator and BBFC can also be individually disabled, via HMCEN (fig. 3.13) and BBFCEN (fig. 3.14). This facility can be used to simplify control software during debugging.

All of the initial testing of the MFG was performed with much simplified driver software. The software needed only to load the SG channel memories with 2 or 3 different SD-byte chains and then use only a small number of seeds. These seeds and chains were chosen to produce tight loops within the MFG which could then be easily examined and traced. Once the I-bus interfaces were complete a two processor system was implemented, with the PG driving the MFG and an MCM accessing the buffer and checking the data which was read out. When the complete PG software was written more thorough checks could be performed using the same two processor system, but now with the MFG performing the full sequence of events for an SMP iteration.

The MFG hardware has now been completely tested and proven to successfully and reliably operate at a clock speed of 112 MHz.

3.5.11 MFG Performance Limitations

The following have been identified as the major limitations on the performance of the MFG:

1/ **MFG buffer memories**: as has been said the current 55 ns memories limit the MFG major cycle to an absolute minimum of 2 x 55 = 110ns/cycle which gives a clock speed of approximately 118 MHz. Indeed it has been found that the MFG will only operate successfully up to a clock speed of just under 120 MHz. The current memories could be replaced by 25 ns 4k x 4 bit RAMS, e.g. the IDT71682LA (which has separate data input and output lines). This would impose a limit of 50 ns/cycle, which equals a clock speed of 260 MHz.

2/ **SG channel memories**: the time allowed between the clocking of the multiplexers (2) and (3), fig. 3.5, is only 3 clock periods during which time the channel memories must be accessed. The present clock speed only gives 27 ns for this function. This currently does not allow for the maximum address access time of the memories (MCM10144) of 26 ns, plus the maximum propagation delay of (2) and the setup time required for (3), which amounts to another 8.8 ns. However it is within the 17ns typical delay of the memory devices. Replacement with the Fairchild F10414, which has a maximum address access time of 7 ns, or the Motorola MCM10422-7 (which is a 256 x 4 bit RAM) which has the same access time, and replacing the multiplexers with the 10KH equivalent would place a limit of 262 MHz.

3/ **PF Operator Encoder Channels**: analysis of the OECs shows that when producing the second operator index a propagation delay of 30.3 ns (worst case) is required for the signals to travel through the 1-to-32 decoder and the 32-bit register and then be ready at the input of the encoder. Only 4 clock periods are currently given to this stage of the OEC, imposing an upper limit of 132 MHz. The simplest method of increasing this is to replace all the OEC devices with 10KH ECL series parts, which would approximately double this upper limit.

4/ **TCU shift register**: in order to operate above 125 MHz the shift registers in the TCU would have to be changed to 10KH devices. This would place an upper limit of 250 MHz on the clock. However the pulse

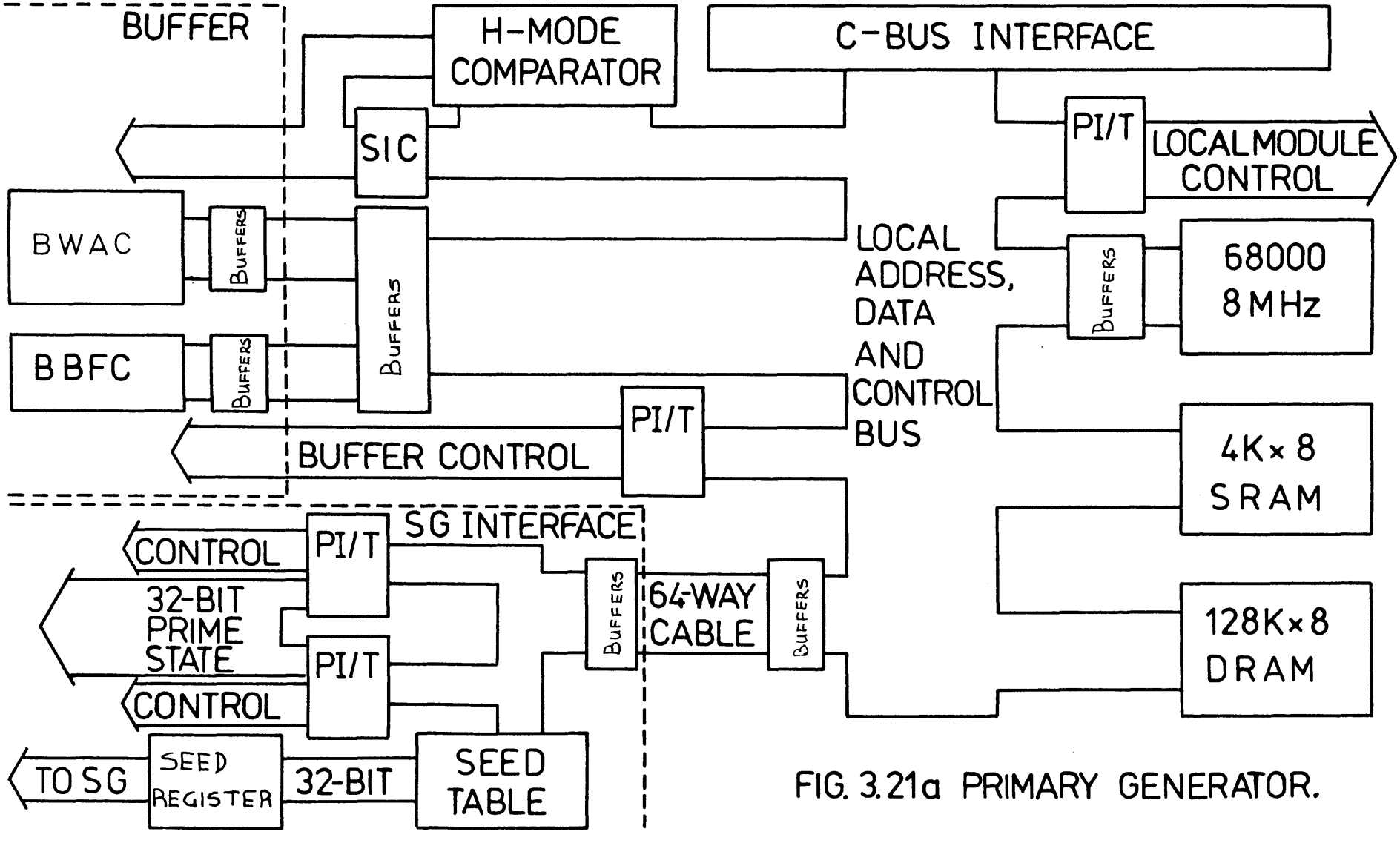


FIG. 3.21a PRIMARY GENERATOR.

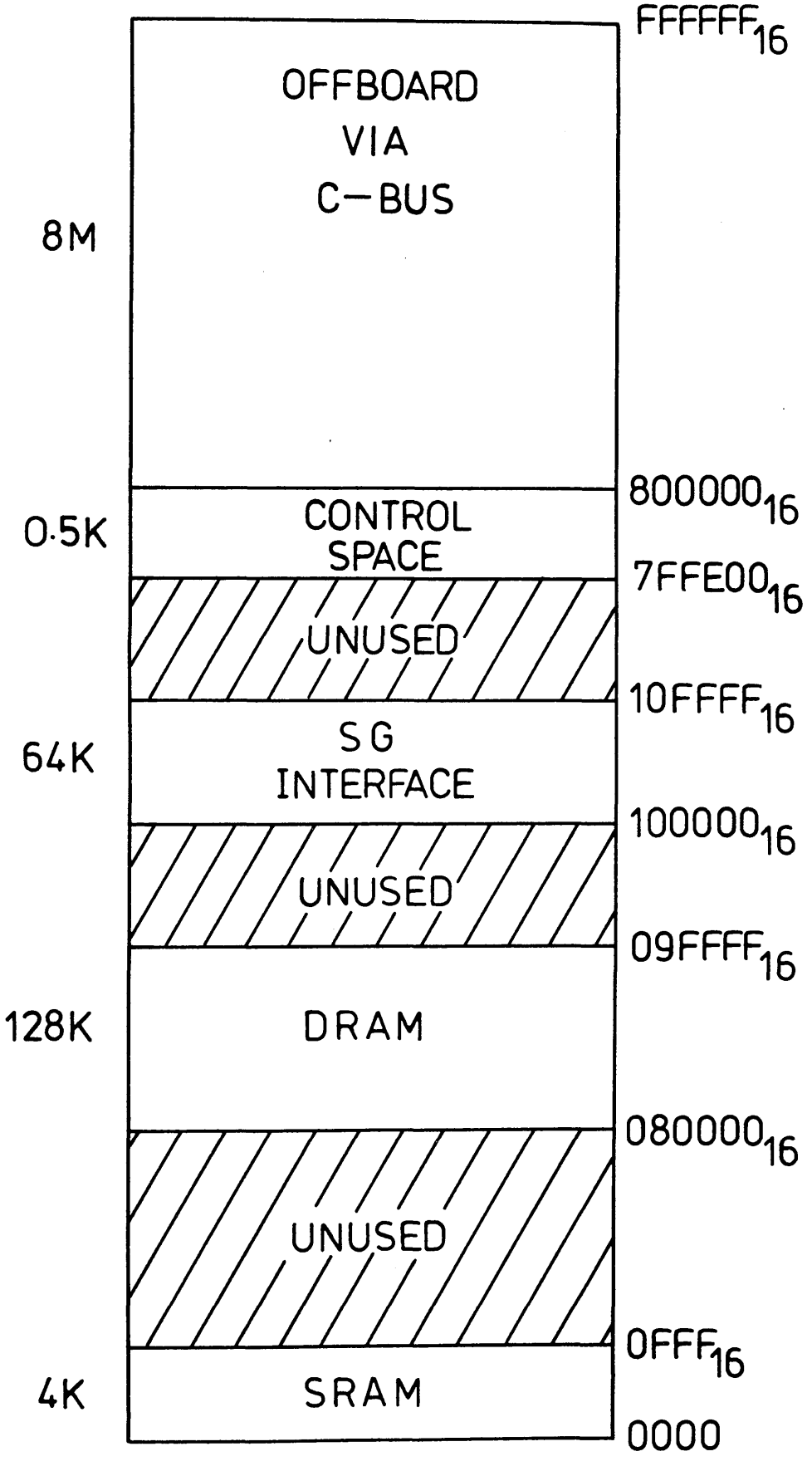


FIG. 3.21b MEMORY MAP

injection logic would become a lot more difficult to control and may not work at this speed.

The above list is by no means exhaustive but it does present some of the more major limitations on the current performance. There are a number of other minor limitations which could possibly be overcome by circuit alterations rather than replacing parts. What is clear however is that with the current design the MFG clock could possibly be increased in speed by a factor of two, at the very most. Beyond this speed the current design could not operate and a major rethink in the design of the MFG would be necessary.

3.6 Primary Generator Hardware

A schematic of the PG hardware is shown in figure 3.21a, with its memory map shown in figure 3.21b. The PG is an MC68000 (8MHz) microcomputer module. It has an interface to C-bus, but none of the other SMP system buses, and has direct control over the SG and PF via the SG interface. The PG also has control over some of the MFG buffer functions as well as access to read the contents of the BWAC. The PG can also write to the preload latches of the BBFC, SIC and H-mode comparator. The buffer functions and all the preload latches along with the two ***PI/Ts (Parallel Interface/Timers)*** are all memory mapped into the ***control space*** (fig. 3.21b).

Only those parts of the PG hardware that are particularly dedicated to its function will be discussed here, and not the more general hardware of its microcomputer architecture.

3.6.1 The SG Interface

The SG interface contains two SG control PI/Ts and an 8k x 8 block of memory, used for holding seed state tables. The SG interface is also the pathway for the PG to initialise the contents of the SG channel memories

and CCM.

The MC68230 PI/T [Mot230] has three 8-bit general purpose bidirectional ports, A, B and C. Each of the bits for the three ports can be independently configured as an input or output pin. The PI/T can be used to generate vectored interrupts to an MC68000 device. Four independent interrupt inputs are provided via the handshake pins H1 - H4. The PI/T will supply a different interrupt vector to the MC68000 depending on which handshake line was the source of the interrupt. All three PI/Ts have their A and B ports in Mode 0, submode 1x, which configures them as bit I/O with the handshake pins as interrupt generating inputs.

The C ports on the two SG control PI/Ts on the interface are used as the ***SG control and status register (SGCSR)***, so that the PG can control certain functions of the SG and also read back its status e.g. idle or running. The A and B ports on the two SG control PI/Ts are combined to form one 32-bit output register, the ***Prime State Register (PSR)***, to hold the current prime state which is fed to the PF. The buffer control PI/T uses its B port as a ***buffer control and status register (BCSR)***.

The 8k block of memory on the interface is used to hold the seed SD-words which are sent to the SG. 2k of these 32-bit words can be held in the memory at any one time. The memory is made up of four 2k x 8 static RAMs, organised as two 2k x 16-bit word blocks. In normal operation the memory acts as any other block in the PG's memory map, being written to and read from 16-bits at a time. However when servicing the SG with a new seed state the PG simply has to read a byte or word of the relevant seed from the memory and the full 32-bit SD-word is read out in one cycle and clocked into the SG seed latch. This utility saves a significant amount of time for the PG servicing the SG and thus reduces the time wasted by the SG while it waits for a new seed state. However this means that a seed state cannot be written into the seed

memory as a contiguous 4-byte word. Instead the most significant 16-bit word of the seed (containing the bytes for channels 0 and 1) is written to the lowest 4k block of the seed memory. The least significant word (containing the bytes for channels 2 and 3) is then written into the same address but with an offset of 4k (bytes) into the highest 4k block of the seed memory.

3.6.2 The Control PI/Ts

A total of 10 lines are used on the two SG control PI/Ts to form the SGCSR. These lines are used as follows;

1/ Input: NSREQ(L) - signals that the SG is requesting a new seed,

2/ Input: IDLE(L) - signals that the SG is idle after completing an M-partition,

3/ Output: REQEN(L) - enables the DREQ(L) signal (see fig. 3.8),

4/ Output: DL4 - a low to high transition on this line injects a pulse into the TCU (see section 3.5.10 and fig. 3.8),

5/ Output: INIT(L) - used to initialise certain flip-flops within the SG/PF,

6/ Output: LOAD_SG - enables the memories in the SG to be loaded prior to running,

7/ Output: LOAD_INT - enables the seed table memories on the SG interface to be loaded and then switched into 32-bit mode,

8/ Output: DL3 - HALT override (see section 3.5.10 and fig. 3.7),

9/ Output: DL2 - single shot enable (" " " "),

10/ Output: DL1 - BFULL override (" " " ").

The buffer control PI/T has 7 lines dedicated on its B port as the BCSR, as follows:

11/ Input: LEMPTY(H) - signals that both the SIC preload latch and the SG seed latch are empty (see section 3.5.5 and fig. 3.13),

12/ Input: BEMPTY(L) - signals that the MFG buffer is empty,

13/ Output: BBFCEN(L) - enables the BBFC (see fig. 3.14),

14/ Output: HMCEN(H) - H-mode comparator enable (see fig. 3.13),

15/ Output: SICLEN(L) - SIC preload enable (see fig. 3.13),

16/ Output: BCLK - this control line is wire-ORed to the BWC clock. A low to high transition clocks the BWC, while it is held low to allow it to run. This line is required to preset the BWC to zero.

17/ Output: BRESET(L) - BWC preset mode line.

Most of the control lines driven by the PI/Ts are completely static during runtime, except lines 7, 13 and 14 which are altered at certain times by the PG software when necessary.

The two interrupts, i.e. the H-mode interrupt and BLKFIN interrupt, are both directed to the PG processor via the buffer control PI/T. This is achieved by connecting the two interrupt signals, SICINT and BLKFIN, to the H1 and H2 lines respectively of the PI/T. The PI/T is then configured by the PG to generate an interrupt to the processor on a negative going edge from either of these signals. Since the two interrupts are both edge triggered the PG must clear them in the PI/T before it will rescind its interrupt signal. The interrupts are both on the second highest interrupt level to the PG processor, i.e. level 6, and so can be masked out if desired.

3.7. Primary Generator Software

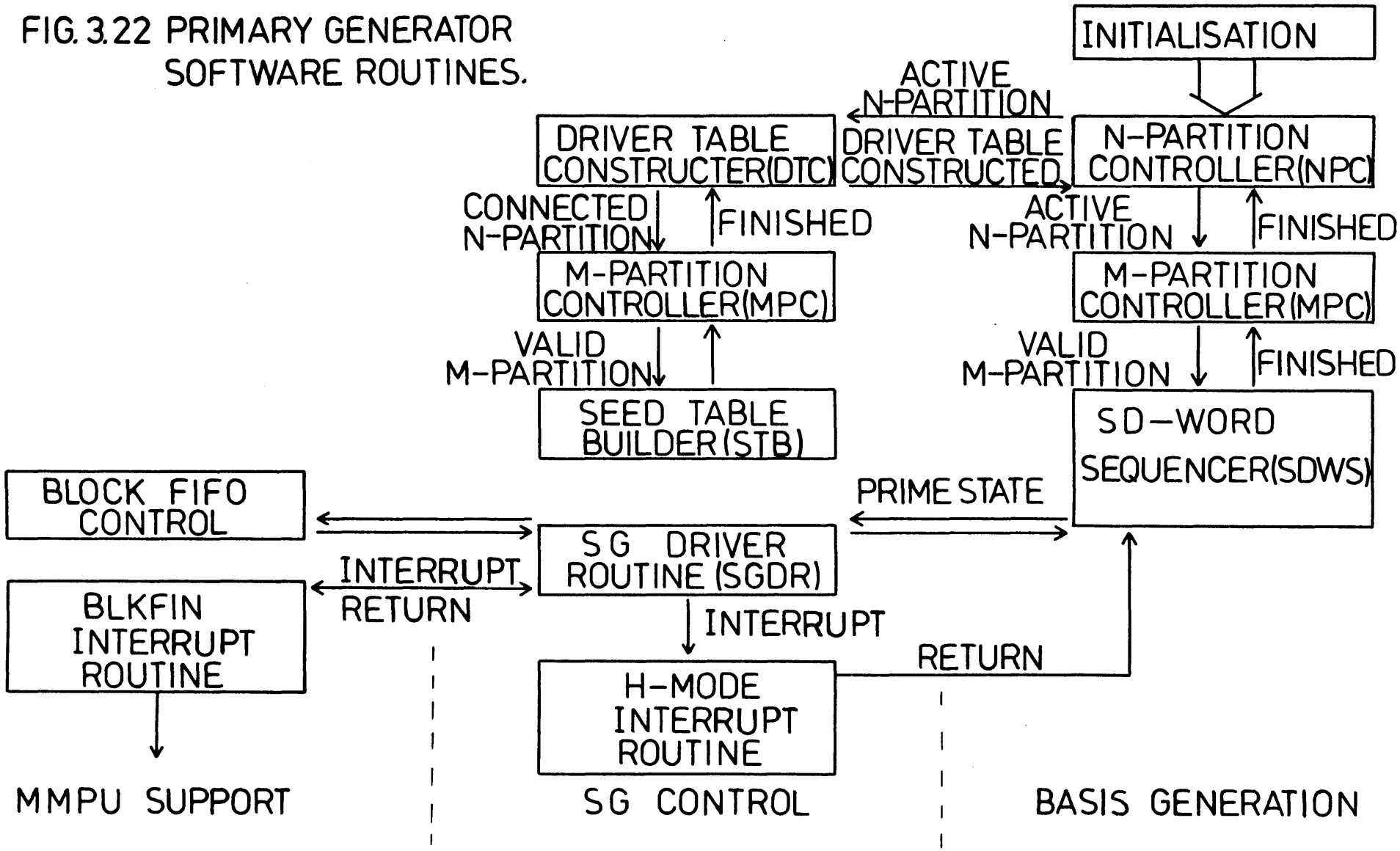
The PG task subdivides into three separate functions;

1/ The Basis Generation Function: this generates, in order, the basis list of SD-words (i.e. prime states) for the nuclei under consideration.

2/ The SG Control Function: amongst other things this function will supply the SG with the necessary seed states, preload the SIC and service the H-mode comparator interrupt.

3/ The MMPU Support Function: this function supplies the MCMs with new

FIG. 3.22 PRIMARY GENERATOR SOFTWARE ROUTINES.



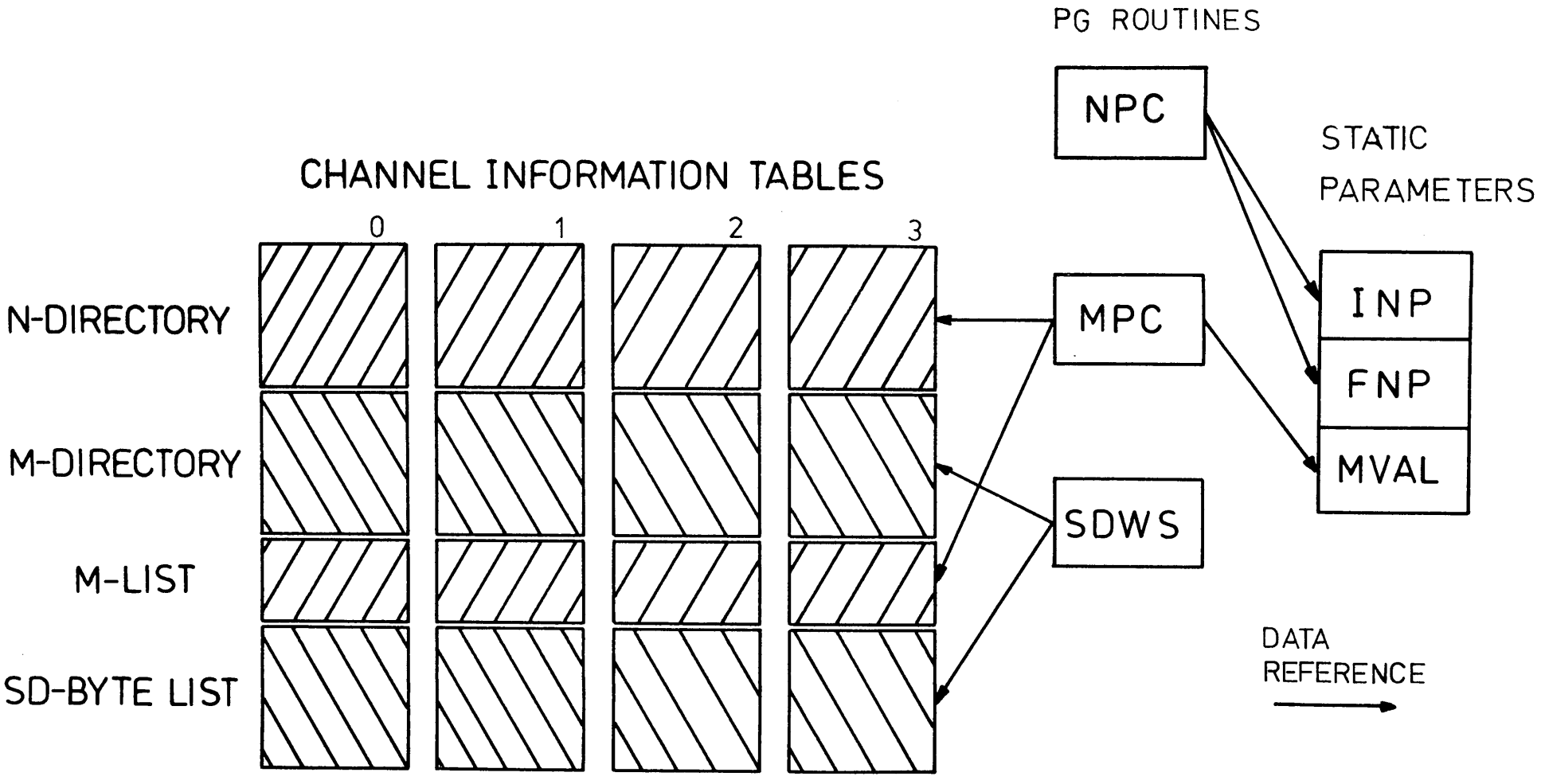


FIG. 3.23 STATIC DATA.

prime state information when necessary.

Figure 3.22 shows the structure of the PG software and the flow of control and data between the different routines. All the software for the PG has been written in Motorola 68000 assembly language. The software was developed on a Motorola EXORmacs 68000 development system using a relocatable assembler and linker package.

To perform its function the PG must generate and maintain a large ***Runtime Data Block (RDB)*** containing lookup tables and PG system parameters. The RDB is split into two sub-blocks; the ***static block*** and the ***dynamic block***. The static block is built prior to runtime and contains read only data. The dynamic block is a read/write block of data which is constantly changing during runtime.

3.7.1 The Runtime Data Block

I/ The Static Data Block

This data block consists of a number of tables and parameters built by the Supervisor Module prior to runtime and then placed in the PG memory. The static block is shown in figure 3.23 along with the PG software routines which reference it. The sole function of the static data block is to aid the PG in generating the basis list of SD-words according to the ordering given in section 3.1. Therefore this data block is only used by the Basis Generation Function.

The static block consists almost entirely of the ***Channel Information Tables (CITs)***. There are four separate CITs corresponding to the four channels of the SG (which in turn correspond to the four SD-bytes that make up an SD-word). Each CIT is itself made up of four different tables, as follows;

1/ The SD-Byte List: this is the lowest level of the CITs. It is a 256 byte table which contains all the possible SD-bytes for the channel it refers to. Within the table the entries are sub-divided into blocks, called ***n-blocks***, with all the entries in an n-block having

the same number of set-bits, i.e. occupied orbitals. There are thus 9 possible n-blocks (for 0 to 8 set bits) in each SD-byte list. The n-blocks are arranged in order, with the n-block corresponding to 0 set bits first in the list. It should be noted that while each of the 4 SD-byte lists of the 4 CITs are split into 9 n-blocks that in some of the CITs some of the n-blocks will be empty. For example under the representation given in figure 3.1 each SD-byte list will have 2 empty n-blocks, since 2 bits in each SD-byte are unused.

Within each n-block the entries are arranged into ***nm-blocks***, where all the entries in an nm-block have the same M-value. The number of nm-blocks in any n-block is variable, depending on the particular n-block and the basis list representation used. The nm-blocks are arranged within the n-blocks in ascending numerical order of their m-values.

Within each nm-block the SD-bytes are arranged in numerical order. All the SD-bytes within an nm-block thus have a constant number of occupied orbitals and M-value. They therefore correspond to the SD-byte chains placed in the SG channel memories (section 3.2).

2/ **The M-list**: This 256 entry table is also ordered into n-blocks with each n-block having a single byte-wide entry for each of its nm-blocks. The entry for each nm-block simply gives the M-value for that nm-block. These entries within each of the n-blocks are arranged in numerical order, i.e. the same ordering as the nm-blocks in the SD-byte lists.

3/ **The M-directory**: The M-directory is organised in exactly the same way as the M-list. However the entry for each nm-block consists of two 2-byte elements and therefore the M-directory takes up 1024 bytes. The first element of each entry in the directory is a 16-bit address offset from the base of the SD-byte list to the base of the associated nm-block. The second element of each entry gives the number of SD-bytes minus 1 contained within the associated nm-block

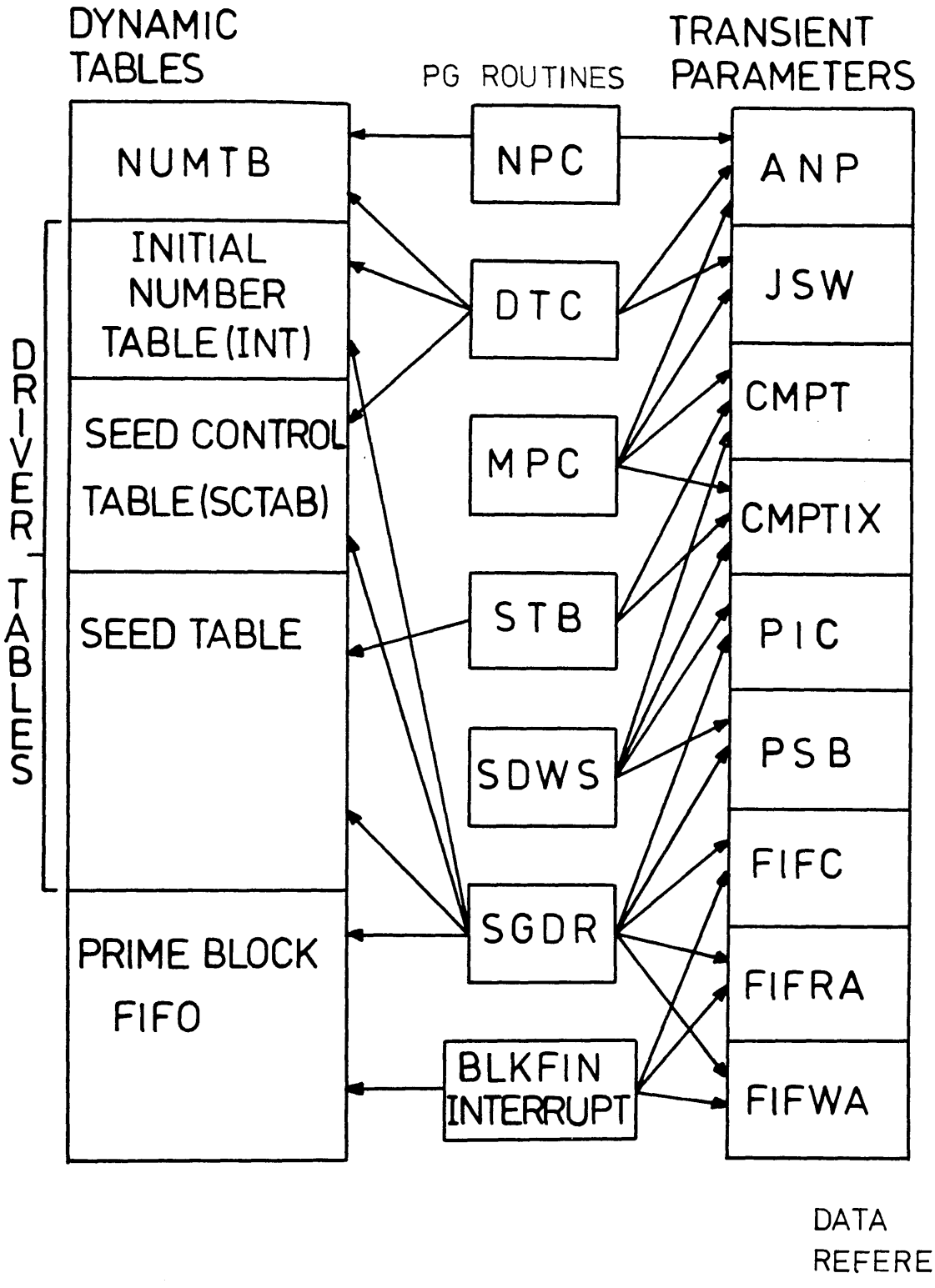


FIG. 3.24 DYNAMIC DATA

of the SD-byte list. (Note that here as in other tables block lengths are stored less one to optimise the use of the 68000 microprocessor assembly language. The decrement and branch conditional instruction (DBcc), which is used to operate most loops, exits a loop when the counter reaches -1. Therefore it is more efficient to store the loop counts less one, rather than to calculate this value).

4/ The N-directory: This is the highest level table within each CIT and is used only by the MPC. It contains 9 different entries, one for each n-block, consisting of two 16-bit word elements. The first element is a 16-bit address offset to the base of an n-block within the M-directory. This offset is used for the M-List as well but since the entries in the M-List are a quarter of the size of those within the M-Directory then it must be divided by 4 (i.e. shifted right 2 places) before it can be used to reference the M-List. The second element is a block length number, which gives the number of entries (i.e. the number of nm-blocks) minus one within the associated n-block. The total size of each N-directory is thus 9 x 2 x 2 = 36 bytes.

The only other entries within the static block are three parameters which define the particular nucleus under consideration, these are *INP*, *FNP* and *MVAL*. INP and FNP are the initial and final N-partitions respectively for the nucleus (an N-partition is specified by a 4 byte number, with byte 0 containing the value of n(P1), etc. (eqn. 3.3)). MVAL is a 2 byte parameter and is the total M-value for the nucleus.

II/ The Dynamic Block

Most of the space within this block is taken up with the ***dynamic tables*** while the remainder is used by ***transient parameters***, figure 3.24. Only the SG Control Function and MMPU Support Function use the dynamic tables, while all the functions use the transient parameters.

The dynamic tables are made up as follows;

1/ The Number Table (NUMTB); This table contains the index of all the

N-partitions within the basis list (where the index of a partition, be it an N or M-partition is the index of the first state within the partition). Each entry in the table is made up of two 32-bit long word elements; the first element specifies the actual N-partition and the second gives its index. This table is built by the Basis Generation Function during the first iteration of the process.

2/ The Initial Number Table (INT): The INT contains the indices of all the N-partitions which are connected to the active N-partition (the active N-partition is the one which the current prime state resides in). Its entries are used by the SG Control Function to preload the SIC each time the seed states enter a new N-partition (section 3.5.5). It is built using the information in NUMTB. This holds no problems in H-mode since in this case only connected N-partitions before and including the active N-partition are searched by the SG. However N-partitions which occur after the active N-partition are searched when processing in W-mode and therefore during the first iteration, since the NUMTB will not be complete, their index will be unknown. Therefore a dummy basis generation run must first be carried out in order to build NUMTB, when processing in W-mode.

3/ The Seed Control Table (SCTAB): The first entry in the SCTAB is the number (minus one) of all the non-empty N-partitions connected to the active N-partition. Therefore this entry gives the number of valid entries in the INT. The remaining entries in the SCTAB give the number of M-partitions minus one in each of the connected N-partitions, i.e. the number of seeds minus one for each N-partition.

4/ The Seed Table: This is the actual list of seed SD-words which are sent to the SG. This table is stored in the seed memory of the SG interface.

The last three tables collectively form the ***Driver Tables***. These tables are built and used only by the SG Control Function. The Driver Tables contain all the information necessary for controlling and

supporting the SG during seeding.

5/ The Prime Block FIFO: This is a software FIFO maintained by the MMPU Support Function. It is used to keep a record of previous prime states and the position of their associated TSWs within the MFG buffer. Each entry within the FIFO consists of a prime state SD-word, its index and the address (read from the BWAC) of its first TSW within the MFG buffer. These entries use 4, 4 and 2 bytes respectively.

Before the SG Control Function begins processing a new prime state, the new prime state details are appended to the FIFO. When a new prime block is reached in the MFG buffer, signaled by the BLKFIN interrupt (section 3.5.6), the MMPU Support Function will broadcast the details, taken from the FIFO, to the MCMs. The BBFC is then reinitialised using the details from the next entry in the FIFO.

In order to maintain the Prime Block FIFO there are three data words, *FIFC, FIFRA and FIFWA*, kept in the transient parameter area of the RDB. FIFRA and FIFWA are the offsets from the base of the FIFO to the next position to read from and the next free position to write to respectively. FIFRA (FIFWA) is incremented each time a read (write) is performed, with the addition performed modulo the length of the FIFO. FIFC is used to keep a count of the number of used locations within the FIFO. Thus FIFC is used to determine when the FIFO is full or empty.

Some of the other transient parameters are:

PIC: this is a 32-bit number which is used to keep a count of the index of the prime state.

ANP: this is another 32-bit location which specifies the active N-partition.

CMPT: this specifies the current M-partition. It is a 32-bit word, with byte 0 holding the M-value of SD-byte 0, etc.

JSW: this 4-byte location is the *Job Status Word*. It is used amongst other things by the PG to determine whether it is in H-mode or W-mode.

It is also used to keep a record of the number of iterations which have been performed.

In total the RDB takes up 8936 bytes of the PG's DRAM system, excluding the seed table which is placed in the SG interface memory.

3.7.2 The Basis Generation Function

The details of the three main routines which the PG uses to generate the basis of SD-words are now considered.

i) The N-Partition Controller (NPC);

This routine generates, in order, all the N-partitions for a nucleus, given its initial and final N-partitions, according to the following steps;

1/ On entry to the NPC from the initialisation routine, the PIC is cleared and the active N-partition is set equal to the initial N-partition.

2/ The NUMTB is updated by appending the active N-partition and the contents of the PIC plus 1.

3/ Control is passed to the SG Control Function to generate the new Driver Tables for the active N-partition. If the JSW shows that the job is being performed in W-mode and that it is the first iteration then this step is not taken so that a dummy first iteration can be performed to build NUMTB.

4/ Control is passed to the M-Partition Controller.

5/ If the active N-partition is equal to the final N-partition then the basis has been completely generated and so the NPCs task is finished. Otherwise the next active N-partition is generated according to the order given in eqn. 3.6. Control then returns to step 2.

ii) The M-Partition Controller (MPC);

The MPC is called by both the Basis Generation Function, via the NPC, and also the SG Control Function, via the Driver Table Constructor

(DTC). In both these circumstances its function is exactly the same, namely to generate, in order, all the M-partitions that belong to a given N-partition. The MPC can determine which routine called it by a flag bit in the JSW. When the DTC is first called it sets the flag bit in the JSW and only when it finishes does it clear the flag.

1/ The individual bytes within the ANP (multiplied by 4) are used as offsets into the 4 N-directories.

2/ The entries read in the N-directories then give 4 offsets and block length numbers for the appropriate n-blocks within the M-directories and M-lists. A 4 word parameter, ***CMPTIX***, is initialised using these 4 offsets. CMPTIX is used to hold the offsets from the base of each of the M-directories to the current nm-blocks being used within the n-blocks.

3/ CMPT is formed. This is done by using the offsets (divided by 4) in CMPTIX to fetch the 4 M-values from the M-Lists. These four M-values are then placed in CMPT

4/ The 4 bytes within CMPT are added together. If the result is equal to MVAL then a valid M-partition has been found and so the the STB or the SDWS is called, depending on whether the MPC was called by the DTC or not. Otherwise the MPC proceeds to the next step.

5/ The block length number of the least significant channel, i.e. that which corresponds to SD-byte 3, is decremented by 1. If the result is equal to -1 then there are no more nm-blocks for this channel and so the relevant word of CMPTIX is reset to its initial value which points to the start of the n-block in the M-directory and the above procedure is performed for the next channel up. If the result was not equal to -1 then the nm-block has not been finished and so the relevant word of CMPTIX is incremented to point to the next nm-block and control returns to step 3.

If the most significant channel runs out of nm-blocks then all the possible M-Partitions for the active N-partition have been generated.

The MPC therefore returns to the calling routine.

iii) The SD-Word Sequencer (SDWS):

This routine generates all the SD-words, in order, that belong to a particular M-partition. As each SD-word (prime state) is built the SDWS will call the SGDR, except when processing the dummy first iteration for W-mode. In this case all that is required is that the PIC be incremented for each SD-word made. The SDWS operates as follows;

1/ Using the 4 offsets in CMPTIX to reference the M-directories, 4 offsets and block length numbers are obtained for the relevant chains in the SD-byte lists. A 4 word parameter, *CSDBIX*, is initialised with these 4 offsets. CSDBIX is used to hold the offsets from the base of each of the SD-byte Lists to the SD-bytes being used to form the current prime state.

2/ Using CSDBIX the 4 SD-bytes are fetched and placed in a 4-byte location, *the Prime State Buffer (PSB)*, in the RDB. The PIC is then incremented by one.

3/ The SGDR is called, except if the JSW indicates that it is the first iteration in W-mode.

4/ The next SD-word is then built in the same manner as in step 5 of the MPC and control then passes back to step 2. When all 4 chains are finished then control is returned to the MPC.

3.7.3 The SG Control Function

The software for the Basis Generation Function need know nothing about the MFG hardware since it is completely independent of it. However the same is not true of the SG Control Function since it is intimately involved in the maintenance of the SG, PF and buffer. This function must be optimised for speed since the vast majority of its workload is in seeding the SG and any delay in this can delay the SG. On the other hand little attention need be paid to optimising the Basis Generation Function since it is a much smaller part of the PG's workload, (note

[An(P1)+2 , An(P2)-2 , An(N1) , An(N2)]

[An(P1)+1 , An(P2)-1 , An(N1)+1 , An(N2)-1]

[An(P1)+1 , An(P2)-1 , An(N1) , An(N2)]

[An(P1)+1 , An(P2)-1 , An(N1)-1 , An(N2)+1]

[An(P1) , An(P2) , An(N1)+2 , An(N2)-2]

[An(P1) , An(P2) , An(N1)+1 , An(N2)-1]

[An(P1) , An(P2) , An(N1) , An(N2)]

[An(P1) , An(P2) , An(N1)-1 , An(N2)+1]

[An(P1) , An(P2) , An(N1)-2 , An(N2)+2]

[An(P1)-1 , An(P2)+1 , An(N1)+1 , An(N2)-1]

[An(P1)-1 , An(P2)+1 , An(N1) , An(N2)]

[An(P1)-1 , An(P2)+1 , An(N1)-1 , An(N2)+1]

[An(P1)-2 , An(P2)+2 , An(N1) , An(N2)]

where An(P1) = the number of occupied orbitals for byte P1 of the active N-partition, etc.

Figure 3.25 Connected N-partitions

that the complete Basis Generation Function takes less than 3 seconds to execute for the largest basis list).

i) The Driver Table Constructor (DTC);

The purpose of the DTC is to generate all the N-partitions which are connected to the active N-partition and to construct two of the Driver Tables, the SCTAB and INT. The N-partitions connected to the active N-partition are related as shown in figure 3.25. However for any particular active N-partition not all of the 13 possible connected N-partitions need exist. This would happen if some of the entries shown in fig. 3.25 were less than the initial N-partition or greater than the final N-partition. Also during H-mode processing the DTC need only produce the first 7 of the entries shown in fig. 3.25.

1/ The DTC first sets the flag in the JSW to signal that it is active. The ANP is then copied since it is overwritten by each new connected N-partition before calling the MPC. A count, called *NCOUNT*, of the number of valid, non-empty connected N-partitions is then initialised to -1. When the DTC finishes NCOUNT will be included as the first entry in the SCTAB.

2/ The first connected N-partition is generated and compared to INP. If the initial N-partition is found to be greater than this then control jumps to step 6, otherwise control proceeds to step 3.

3/ A valid connected N-partition has now been identified, however it still remains to be seen if it is non-empty. Another count, called *MCOUNT*, is therefore initialised to -1 to count the number of M-partitions belonging to the current connected N-partition. The connected N-partition is copied into the ANP location to be passed to the MPC.

4/ The MPC is called and each time it finds an M-partition it will call the STB which will increment MCOUNT.

5/ If on return from the MPC MCOUNT is still equal to -1 then the connected N-partition is obviously empty. The N-partition is

therefore not included in the SCTAB and so control advances to the next step. If MCOUNT was not equal -1 then it is placed in the next location of the SCTAB and NUMTB is searched to find the index of the N-partition. When this is found it is placed in the next position of the INT. NCOUNT is then incremented.

6/ The next connected N-partition is then generated and compared with INP. If it is less than INP then the step starts again. Otherwise the N-partition is compared with either FNP or the copy of the original value of ANP, depending on whether the MFG is processing in W or H-mode respectively. If it is less than or equal to the appropriate partition then control jumps back to step 3, otherwise all possibilities have now been tried. In this latter case NCOUNT is placed at the start of the SCTAB, the DTC flag in the JSW is cleared and control is returned to the NPC.

ii) The Seed Table Builder (STB);

Once the MPC has identified an M-partition it passes it to the STB via the offsets in CMPTIX. The STB then uses CMPTIX to look-up the M-directories to get 4 new offsets into the SD-byte Lists. The 4 initial SD-bytes which this gives thus make up the seed state for the M-partition and so are placed in the Seed Table. When this has been done MCOUNT, used by the DTC, is incremented and control is returned to the MPC.

iii) The Secondary Generator Driver Routine (SGDR);

When the SDWS identifies the next SD-word in the basis list (i.e. the next prime state) the SG must then be sent all the relevant seed states which have previously been prepared by the DTC and STB.

1/ The SGDR must first wait until the SG has stopped processing. The SGDR determines this by testing the state of the IDLE bit in the SGCSR. This is done to ensure that spurious results are not obtained when various hardware control registers are changed later.

2/ The Prime Block FIFO Control routine, part of the MMPU Support

Function, is then entered. This has to add the details of the new prime state to the FIFO.

3/ If in H-mode the index of the new prime is written to the latches feeding the H-mode comparator and the H-mode comparator interrupt is enabled.

4/ The prime state is copied from the PSB to the Prime State Register in the SG interface PI/Ts. The SIC is then initialised with the index, taken from INT, of the first connected N-partition.

5/ The SG can now have the seed SD-words sent to it. The time taken for the PG to send the seeds to the SG is very important to the performance of the MFG, since if this time is too long then it could produce unacceptable delays while the SG waits. As has already been said a full 32-bit seed word can be latched into the seed register in one MC68000 bus cycle. The code required by the SGDR to service the SG is as follows;

```
LOOP   TST.W     (A3)          1 us
       BMI.S     LOOP          1 us
       MOVE.W    (A1)+,D2      1 us
       DBF       D1,LOOP       1.25 us
```

giving a total of 4.25 us. However in this time the SG can produce up to 36 states.

The first two lines test the SGSCR to see if a new seed is required by the SG and if it is not then the test is repeated until one is requested. Line 3 reads the seed from the seed table memory causing the hardware to latch the 32-bit word into the seed register. Line 4 then decrements the counter for the number of seeds in the current connected N-partition being searched. When the count reaches -1 then a new connected N-partition is entered and so the SIC must be reinitialised, and step 5 is repeated.

6/ When all the connected N-partitions are finished the SGDR disables the H-mode interrupt.

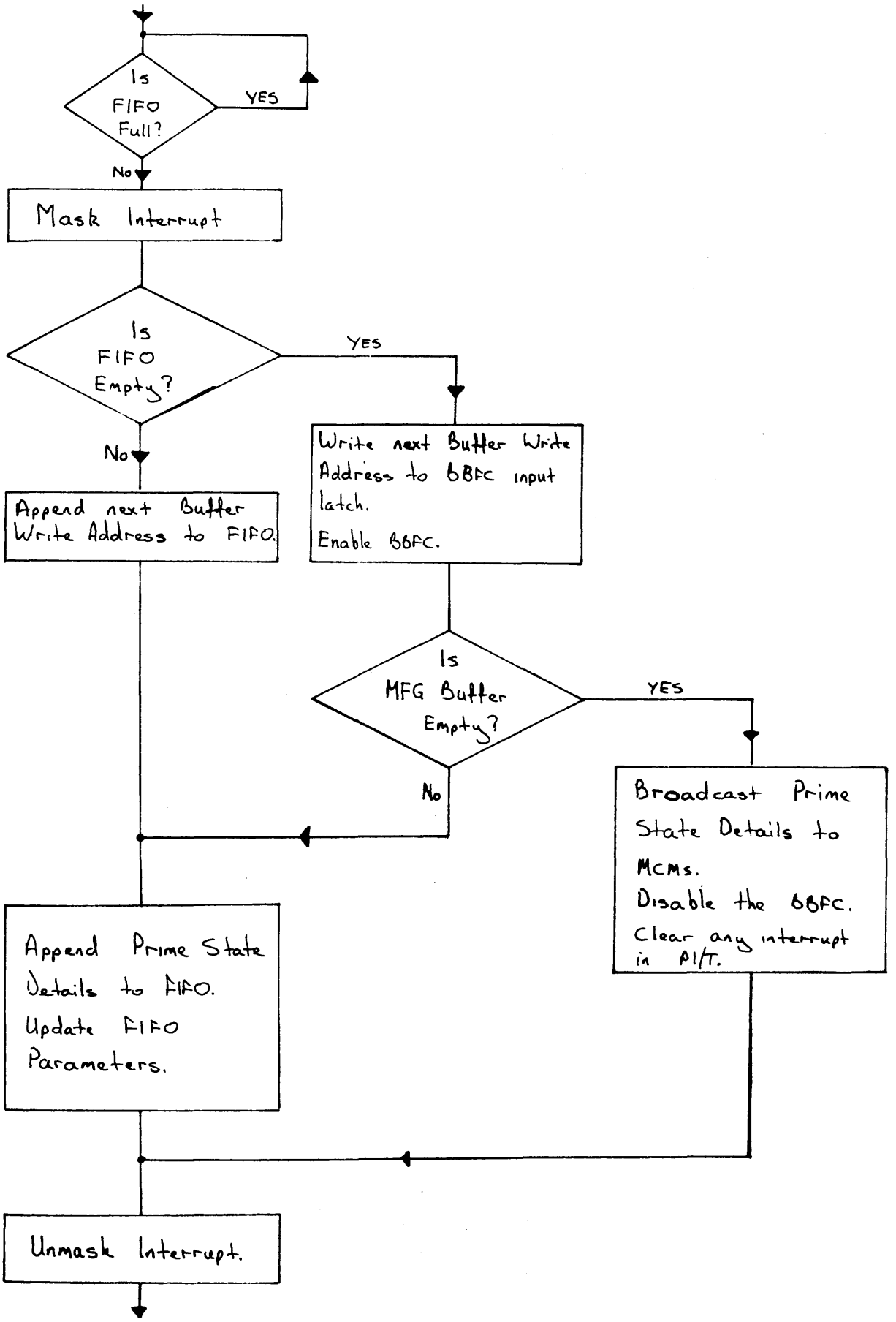


FIGURE 3.26 Prime Block FIFO Update Routine

7/ The SGDR then terminates and returns to the SDWS.

iv) The H-mode Interrupt Service Routine:

This interrupt, generated by the SIC and H-mode comparator, occurs only in H-mode when the diagonal element of the matrix has been produced. Its purpose is to cause the SGDR to abort its task of seeding the SG for the current prime state and for the PG to return to the Basis Generation Function to select the next prime state. The interrupt routine therefore has only two main steps:

1/ The first step is simply to clear the interrupt in the PI/T which directed the interrupt at the PG processor.

2/ Program control must now return to the SGDR but not to the point at which it was interrupted. Instead control must be returned to the end of the SGDR (step 7) so that no more seeds are sent to SG. In order to do this the return address on the processor system stack is overwritten with the address of the relevant part of the SGDR. The interrupt routine then simply executes the normal return from exception (RTE) instruction.

The H-mode interrupt is disabled in step 6 of the SGDR for two reasons. The first is that it is simply no longer required if the SGDR has naturally run out of seeds for the SG. The second is due to the last step of the H-mode interrupt service routine. Since if the PG was interrupted outside of the SGDR then it would return to the wrong routine and cause a fatal error.

3.7.4 The MMPU Support Function

i) The Prime Block FIFO Update Routine:

The two software routines of the MMPU Support Function both manipulate and alter the BBFC and the Prime Block FIFO and its associated parameters. Since the second routine is entered by a BLKFIN interrupt, which in principle can occur at any time, then great caution must be taken by this first routine. A flow diagram, figure 3.26, is used to aid

in understanding the flow of control for this routine.

The BWAC in the MFG buffer can be safely read at the start of this routine since the SGDR, which calls it, has previously made sure that the SG and PF have stopped processing. Therefore there will be no more writes to the buffer for the previous prime state.

The routine must obviously determine if the Prime Block FIFO is full. If it is then the processor must wait until a position becomes free. This will only happen when a BLKFIN interrupt occurs during which the interrupt routine will read from the FIFO. A BLKFIN interrupt must eventually occur since the MMPU is continually reading from the MFG buffer. The size of the FIFO is governed solely by software. A size of 100 entries was used so that the FIFO consumed only 1000 bytes of PG memory. It is highly unlikely that this would fill up, since if it did it would imply that the MFG buffer contained blocks of TSWs for more than 100 different prime states, with each block containing less than 21 elements on average. Even if the FIFO did fill up this would imply a back-log of TSWs in the MFG buffer and so holding up the MFG for a while would have little or no affect on system performance.

At this point the PG processor masks out any external interrupts in its status register to ensure that the rest of the routine is free from the BLKFIN interrupts. This has to be done to make certain that only one routine is manipulating the FIFO and the BBFC at any one time, thus ensuring the integrity of all the FIFO parameters. Since the interrupts are only masked out, then any attempted interrupts which do occur will be "saved" until they are unmasked.

If the FIFO is not empty then it is updated. The three entries written to it are; the next write address of the MFG buffer (read from the BWAC), the new prime state SD-word and its index. The FIFWA and FIFC parameters are also incremented.

If the FIFO is empty then the MCMs must currently be processing the TSWs for the previous prime state. In this case the BBFC will currently

be disabled. Therefore the next write address of the MFG buffer is written directly to the BBFC input latches and the BBFC is enabled, via the SGCSR. However it is quite possible that the MFG buffer has now been emptied by the MCMs. If this were the case then the MCMs would be waiting for the new prime state details. Therefore if the MFG buffer is empty then the new prime state details are broadcast to the MCMs and the BBFC is disabled. BLKFIN interrupts are cleared from the PI/T since it is possible that one may have ocurred. If the MFG buffer was not empty then the FIFO is simply updated.

The routine then unmasks all interrupts and returns to the SGDR.

ii) The BLKFIN Interrupt Service Routine:

This routine first clears the interrupt in the PI/T. It then reads the prime state details from the Prime Block FIFO, at the location indicated by FIFRA, and broadcasts them to the MCMs. FIFRA is then incremented to point to the next entry and FIFC is decremented by one. If this indicates that the FIFO is empty then the BBFC is disabled. Otherwise the start address of the next prime block in the MFG buffer is read from the FIFO and written to the BBFC input latches. The routine then terminates.

3.8 Conclusion

The complete details concerning the method of operation of the MFG along with its hardware and software details have now been given. The performance capabilities of the MFG will be summarised later along with those of the MMPU. However first the details of the MMPU and in particular the MCMs will be discussed.

CHAPTER 4

The Multiple Microprocessor Unit

4.0 Introduction

As the MFG searches the Hamiltonian matrix to identify the positions of non-zero elements so the MMPU, in parallel, processes the MFG's output. As has been said the job of processing the MFG's output sub-divides into a large number of asynchronous, non-identical, independent tasks which are dealt with, in parallel, by the MCMs. The prototype MMPU is made up of up to 16 of these MCMs as well as the Supervisor Module and Central Memory.

In designing any multiprocessor system the nature of the communications subnet is just as crucial in defining the characteristics and performance of the system as the nature of the Processing Modules (in our case the MCMs), and the Global Resources (in our case the CM, SM and MFG Buffer). When defining the SMP communications subnet we must take into consideration the requirements of the different modules present within the system. We also place the following additional demands on its capabilities (in order of priority):

1/ High bandwidth; the subnet should be able to cope adequately with the demands of the MCMs to access the Global Resources. Equally it should be able to cope with the needs of the SM to communicate with and control the rest of the system. The subnet should not be a system bottleneck.

2/ Modularity; it should be a simple task to add new MCMs or Global Resources to the system, requiring no changes to either the subnet or

any other part of the MMPU.

3/ Reliability; as far as possible hardware faults on any of the MCMs should not degrade the performance of the subnet or of any of the other MCMs.

High bandwidth is by far the most important requirement since it will determine an upper limit on the MMPU's performance, which will in turn impose an upper limit on the performance of the SMP (just as the MFG does). Indirect interconnection between modules on the subnet (e.g. as in a loop configuration) would tend to reduce all the above capabilities and so a direct connection between all modules is preferred, i.e. for two modules to communicate with each other no other module need take part in the process.

For these reasons the SMP subnet is based on three shared buses as described earlier in section 2.5.3. The advantages of bus structures have been mentioned earlier in section 1.4.3.

Each module within the MMPU is built using at least two, but usually four, double Eurocards, thus providing four 96-way edge-connectors on one side of a module. The modules are housed in a 19 inch card-cage holding four backplanes which supply the modules with power. Two of the backplanes are standard 96-way backplanes while the other two are VME-bus backplanes. Each backplane has 19 slots for connecting with the SMP modules. Together the four backplanes form the basis of the SMP communications subnet.

4.1 Bus Arbitration Protocol

We have already described and discussed certain bus arbitration protocols in section 1.4.3. However for the purposes of the SMP none of these methods is followed exactly, rather a variation on the VME-bus centralised daisy chain arbitration scheme is used on the three SMP buses. The SMP protocol, which uses a ***decentralised daisy chain***, shares

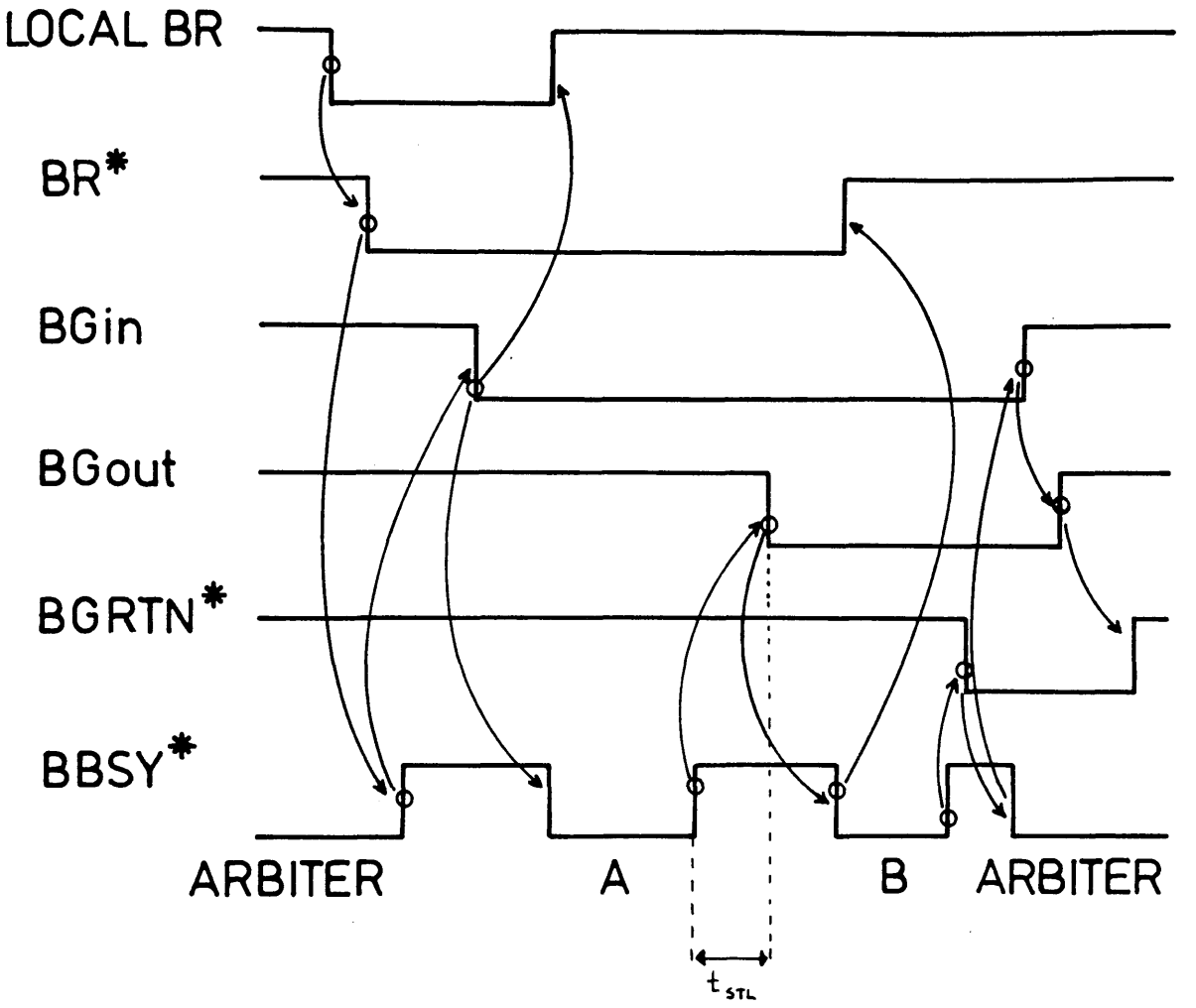


FIG. 4.1 C-BUS ARBITRATION PROTOCOL

the advantages already mentioned of daisy chains. However it overcomes the disadvantages of the centralised daisy chain, where modules competing for use of the DTB have a fixed priority imposed on them by their physical location on the backplane. With the decentralised protocol a round-robin priority arbitration system is implemented thus giving equal access to the DTB for all competing modules.

With the decentralised scheme, as before, when a module requests the use of the DTB it activates the (wire-or) bus request line and the central arbiter then sends a bus grant signal down the daisy chain line. Any module not currently requesting the DTB simply passes the grant signal on. When the bus grant signal arrives at a module which is actively requesting the bus that module will block the grant signal from propagating any further down the daisy chain. Instead the module assumes mastership of the DTB by asserting the bus busy signal, BBSY*. However instead of the grant being rescinded at this point by the arbiter, as in the case of the centralised daisy chain, it is still held active. Then when the current master finishes with the DTB the grant signal is allowed to propagate down the daisy chain to the next module.

When the grant reaches the end of the daisy chain it is fed onto a ***grant return*** line on the bus. The arbiter constantly monitors this signal and only when it is activated does the arbiter negates the bus grant (figure 4.1).

This simple extension to the protocol thus overcomes the rigid, fixed priority of the centralised daisy chain at the expense of only one extra line on the backplane. To implement this decentralisation a few other minor changes, which we will now detail, are made to the protocol.

As has been said any module which is actively requesting the DTB will block the bus grant signal from propagating down the daisy chain. Therefore since a bus master must have the grant signal present throughout its DTB cycle, then all the modules between the arbiter, in slot 1 of the backplane, and the current bus master must be inhibited

from initiating new bus requests. That is any module which is actively propagating the grant signal should have its bus request circuit inhibited. If this condition were not imposed then a module in this position which started to request the bus would find its bus grant in active and therefore inhibit the grant out. This would cause the grant to fail at the input of the current bus master and so remove him prematurely from the DTB and cause a system error.

Similarly if a module starts requesting the DTB just as the grant propagates through its request circuitry, there is the danger that its grant out line may be driven active momentarily. This may give the next module down the daisy chain the impression that a bus grant has been received. In this case both modules could assume mastership of the DTB, again causing either spurious results or at worst a system failure. To prevent this occurring the further condition is imposed that no module is allowed to initiate a new request while the grant is being transferred between modules. This condition is signalled by BBSY* in the inactive state.

However BBSY* would normally be inactive when the arbiter has not issued a bus grant, i.e. when none of the modules are using or requesting the bus. Therefore when this happens the arbiter itself must drive BBSY* active, and so allow new requests to be issued. Also when each bus master releases BBSY* it must wait a time t_{s+1}, (figure 4.1), before propagating the grant along the daisy chain. This delay allows each module to settle its requesting state, i.e. whether it will pass or block the grant, before the grant is propagated.

The SMP decentralised protocol allows overheads introduced due to the arbitration time to be "lost", by pipelining the arbitration with the DTB cycle. This is achieved by making the master negate BBSY* as soon as he actually holds the DTB, i.e as soon as he is actively driving the address strobe, AS*, on the bus. Thus the grant is allowed to propagate down the daisy chain while the bus is being used by the

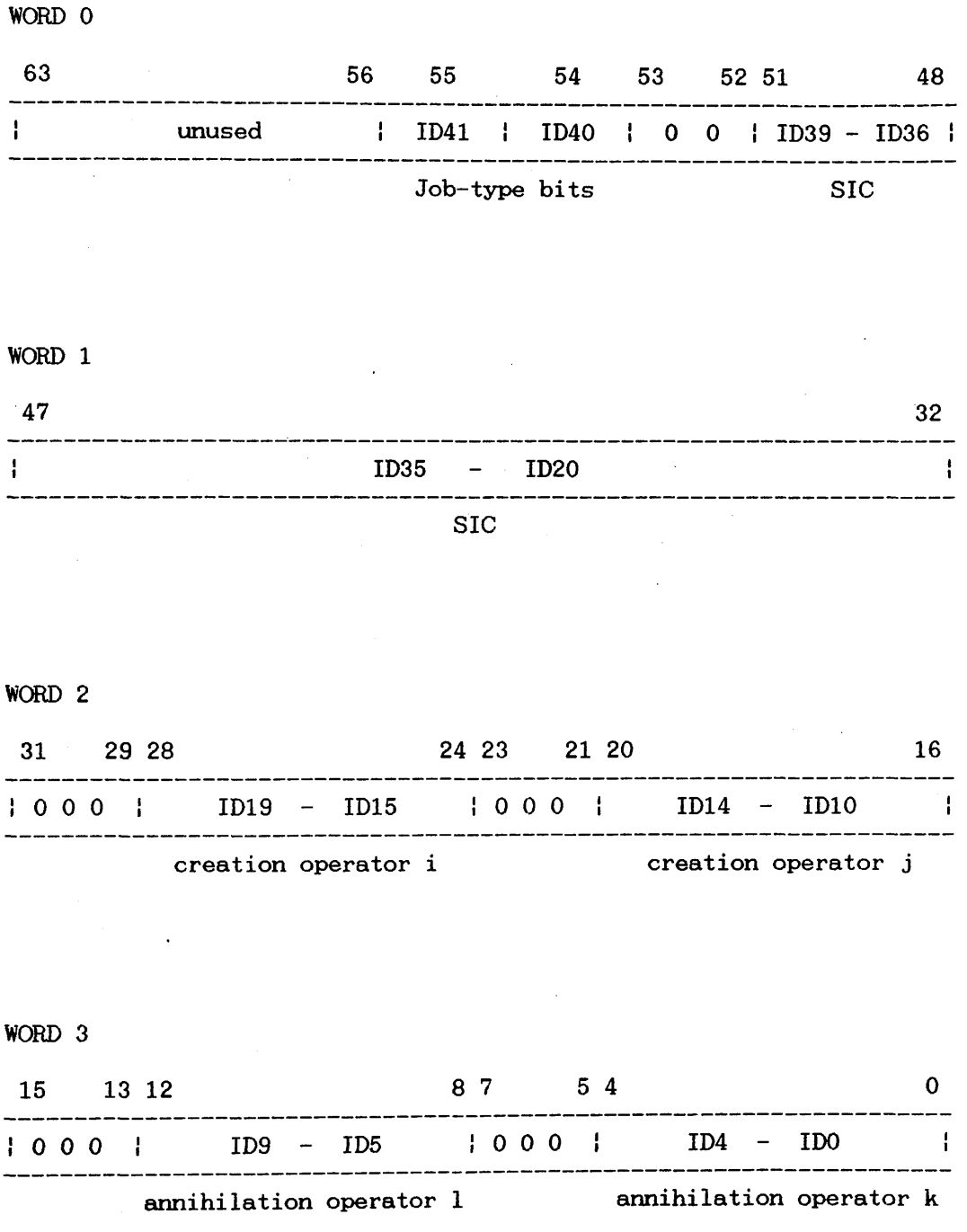


Figure 4.2 I-Bus PFB Word

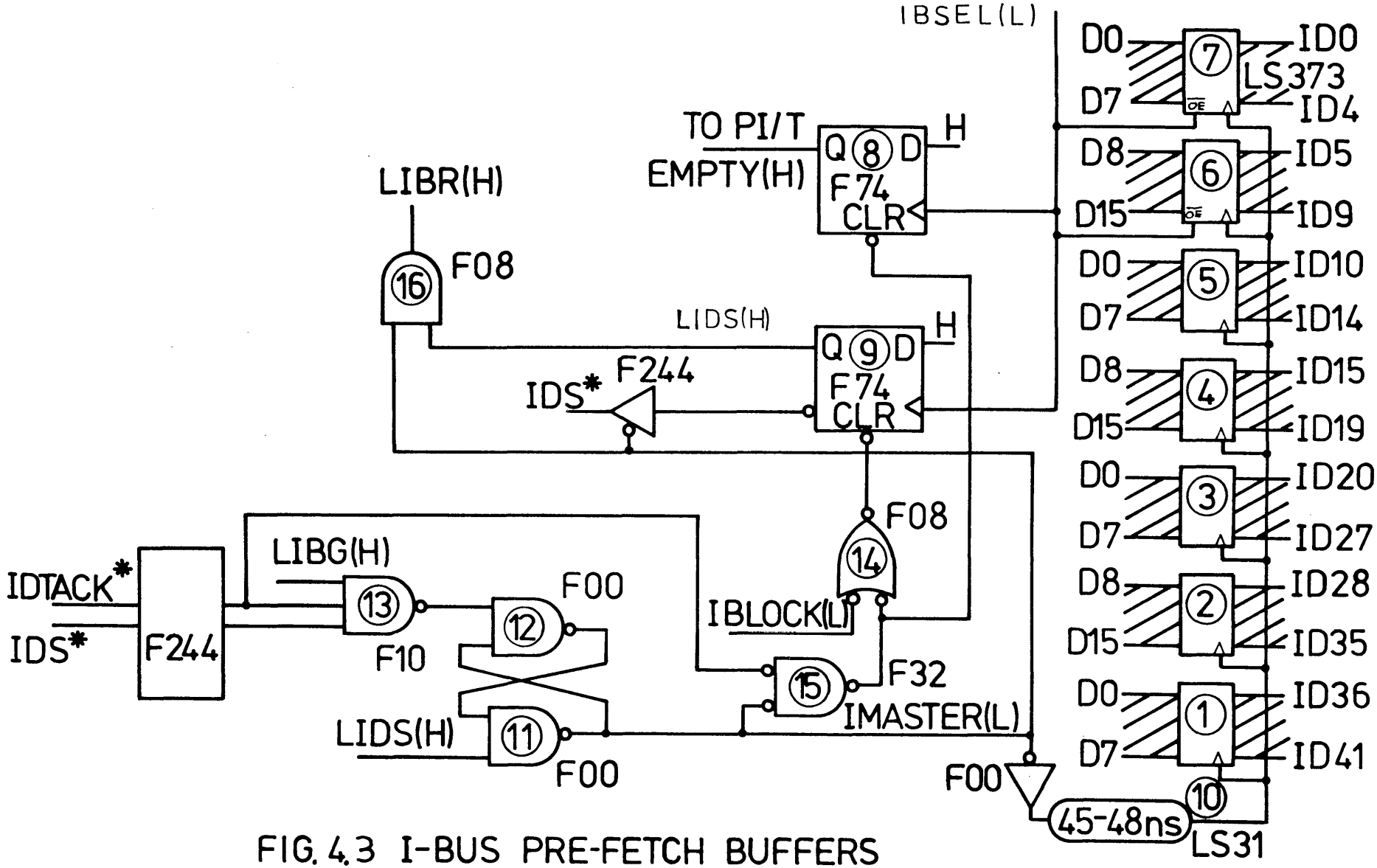


FIG. 4.3 I-BUS PRE-FETCH BUFFERS

current bus master. The time taken to transfer the bus between masters is thus reduced to a minimum. When the requesting module receives the grant he will of course drive BBSY* but will not actually use the DTB until the AS* and DTACK* (the data transfer acknowledge) signals have been negated on the bus. There will thus be a time during each DTB cycle when the module which holds the grant and drives BBSY* will not actually be the one using the DTB, but will in fact be the next module to use the DTB.

We now proceed by giving the details of the hardware implementation for each of the SMP buses.

4.2 I-Bus

We have already given some details of the I-bus data transfer protocol (section 3.5.9). In this section we will give the details of the I-bus interface and bus request and arbitration logic.

4.2.1 MCM/I-Bus Interface

When an MCM reads a TSW from the MFG buffer it is latched into the I-bus ***prefetch buffer (PFB)*** on the MCM. This register is memory mapped into the MCMs address space and appears as an 8 byte location whose format is shown in figure 4.2. Figure 4.3 details the I-bus PFB and its associated control logic. The I-bus data lines (currently 42 are used) are fed to the inputs of seven 8-bit latches, i.e. the PFB. These are latched when the modules I-bus data strobe signal, IDS*, is negated. Latches 1 to 3 form the most significant long word and hold the 20-bit SIC as well as the two job-type bits JT0 and JT1. Latches 4 to 7 form the least significant long word and hold the four 5-bit operator indices I,J,K,L, where I, J are creation operators (I < J) and K, L are annihilation operators (K < L) (figure 4.2).

As the PFB is filled, the D-type flip-flop, 8, is cleared bringing

EMPTY(H) low. The MCM can read the level of this signal via a PI/T and thus knows when the PFB has valid data in it. When the MCM reads the last word of the PFB the IBSEL(L) signal enables the 373s, 6 and 7 which contain word 3 of the TSW. The IBSEL(L) signal is decoded from the MCM processors address and control bus and when it is negated the flip-flop 8 is clocked signalling that the PFB is now empty.

The flip-flop 9 is also clocked by IBSEL(L) to produce a local I-bus request signal, LIDS(H), which is sent to the onboard I-bus request circuitry. When the MCM I-bus control logic receives an I-bus grant, LIBG(H) active, it must wait until the current bus master finishes his cycle, indicated by IDS* and IDTACK* being negated, before assuming control of the bus, signalled by IMASTER(L) active.

The operation of accessing the MFG Buffer via I-bus thus happens completely transparently with respect to the MCM processor. Also each I-bus access is pipelined with the MCM processing the previous I-bus PFB data.

The operation of IDTACK* being activated clears the MCMs IDS* signal and latches the data into the PFB. However to ensure that the data has arrived at the inputs of the PFB before latching, a delay must be introduced in the PFB clock signal. Examination of figure 3.14 shows that the maximum delay between the RGRNT(L) signal going high on the MFG buffer, to the new data being valid at the output of the LS 244's is equal to 50.4 ns. However the time from RGRNT(L) going high to IDTACK* being activated and the PFB being clocked (fig. 3.15 and 4.3) is a minimum of 21.2 ns (excluding the delay introduced by 10). In worst case conditions therefore it is possible for the I-bus PFB to be latched 29.8 ns before the data has arrived at its input (note that the LS 373 needs no set-up time). A delay of 43-48 ns is therefore introduced, using an LS 31, which allows 13.2 ns settle time on the backplane and guarantees I-bus PFB operation under worst case conditions.

New requests to the MFG Buffer can be locked out at any time by the

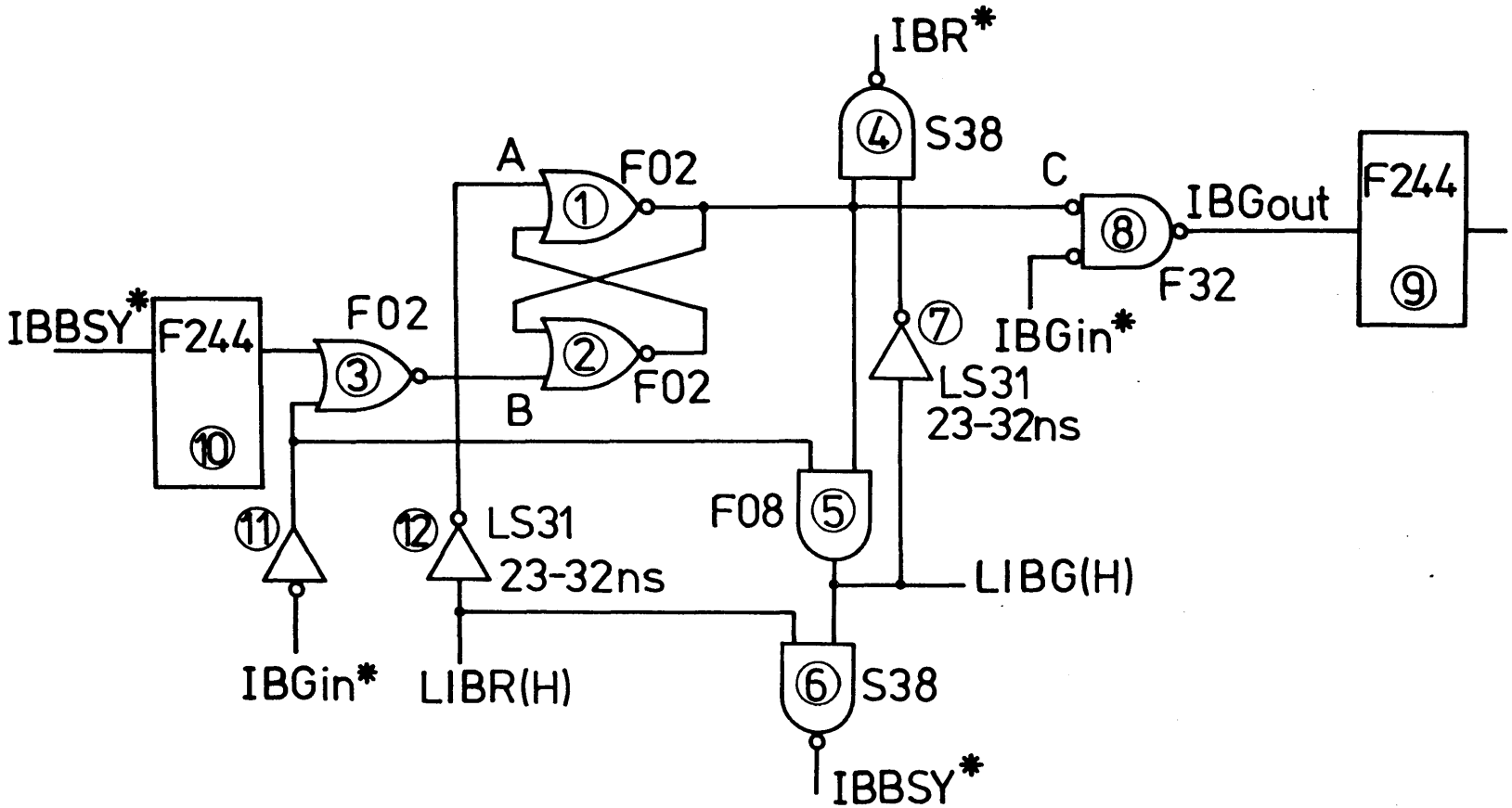


FIG. 4.4 I-BUS REQUESTOR (CORE REQUESTOR)

assertion of IBLOCK(L). This will also act to abort any currently pending or active I-bus requests.

4.2.2 I-Bus Requester

When the I-bus PFB has been emptied by the MCM processor, a local I-bus request, LIBR(H), is generated and passed to the onboard I-bus requester, figure 4.4. The I-bus requester is in fact the core SMP requester for the decentralised daisy chain protocol used on all three SMP buses and therefore its details are extremely important to the whole system.

The local I-bus request signal triggers off the I-bus request logic assuming that the output of 3 is not low, indicating either that the I-bus grant has already passed the module (IBGin* low) or is currently being propagated between modules (IBBSY* high). If neither of these conditions exists then the output of 1 (1 and 2 forming an RS flip-flop) is brought high, activating the I-bus request signal, IBR*. When the bus grant arrives at the module, a local bus grant, LIBG(H), is produced and the module starts to drive the IBBSY* signal.

Once the MCM has actually gained the bus, i.e. the module is actively driving IDS*, it will rescind the LIBR(H) signal and thus stop asserting IBBSY*. The LS 31, 12 figure 4.4, is introduced to produce the delay between negating the IBBSY* signal and propagating the I-bus grant out to the next module. As has been said this ensures that if the next module down the daisy chain gets in a new bus request just as the IBBSY* is negated, then its logic will have settled and be ready to block the grant from propagating any further when it arrives. That is if the inputs A and B of 1 and 2 transition low at the same time with the output of 1 winning and going high, then the input C of 8 will transition high in time to block the grant when it arrives.

To determine the length of the delay which is required we consider two modules, 1 and 2, next to each other on the backplane with module 1

relinquishing mastership and propagating the grant on to module 2. The delay must therefore equal;

propagation delay for IBBSY* to be produced on module 1 and arrive at input B of 2 on module 2

\+ propagation delay for RS flip-flop, on module 2, to transition and bring input C of 8 high blocking grant

\- propagation delay for RS flip-flop, on module 1, to transition and allow grant to propagate, bringing IBGin* on module 2 low.

(For worst case conditions the first two terms will have maximum propagation delays, while the last term will have minimum delays).

This delay is therefore;

$$[\ t_{PLH}(S38) + t_{PLH}(F244) + t_{PHL}(F02)\ ]max$$

$$+\ [\ t_{PLH}(F02)\ ]max$$

$$-\ [\ t_{PHL}(F02) + t_{PHL}(F32) + t_{PHL}(F244)\ ]min$$

$$=17.5\ ns$$

The 23-32 ns delay of the LS 31, therefore guarantees the safe propagation of the grant along the daisy chain.

The propagation delay time of the bus grant through any module is also important since it will determine the length of time any requesting module must wait before the grant reaches it. At present each module imposes a delay of only two gates, an F32 and F244, on the grant. The F244 is considered necessary because of its drive capability which may be required if a termination is needed. The maximum delay these gates will impose is 10.5 ns, but will typically be only 8 ns. Therefore assuming the worst case a module at the end of the backplane would have to wait 157.5 ns (assuming 16 modules) between the grant being produced by the arbiter and reaching the module.

However in most cases there will be more than one module requesting the DTB at a time, in which case the propagation delay of the grant will be pipelined with the current bus cycle. In the last analysis though, the bandwidth of the bus is determined by the bus cycle time and not the

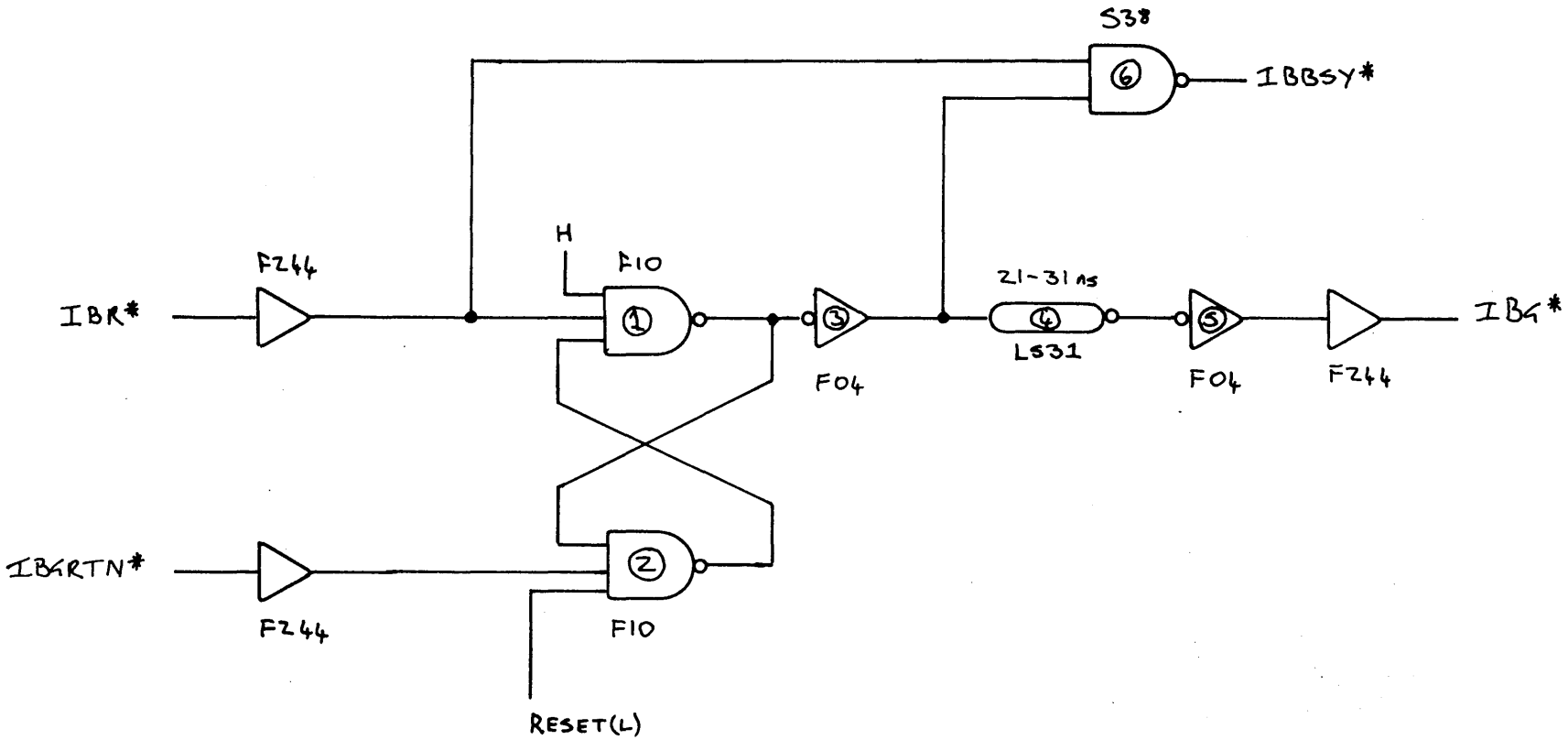


FIGURE 4.5 I-BUS SINGLE LEVEL ARBITER

Bus Requestor Timing Parameters (in nanosecs)

	Min	Typ	Max
LBR(H) high to IBR* low		37.5	46.5
BGin* low to LBG(H) high	5.5	8	10.5
BGin* low to BR* high		41	51.5
BGin* low to BGout* high	5.5	8	10.5
LBR(H) low to BBSY* high		6	10
LBR(H) low to BGout* low	30.5	38	46

Bus Arbiter Timing Parameters (in nanosecs)

	Min	Typ	Max
BR* low to BGout* low	30.5	38.5	46.5
BGRTN* low to BGout* high	32.4	42	50.5

Table 4.1 Bus Request and Arbitration Timing Parameters

daisy chain propagation delay, since this delay is easily made less than the cycle time.

4.2.3 The I-Bus Arbiter

The I-bus arbiter, figure 4.5, is a simple device, again based on an RS flip-flop and is again a core device used elsewhere in the SMP system. There is of course only one I-bus arbiter (whereas there is an I-bus requester on every MCM) which must be located in slot 1 of the backplane in order to drive the daisy chain bus grant line. The I-bus arbiter is therefore placed on the Supervisor Module along with the arbiters for the other buses.

As soon as a bus request arrives, the arbiter releases IBBSY* and then after a 23-31 ns delay drives the bus grant down the daisy chain. The arbiter will then continue to drive the bus grant until the bus grant return line, BGRTN*, is activated signalling that the grant has propagated to the end of the backplane. When this occurs the arbiter will remove the grant signal and assume bus mastership by driving IBBSY* low. At this point new I-bus requests will be enabled on the MCMs.

Table 4.1 gives some relevant timing parameters for the I-bus requester and arbiter. As we can see from this table, the time between a local bus request, LIBR(H), being created and a grant being produced by the arbiter is 93 ns (max) and 76.5 (typ). Therefore the time taken for the last module on the backplane to receive a LIBG(H) from the point at which he activated his LIBR(H) is 93 + 157.5 + 10.5 = 261 ns (max) and 76.5 + 120 + 8 = 204.5 ns (typ).

4.3 C-Bus

C-bus is the command, control and communication bus for the SMP system. As such it is the main path for data (e.g. program code) and message transfers between the MCMs, Supervisor Module and the PG. The Central

Memory will also be interfaced to it, as can any other possible global resources. It is used by the SM to initialise the MCMs and PG; by providing them with their necessary program code, initialising certain tables and parameters in their data blocks and also initialising specific hardware locations. C-bus is also used by the PG to communicate changes in prime state data to the MCMs.

C-bus is significantly different from the other two SMP buses in a number of ways;

1/ There is more than one bus slave interfaced to C-bus and indeed all C-bus masters are potential C-bus slaves and vice-versa. This is not true for either the I-bus or the CMA-bus which have only one bus slave each, namely the MFG Buffer and CM respectively. Also the MCMs which are interfaced to both these buses only ever act as bus masters on them and never bus slaves.

2/ During accesses via C-bus the internal bus of the C-bus master is connected, via buffers, to the C-bus lines. Thus the onboard processor itself controls the C-bus data transfer and not an automatic prefetch buffer as is the case with the other two buses. Thus a C-bus master could potentially access the complete address space of all modules and devices interfaced to C-bus. Since this bestows great power to C-bus masters certain areas are protected so that only a few privileged C-bus masters can access them.

These differences necessitate an expansion of the C-bus structure over that found on the other SMP system buses and also a major change to the nature of the interfaces. For example C-bus must have a means by which bus masters can select the appropriate bus slave that they wish to access. Also a modules C-bus interface must have the flexibility of being able to support the module when it acts as a bus master and a bus slave.

In essence C-bus is a slightly modified VME bus [Fi85, VME82]. C-bus retains the four sub-buses of VME bus, namely;

1/ The Data Transfer bus (DTB); the main bus by which modules transfer data. It contains the address and data lines and associated control signals.

2/ The DTB Arbitration bus; this group contains all the signals necessary to transfer control of the DTB between modules.

3/ Priority Interrupt bus; the means by which modules can interrupt other modules on the bus and request their services.

4/ Utility bus; this includes system clock and reset signals, as well as failure detection signals.

The *functional modules* identified on VME-bus also exist on C-bus, e.g. DTB masters, DTB slaves, DTB requesters, etc. However there are one or two alterations and additions which increase the capabilities of C-bus and make it more suitable for the particular needs of the SMP system.

The most important improvement to the C-bus specification relative to VME bus is the provision of a *bus-broadcast* utility whereby a bus master can write to more than one bus slave per bus cycle. This utility obviously improves the performance of C-bus over VME bus in situations where global data must be transferred to more than one bus slave, which is often the case during shell-model processing. Only the MCMs are potential bus slaves for a broadcast cycle. However each module has the facility whereby it can be locked-out during broadcast cycles. The bus master for a broadcast cycle can therefore be selective about which modules receive the information being broadcast. Only certain key modules, at present the SM and the PG, are able to initiate bus-broadcast transfers since it is obviously a very powerful, and potentially destructive, utility. Similarly only these modules are able to select which MCMs are locked-out during a bus broadcast.

Another addition to the VME bus specification is the alteration of the lowest bus request level, BR0*, to make it conform to the decentralised daisy chain protocol already described. All the MCMs request C-bus on this level and using this protocol, and therefore have

equal access among themselves to C-bus. The other three request levels all follow the VME bus arbitration protocol, thus giving C-bus compatibility with VME bus and therefore allowing standard, "off the shelf" boards to be used on C-bus.

The 6 address modifier lines, AM0-AM5, remain on C-bus as defined in the VME specification. The user defined codes ($10-$1F) which the VME bus specification allows for can be used to identify non-VME type bus cycles, e.g. bus broadcasts, to standard VME modules to prevent them from interfering in these cycles. The interrupt protocol and DTB protocol remain the same on C-bus as on VME bus (although as we have said the DTB protocol is extended to permit bus broadcasts).

4.3.1 C-Bus Lines

As a result of the additions to the VME bus specification, the C-bus DTB structure is different to VME bus in that a number of lines have been added and some redefined. We shall here only describe those lines that are different from those on VME bus.

1/ **MA7-MA0 : the map-select lines**

C-bus is intended, in its final version, to be a full 32-bit address and data bus, as is VME bus. The 8 additional address lines needed to bring C-bus up to this standard are at present named the map-select lines. To explain their function we must first describe how the address space of any processor module is partitioned. At present each MC68000, with its 16M-byte direct addressing range, has its local memory and devices in the lower 8M-byte map, i.e. local address line A23 low. All offboard address spaces are then allocated the upper 8M-byte map, i.e. local A23 high. The top 5 map-select lines, MA7-MA3, are then used to select between the C-bus slaves, allowing a total of 32 different modules to be selected by any C-bus master. A total of 16 8M-byte maps (i.e. a 128M-byte map) can then be addressed within each slave module by the master module, using the remaining 3

map-select lines and A23. This is possible since A23 driven by the MCM processor and A23 on the bus are not the same. The MC68000 must therefore drive MA7-MA0 and A23 on the bus from a latch, e.g. from a PI/T.

In the future when 32-bit processors are used on C-bus modules, address lines A24-A31 will replace the map-select lines. However each module will still be allocated a 128M-byte local map, selected by A0-A26, and have 31 offboard maps, selected by A27-A31. A request by the MCM processor to use C-bus will then be identified by A27-A31 not all low.

2/ BBCST* and BBACK : Bus Broadcast strobe and acknowledge signals

These are the only two extra signals required for the bus broadcast utility. At present the SM and PG are the only two modules with the ability to drive the bus broadcast strobe, BBCST*, and monitor the acknowledge signal, BBACK. All the MCMs monitor BBCST* and drive BBACK. The fact that a bus broadcast cycle is signalled by the dedicated line, BBCST*, rather than say the address modifiers or map-select lines, gives further protection to this utility.

The BBACK line is an active high signal driven by open-collector gates, thus producing a wire-AND. Therefore all bus broadcast slaves must acknowledge the successful transfer of data by driving the BBACK line high before the master can complete his cycle. During a bus broadcast cycle the state of the map-select lines MA7-MA3 is ignored by the MCMs and all MCMs are selected, except those that have previously been locked out of broadcast transfers. MCMs which are locked out of a broadcast cycle will still automatically drive the BBACK signal high.

The bus broadcast facility is of course only intended for write operations. However should a read operation be mistakenly attempted then the MCMs will still be selected but their C-bus buffers will not be enabled. In this case the bus master will terminate his cycle as

normal but read invalid data.

3/ PRIV* : Privileged Module strobe

This strobe identifies those C-bus masters which have privileged access rights and is used in the selection of key C-bus modules. At present only the Supervisor Module either uses this line in its selection decoding or drives it. Therefore the SM cannot be a bus slave to any of the MCMs or the PG.

4/ CONTROL* : Control Map strobe

Associated with the normal address map of each MCM, which contains the local memory and devices, there is also a ***control map***. This map overlays the normal map of each MCM, and is only selected when the CONTROL* line is activated, which can only be done by the SM and PG. Contained within the control map are the devices required to dynamically supervise, control and configure the operation of the MCM e.g. devices to perform processor reset and halt operations, interrupt the processor, control bus broadcast lockout etc. Thus the SM and PG both have the (privileged) option of accessing either the normal map or control map of a particular MCM. It is possible to perform broadcast cycles to the control map.

The MCMs bus for the control map, the ***control bus***, is completely separate from the bus for the normal map, the ***local bus***. This allows accesses to the control devices via the control bus to be carried out without disturbing the MCM processor in any way. This is of course completely different from accesses to the normal map from C-bus, when the MCM processor must be removed from the local bus by its local bus request circuitry and remain idle throughout the access.

At present there is only one device resident within the control map, that is the ***Global Module Controller (GMC)*** PI/T. This device receives/drives the already mentioned control, status and interrupt lines via its I/O ports.

The C-bus DTB arbitration bus has only one addition as follows:

1/ BGRTN0 : Bus grant return (level 0). This is the return line for the bus grant on level 0, the level on which the non-VME, decentralised daisy chain has been implemented.

The interrupt bus and utility bus both remain as defined for VME bus. However at present the ACFAIL* and SYSFAIL* signals are not supported on C-bus.

4.3.2 C-Bus Interface

As we have already seen the C-bus interface on any module must support it in a number of different configurations, namely;

1/ **Isolated mode** : when the module is not selected in any way the local and control buses must be completely isolated from any activity on C-bus. However the map-select lines, bus broadcast line and C-bus address strobe must all be monitored to identify any requests to access either the local or control bus.

2/ **Control mode** : when the module control map is being accessed by the current C-bus master the interface is placed in control mode. In this mode the local bus must remain isolated from C-bus and only the control bus should be connected to C-bus allowing either read or write accesses. In both this mode and isolated mode the local processor is free to access all his local memory and devices.

3/ **Local mode** : in this mode the local bus is connected to C-bus allowing both read and write accesses by the C-bus master to any of the local devices. The local processor is therefore not allowed the use of his local bus and so must remain idle. In both this mode and control mode the module acts as a C-bus slave.

Local mode supports two types of access to the local map, namely ***single cycle*** and ***burst cycle***. For single cycle accesses the local bus is arbitrated for on a cycle-by-cycle basis. For burst cycle accesses the local bus is arbitrated for once and then held for as long as is wanted. This reduces the time taken for block transfers of data to a

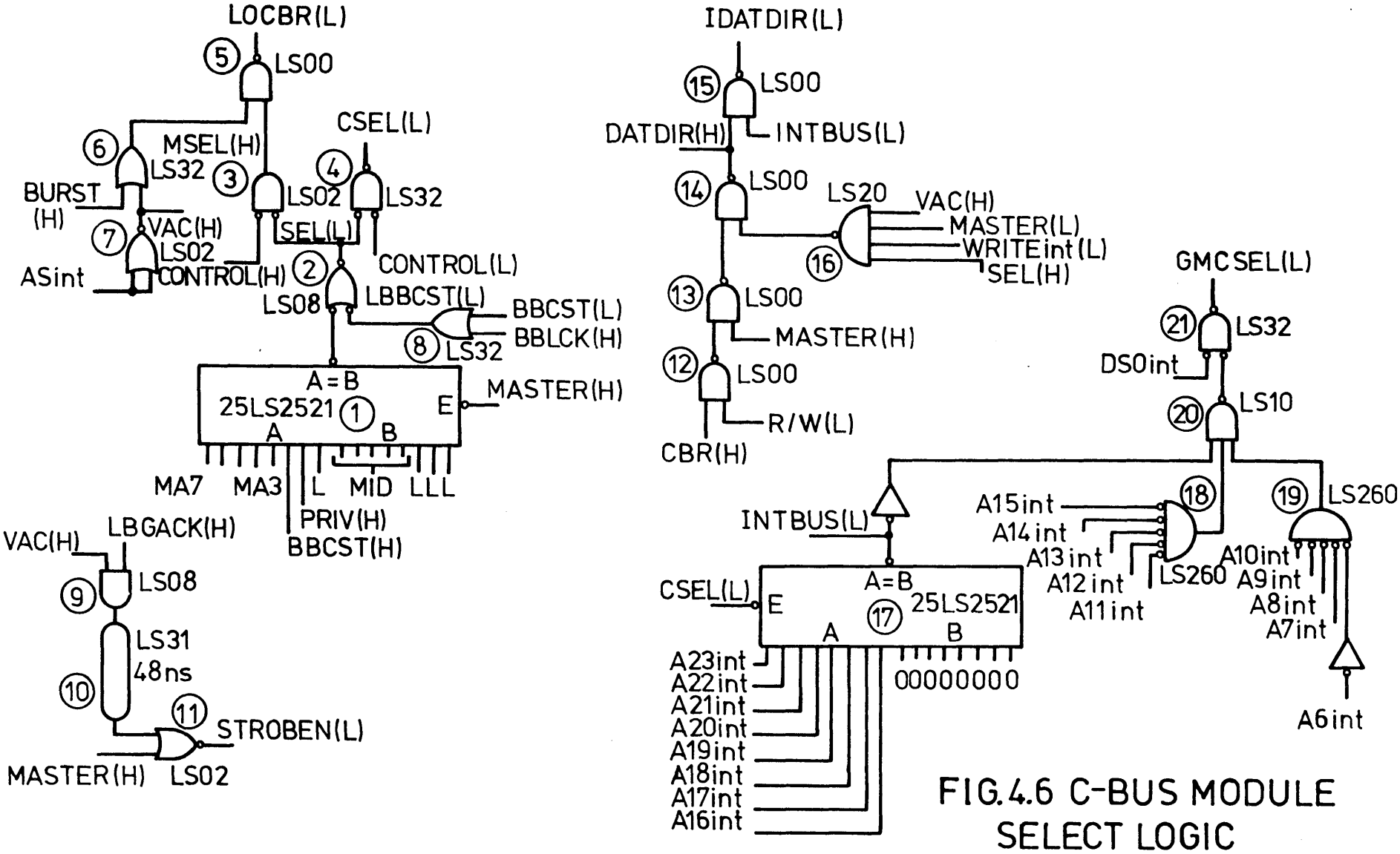


FIG.4.6 C-BUS MODULE SELECT LOGIC

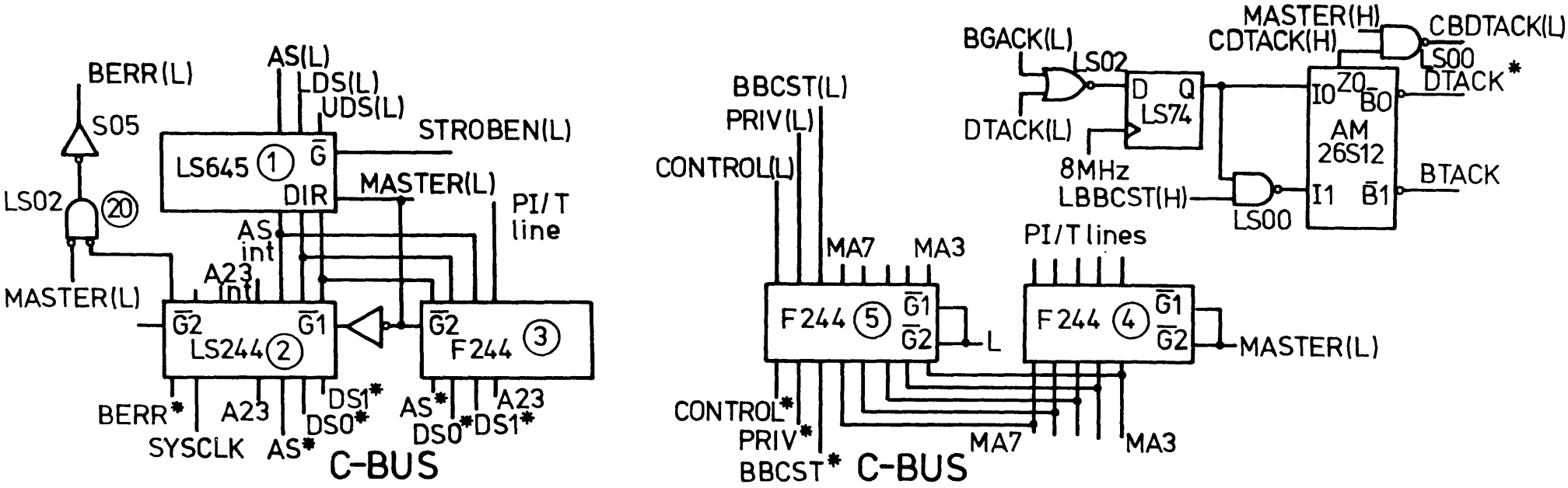


FIG.4.7a C-BUS DTB INTERFACE

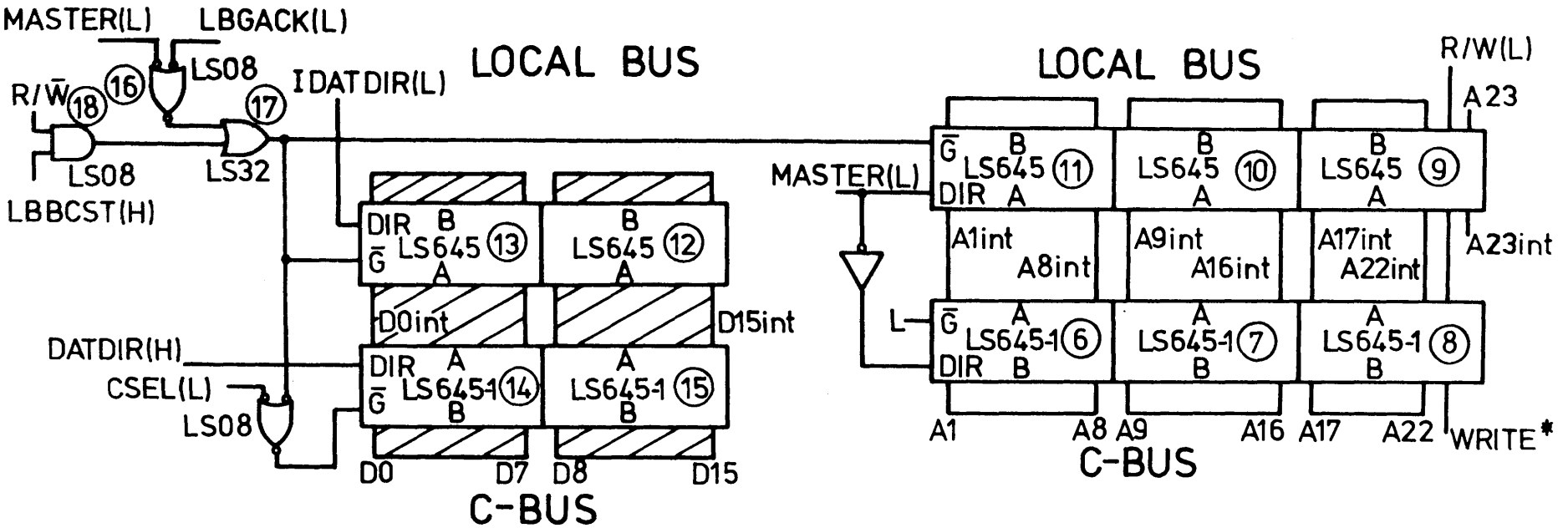


FIG.4.7b C-BUS DTB INTERFACE

module since the C-bus master is not slowed down by the arbitration process for the local bus.

4/ **Master mode** : when the local processor has been granted C-bus mastership the interface enters master mode. The local bus is connected to C-bus allowing read and write accesses to C-bus slaves.

Both the local and control modes support bus-broadcast cycles, although only for write accesses, as well as normal cycles requested via the map-select lines.

Figures 4.6 and 4.7 detail the C-bus module select logic and the DTB interface buffers for the MCMs. There are only slight variations between these circuits and those for other C-bus modules, the differences being mainly in the select logic.

To select a module, indicated by SEL(L) active, the map-select lines MA7-MA3 must match the MCMs ***Module ID (MID)***, which is a unique 5-bit number for every C-bus module. Since no module should ever be allowed to select itself, not that one should ever want to, the MASTER(H) line is used to enable the 8-bit comparator, 1 (the AMD 25LS2521). The MASTER(H) signal indicates, when it is active, that the local processor is currently the C-bus master. The bus broadcast strobe overrides this, since it will select a board regardless of the state of the map-select lines or PRIV line, but only if the local bus broadcast strobe, LBBCST(L), is not locked out by the BBLCK(H) signal.

Once a module is selected, either its control map or its normal map is then selected, CSEL(L) or MSEL(L) active respectively, depending on the state of the CONTROL(L) line. If the normal map is selected then the local bus will be requested from the MCM processor by the LOCBR(L) signal when a valid access is being performed on C-bus. When there is a valid access from C-bus (signalled by VAC(H) active which is produced by AS* active on C-bus) the VAC(H) signal will generate the LOCBR(L) signal. It should be noted that the LOCBR signal can also be activated if the BURST(H) signal is active. This signal places the interface in

burst cycle mode and keeps the local bus of the MCM permanently selected as long as the map select lines match its MID. Both the BBLCK and BURST signals are driven from the GMC and so cannot be changed by the MCM itself.

The double buffering for the DTB interface, figure 4.7b, is necessitated by the two separate buses on each MCM. The control bus is placed between the two sets of buffers, while the local bus is within the inner buffers. Thus accesses to the control bus can be carried out without interfering with the local bus while allowing the module to meet C-bus signal loading requirements, i.e. that there should be no more than one driver and one receiver (or one transceiver) per module connected to a C-bus line.

The map-select lines, address lines and strobes on C-bus are constantly monitored for any requests to access either of the internal buses. If a request for the local bus is made, then once the local processor has granted that request and removed itself from the bus, a local bus grant acknowledge signal, LBGACK, is generated. The inner address buffers and both sets of data buffers are then enabled. The buffer 1 (figure 4.7a), enabled by STROBEN(L), will not be enabled until more than 48 ns after this, in order to provide a setup time for the data and address buses before the data and address strobes are activated on the local bus. This will happen even on block transfers to and from the local bus (BURST active) when the data and address buffers are permanently enabled.

In master mode, signalled by MASTER(L) active, when the local processor is the bus master, the address bus buffers (figure 4.7b) are enabled and turned to drive C-bus. The data bus buffers are also enabled by the same signal and their direction is determined by IDATDIR which is ultimately generated by the processors own read/write signal, R/W(L) figure 4.6. The data transfer acknowledge and bus error signals on C-bus, DTACK* and BERR* respectively, are then monitored by the module to

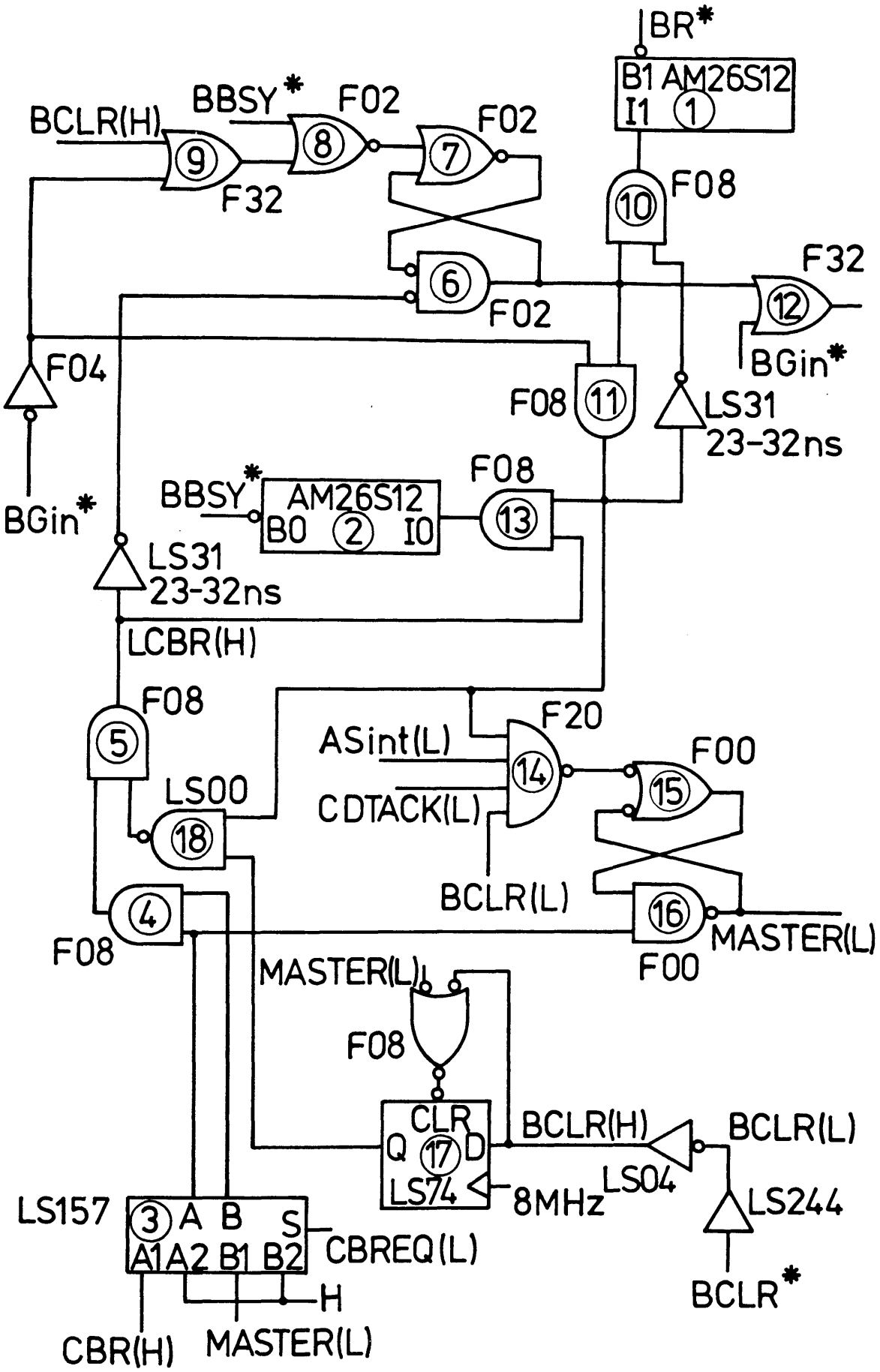


FIG. 4.8 C-BUS REQUESTOR

receive the response from the bus slave. If the module does not correctly select any device or memory on the slave or if a memory parity error occurs then BERR* will be asserted by the slave, generating the BERR(L) signal on the MCM, figure 4.7a. A normal transfer is terminated by the slave asserting DTACK*, which in turn generates the CBDTACK(L) signal on the MCM, figure 4.7a.

In the most severe case where the C-bus master does not correctly select any C-bus module or where the bus slave does not respond at all then a BERR* signal will be generated by the SM C-bus watchdog timer. This timer monitors the AS* on C-bus such that after a (software programmable) delay if the AS* is still active, then BERR* will be generated and the master removed from the bus. The selected delay for the timer should obviously be longer than the maximum response time of all the devices in the system.

As recommended by the VME bus specification the address and data strobes are driven by 64 ma drivers (74F244s) onto C-bus and the remaining three state drivers all have 48 ma drive capacity (74LS645-1). On most of the open-collector drivers the AM26S12 quad bus transceiver is used. This device has four high drive (100 ma), open-collector bus drivers which are connected internally to four bus receivers with hysteresis characteristics (typically 0.6 V threshold margin) [AMD86]. It therefore has superior capabilities to a S38/LS244 combination. The hysteresis is especially necessary on high-speed, open-collector lines due to the "wire-OR glitch" which they tend to produce when used on bus backplanes even when properly terminated [GT83].

4.3.3 C-Bus Requester

The C-bus requester, figure 4.8, is designed around the core requester used on I-bus, figure 4.4, with a few alterations. Its major difference is the addition of the logic to deal with the C-bus ***bus clear*** signal.

A local C-bus request, LCBR(H) active, is initiated in either of

two ways;

1/ The CBR signal. This is produced by the modules decoding logic in response to a processor cycle which requires the use of offboard resources.

2/ Or under software control by the CBREQ(L) signal which is driven by a PI/T line.

The CBR signal thus requests C-bus on a (processor) cycle-by-cycle basis while CBREQ activates a C-bus request as soon as it is driven low. With this latter form of request the MCM will hold C-bus, once it has been granted, until CBREQ is driven high. Large block transfers can thus be carried out over C-bus, using the CBREQ signal without the need for the request/arbitration delay between C-bus cycles. However with this type of transfer the arbitration time is not pipelined with the bus cycle. This is because the local request, LCBR, is not removed and therefore the grant is not propagated, until CBREQ is brought high.

The bus clear signal, BCLR, is activated by the C-bus arbiter in response to a request for C-bus from a module with a higher priority than the module which is currently using the bus. When this happens the lower priority master must terminate its usage and any other requests on lower priority modules must be denied. For the current master there are two choices for the bus clear process; for block transfers the processor is interrupted and CBREQ is removed in the interrupt routine thus releasing C-bus, otherwise for single cycle bus transfers the cycle is allowed to follow through to its normal completion.

The bus clear process is more complicated for modules which are actively requesting the bus and which are in the path of the grant as it propagates down the chain after being released by the previous master. When the grant-in arrives LCBR(H) must be negated immediately, by 18, to allow the grant-out to propagate. However the grant-in must not allow the module to take control of the bus i.e. MASTER must not be asserted. Only once BCLR is released can LCBR be reasserted and the module is

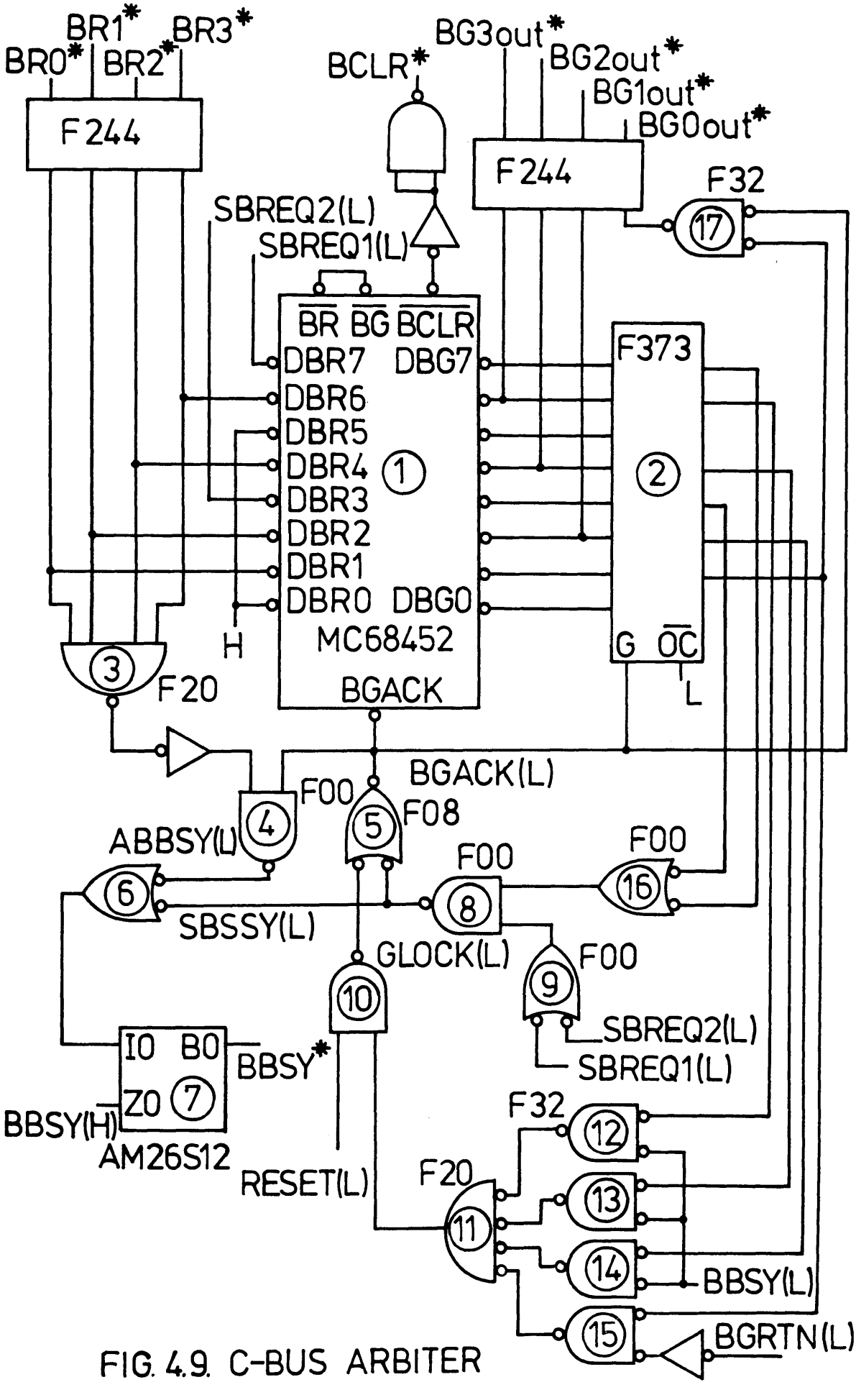


FIG. 4.9. C-BUS ARBITER

allowed to place a new request for the bus. All of this happens completely transparently to the local processor.

The requester shown is used only for the MCMs since it is they that use the decentralised daisy chain protocol. The PG on the other hand uses a higher request level than the MCMs, since it requires a much higher priority usage of C-bus. It therefore follows the VME-bus request and arbitration protocol.

4.3.4 C-Bus Arbiter

The C-bus arbiter (figure 4.9), placed on the Supervisor Module in slot 1, is very different from the simple I-bus arbiter discussed earlier. This is due to the fact that it must cope with the 4 prioritised request levels of C-bus as well as two other levels used solely by the SM. The heart of the arbiter is the Motorola MC68452 ***Bus Arbitration Module (BAM)***, 1 [Mot452]. This is a bipolar asynchronous device which can arbitrate between up to 8 independent prioritised request levels using a protocol along the same lines as VME-bus. The top priority request level, level 7, is given over to the SM which therefore has supreme priority over all other requesters when needed. However the SM also has the option of using level 3 instead and therefore can operate at a reduced priority when its needs are less important.

The C-bus arbiter drives BBSY* itself in two cases: when the SM is using C-bus, signalled by SBBSY(L) active, or when there are no current C-bus requesters or masters, signalled by ABBSY(L) active. Therefore as soon as one of the C-bus request lines is activated the arbiter will stop driving BBSY*. 52 ns after negating BBSY* the BAM will issue a bus grant on the appropriate level, assuming that BGACK(L) is not already asserted (i.e. the bus grant is already issued). In the case of the decentralised protocol, on BR0* of C-bus, BGACK(L) will not be asserted and the bus grant will not be propagated until the bus grant return, BGRTN(L), is rescinded showing that the grant has been driven high all

along the daisy chain.

When BBSY* is driven low by the module which received the grant, GLOCK(L) will be brought low, 10, and so BGACK(L) will be asserted, 5. This signals to the arbiter that successful transfer of the bus has occurred and also latches the state of the bus grant on the output of the F373, 2. BGACK(L) will then remain low either until BBSY* is driven high at the end of the bus cycle (centralised daisy chain on BR1-3*), or until BGRTN(L) is brought low (decentralised daisy chain). For the decentralised daisy chain the delay produced by the BAM between negating BBSY* and driving the bus grant (being greater than 20 ns) is enough to ensure correct operation of the protocol.

Should a new request be initiated when BGACK(L) is low, the BAM will compare its priority with that of the current master which the BAM has latched internally. If this priority is greater than the current masters then BCLR(L) will be asserted otherwise the BAM will wait until BGACK(L) has been negated when it will issue another grant.

The BAM therefore greatly facilitates the arbitration procedure amongst the different request levels on C-bus, requiring little extra logic even to accommodate the decentralised daisy chain.

4.4 Central Memory and CMA-Bus

For calculations within the sd shell the nuclei with the largest basis list would require approximately 800K bytes of storage to hold both the initial and final vectors present during any iteration. It is not inconceivable that this amount of primary memory should be included on each MCM to provide them with their own private copy of the vectors. However this would be a wasteful duplication of data and the Central Memory Module and CMA-bus subsystem will provide an efficient means by which all MCMs can have shared access to these vectors. Dedicated prefetch buffers will operate concurrently with MCM processing and will

thus allow the necessary vector elements to be fetched transparently to the MCM thus giving this method very few overheads. Also at the end of each iteration the completed final vector will be immediately available in CM. On the other hand if each MCM had a local version of the final vector then these would all have to be accumulated together before the true final vector was complete. In a four times the size system capable of performing calculations within the pf shell these vectors would become too large to be stored locally. Therefore in such a system CM and CMA-bus would become vital to the success of shell-model calculations.

It has not yet been possible to implement the CM / CMA-bus subsystem. Without it the SMP is still capable of performing iterations on nuclei with a configuration space of up to 13,500 elements using the MCM's own local memory to store the vectors. However the main elements of this subsystem, e.g. DRAM control, PFB design, bus interfacing, have been tried and tested in other parts of the SMP. Therefore once the resources become available it should be possible to construct both CM and CMA-bus fairly quickly.

4.4.1 Central Memory Overview

The Central Memory Module is designed to be able to store up to 4 Lanczos vectors at any time and therefore will have 4M-bytes of primary storage built using 256K DRAM devices. It will be split into two independent 2M-byte banks with each capable of storing two vectors of up to 256K 4-byte elements. Each bank will be interfaced to CMA-bus via a 64-bit data bus and also interfaced separately to C-bus via a 16-bit local data bus. It will also be possible to access the CM local bus via a VME-bus compatible port, intended for DMA transfers to and from the CM by the backing store controller.

Both the CM banks will be dual-port with respect to the local bus and CMA-bus. Since each bank will be completely independent of the other, having their own DRAM refresh and control circuitry, it will

therefore be possible for one bank to be accessed from CMA-bus at the same time as the other bank is being accessed from the local bus. However in the event of both buses simultaneously requesting access to the same bank then arbitration logic will decide on a first-come-first-served basis which request will proceed first. Each bank will have its own arbitration logic, which will be based on the design used in the other arbiters within the SMP system. Any request to access a bank may also have to compete with the DRAM refresh cycles thus giving another level of arbitration to CM accesses. Similarly accesses via C-bus must go through another level of arbitration before they can gain use of the local bus, this time with accesses from the external system port.

The independence of the two memory banks allows one bank to be devoted to holding the vectors currently being processed by the MMPU while DMA transfers can take place on the other bank in preparation for the next iteration. The elements of the initial and final vectors will be interleaved in CM so that V_{im} and V_{fm} will be in one 64-bit location. Thus both elements can be read in one CMA-bus cycle, a facility which will greatly enhance H-mode processing.

For each task performed by an MCM in H-mode the elements V_{im} and V_{fm} must be read from CM (eqn 2.12). The element from the final vector, V_{fm}, must be updated (according to eqn 2.12b) and written back into the same location in CM, overwriting the previous value. During the time between an MCM reading an element from CM and writing the new value back in, no other MCM should use the same element as part of another task, since the update from one of the tasks will inevitably be lost. To prevent this occurrence each half word (4-byte) location in CM will have a ***lockout bit (L-bit)*** associated with it. This bit will be set when the vector element contained at the location is read and reset only when it is written back again. An MCM will thus be informed via the L-bits that the element it requires from the final vector is being updated by another MCM and so the current value of the element is indeterminate. In

this case the MCM must attempt to read the vector element again until he is successful. The L-bits for the initial vector elements are ignored by the MCMs during any read since these values are never altered during an iteration.

The MCMs must also read from central memory at the start of each new prime state in order to read V_{in} (eqn 2.12b). Therefore in order not to alter the L-bit for the final vector element at the same location the MCM performs only a half-word (32-bit) read from CM in order to obtain this value.

Each L-bit will in fact be duplicated to protect against soft and hard memory errors. The two L-bits, held in separate devices at the same location, will be exclusive-ORed to test for such errors each time a read is made from CM. As well as two L-bits for each four byte word there will also be one parity bit provided per byte in CM. Thus for each 64 bit data word there will be an additional 8 parity bits and 4 L-bits and so each bank in CM will require a total of 76 256K DRAM chips.

It is proposed that the Hitachi HM51258-8 256K x 1 Static Column Dynamic RAM be used for CM. This device has a read/write access time of 85ns and cycle time of 155ns and a read-modify-write cycle time of 180ns. It is also intended that the Intel 8207 can be used as the CM DRAM controller.

The DRAM cycle time will place a lower limit on the CM access time as this will be the slowest link in the access chain. Since each access will require a read-modify-write cycle to test and update the L-bits then this lower limit will be 180ns using the HM51258-8. Although the dedicated prefetch buffers will be high-speed, built with fast bipolar logic, they will inevitably impose further delays as will the DRAM controller. Taking all these delays into consideration CM access time, via CMA-bus, is estimated at around 240ns, thus allowing approximately 4 accesses/microsecond.

4.4.2 CMA-Bus

CMA-bus is intended as the pathway between the MCMs and the Central Memory Module during a Lanczos iteration. As such it must be capable of supporting both read and write accesses to a large, random access memory store. It must therefore have, unlike I-bus, an address bus and bi-directional data bus. Although the CM will also be interfaced to C-bus there are still good reasons for a dedicated pathway for the MCMs to use in accessing CM:

1/ The potential usage of CM by the MCMs is very high and C-bus would soon become a bottleneck were it the only means of accessing CM. The systems use of C-bus as its prime means of communication would thus be greatly reduced. Therefore an extra dedicated bus, for use purely by the MCMs to access CM, greatly reduces C-bus traffic allowing C-bus to perform its important system control and communications task.

2/ The initial and final vector elements are held in single precision (32-bit) floating-point format. Therefore since each read access to CM will require one initial and one final vector element, a pathway which can support 64-bit data transfers will greatly increase bus bandwidth over the 16-bit data path of C-bus.

3/ When an MCM reads two vector elements from CM, the two L-bits associated with the elements must also be read. On C-bus this would require another bus cycle to read the two bits, which is obviously extremely wasteful. A dedicated bus therefore, which provides two lines for carrying the L-bits reduces this extra bus usage.

4/ Similarly a dedicated bus interface with prefetch buffers which operate in parallel with the MCM processor (like the I-bus PFB) will also greatly increase MCM performance.

Although CMA-bus has not yet been implemented the most complex and important parts of its design have already been tried and tested in other parts of the system, e.g. the proposed request and arbitration logic and some of the ideas for the interface. The arbitration protocol

used on CMA-bus and thus the actual arbiter and requester will be identical to that used on I-bus. The prefetch buffers will however have to be more sophisticated than those used on I-bus because of the presence of an address bus and bi-directional data bus. The nature of CMA-bus means that the MCM must first write the address of the location in CM which it requires to access to the CMA-bus PFB. The MCM is then free to perform a task while the PFB accesses the appropriate location in CM.

As a result of its dedicated task CMA-bus can have a much simpler structure than C-bus, requiring neither its utility bus nor its interrupt bus and using a simplified DTB and arbitration bus. The single level arbitration bus is identical to that on I-bus and so requires only 3 lines and one daisy chain bus grant line. The details of the DTB remain the same as in [Mac83], apart from one alteration. In summary the DTB consists of:

1/ **CMD63-CMD0** : the 64-bit CMA data bus. This consists of two half buses, the upper (CMD63-CMD32) and lower (CMD31-CMD0) bus.

2/ **CMA26-CMA3** : the CMA address bus, capable of addressing up to 16M 64-bit words.

3/ **CMLD1, CMLD0** : the L-bit data lines, used only during a read cycle from CM.

4/ **CMDS1*, CMDS0***: the data strobes for the upper and lower halves of the data bus respectively. These independently signal a transfer on the upper and lower halves of the data bus.

5/ **CMWE1*, CMWE0*** : the two write enable strobes, independently governing write cycles on the upper and lower halves of the data bus respectively.

6/ **CMDTACK*** : the data transfer acknowledge signal. When this signal is asserted the bus master must latch any data being read, negate all other DTB signals and release the bus for the next master.

7/ **CMBERR*** : the bus error signal is asserted if an invalid address is

used or a parity error occurs.

Depending on the state of the data and write strobes, accesses on CMA bus are either a full 64-bit read or write, a concurrent 32-bit read and 32-bit write, or a 32 bit read/write on either half of the data bus.

No discussion of the arbiter or requester for CMA-bus will be included here since, as we have said, they are identical to the ones used on I-bus. The CMA-bus PFB interface is detailed in [Mac83] and at present remains completely unchanged.

As has already been said the bandwidth of CMA-bus is primarily determined by the cycle time of the CM dynamic RAMs and not the bus arbitration time. Thus assuming 16 MCMs with a CMA-bus cycle time of 240ns then each MCM will take at most 16 x 240ns = 3.84us to perform a read from CM. This assumes that all modules are requesting at the one time and also that arbitration time is buried in the bus cycle time.

4.5 The Microcomputer Modules

The MCMs are self-contained microcomputers which act as slaves to the SM. In order to reduce usage of the SMP communications subnet by the MCMs, they are endowed, as much as possible, with their own local resources, e.g. local memory capable of storing all program code and frequently used data. To this end the MCMs have 128K bytes of dynamic RAM suitable for holding data tables and 8K bytes of fast static RAM to hold program code and workspace area. The MCMs are also provided with the hardware and software to interface to the communications subnet already described.

The requirement for sufficient local memory and subnet interfaces are the only hardware constraints on the design of the MCMs. Indeed the subnet interface need be the only application specific hardware on the MCMs. Were it not for this "off the shelf" microcomputer boards could have been bought to perform the task. Another reason for designing

custom MCMs is that they can be made to have much higher performance capabilities than any microcomputers which are currently available.

The complexity of the MCM's task demands that a high-performance microprocessor be used, one which is capable of fast table searching, data manipulation and arithmetic processing. The Motorola **MC68000** 16/32 bit microprocessor is well suited to this application, with its 32-bit internal data and address registers, a powerful and regular instruction set, and 16 Megabyte direct addressing range [Mot68000, SG79]. Its full 32-bit successors, the **MC68020**, **MC68030** and **MC68040**, are completely object code compatible with the MC68000. For example the 16.67 MHz MC68020 has 4 to 5 times the performance of an 8 MHz MC68000 [MMM84]. The successors also have a coprocessor interface to which a ***floating point arithmetic unit***, the **MC68881** or **MC68882**, can be attached giving even further improvements in processing power. Thus, because of the power of the MC68000 and its successors, it is an ideal processor on which to base the MCMs.

It should be apparent that the internal architecture of each MCM need not be identical, as long as each conforms to the external constraints already mentioned by providing the necessary local resources and subnet interfaces. In terms of design effort it is obviously sensible that the MCMs are identical. However having said this the only two MCMs at present in operation are very different in their hardware and software. The first MCM built, ***MCMI***, is a simple, single processor module, while ***MCMII*** has two processors working on a master/slave basis and a hardware floating point arithmetic unit. This change in architecture was dictated by the ability to radically increase the MCM performance by utilising advances in technology.

4.6 The Supervisor Module

The role of the SM within the MMPU is that of a master, controlling and

monitoring the different parts of the SMP system. In order to support this specialised function it has the most privileged rights of all the modules and is the one with the most resources available to it.

4.6.1 Supervisor Module Hardware

Central to the SM is an **MC68010** (8 MHz) virtual memory microprocessor. This is an enhanced MC68000 microprocessor being able to support a virtual memory/machine system [MM83]. It also has improved instruction execution times while still remaining fully object code compatible with the earlier MC68000. Using virtual memory techniques an MC68010 system can be made to appear to the user as having the full 16 Mbytes of primary memory available to him, while in reality only a fraction of the address space actually contains physical memory. This is supported with an MC68010 since it has the capability of suspending an instruction's execution when a bus error is signalled and then completing the instruction after the required action has been taken within the bus error exception routine, an ability which the MC68000 does not have. Another addition to the MC68010 is a vector base register which is used to determine the base of the exception vector table in memory, thus allowing this table to be relocatable and so enabling multiple vector tables [Mot010].

The SM also has a ***memory management unit (MMU)***, the **MC68451**, which further supports virtual memory on the SM by performing address translation and protection on the full addressing range of the processor [Mot451, Mot82]. The internal registers of the MMU can be accessed by the SM's MC68010 (in supervisor mode only) in order to program it and when correctly programmed the MMU will translate all logical addresses to their physical counterparts. The MMU can also interrupt the MC68010 when a chosen section of memory is accessed as well as prohibit write accesses to any sections of memory.

Using the MMU the logical address space of the SM can be tailored

to fit any requirements. For example the SM can be made to "see" the physical address space of any external module (e.g. an MCM) within its own local logical address space. This enables the SM to immediately run and debug any software written for one of the external modules using their memory and devices without the need for modifying the code in any way. However the MMU reduces processor performance by slowing down memory accesses due to the time taken to translate addresses (approximately 150ns maximum). In order to avoid this delay the MMU may be completely bypassed in circumstances when it is not required.

The SM is also equipped with 128K of DRAM, 8K of static RAM, 8K of EPROM (with provision for up to 20K), two MC68230 PI/Ts and a Signetics/ Mullard SCN68681 **dual universal asynchronous receiver/transmitter (DUART)**. The arbiters for each of the system buses are also placed on the SM.

The DUART provides the SM with two very flexible serial, full duplex RS232 type links. On the SM one of these serial links is normally connected to a terminal and the other to a remote "host" computer system. The two serial interfaces on the SM have been built such that they can be connected together by bringing one of the programmable output lines low on the DUART. This causes the SM to become completely transparent with respect to the serial links and creates a full duplex link directly between the terminal and host. The SMP system host is currently a Motorola EXORmacs microcomputer based on the MC68000 microprocessor and running under the VERSAdos disk operating system. Using the facilities available on the host system software for the SMP can be written, assembled, linked and loaded into the SMP via the serial link. Similarly data can be passed to the host from the SMP and stored on disk for future reference.

Apart from being interfaced directly to C-bus the SM also has a general purpose interface provided on it. This interface is a simple 16 bit data and 24-bit address expansion bus which is intended to connect

the SM to peripheral system devices e.g. an EPROM programmer.

The SM is the only module in the SMP system which will have software resident in EPROM. It is intended that this should hold a monitor/debugger program and also, when a disk and disk operating system become available, a bootstrap loader.

While it is possible for external modules to directly access the SM from C-bus this is only done in unusual circumstances due to its highly privileged nature. Indeed the SM cannot be accessed by any of the modules normally present within the SMP system since the PRIV* line must be active to select it. Prior to its construction the SM function was carried out by a standard Motorola VME module, the VECPU105 monoboard computer, and at present only this module has the capability of accessing the SM's address space. The VECPU105 has an MC68000 microprocessor, two serial links (RS232), a PI/T and a resident monitor/debugger. Although the proper SM completely replaces the VECPU105 so that it is not required during any Lanczos iteration, it still has its uses during initialisation and testing of the system because of the current lack of a full resident monitor program on the SM.

4.6.2 Supervisor Module System Monitor

A set of rudimentary monitor routines have been written for the SM system to provide an environment in which users can more easily integrate software into the SMP system. The routines fall into three categories; SM initialisation, I/O and intermodule data transfers.

Supervisor Module Initialisation:

Included in this category is a routine to initialise the SM's exception vector table. This initialises all the exception vectors, except the user interrupts and two of the TRAP exceptions, for the MC68010 to call a default error handler routine. This default routine will then send an error message to the terminal and allows the user to identify the

exception which occurred. Two of the 16 MC68010 TRAP instruction exceptions are reserved for SM use by the monitor, these are TRAP #0 and TRAP #15. The TRAP #0 exception is reserved for user program termination so that a user program can easily pass control over to the system monitor at the end of execution. The TRAP #15 exception is reserved for calling system routines to transfer data between the SM and the other SMP modules as will be explained later.

Other routines set up the SM PI/T and the DUART. The DUART is set up so that if it receives a break from the terminal then it will interrupt the processor. The necessary interrupt vector is placed in the SM exception vector table for this.

Supervisor Module I/O

A library of routines for input/output via the DUART have been built up. These include routines to receive and transmit single ASCII characters as well as ASCII strings. Routines are also included to convert between ASCII coded decimal integers and binary integers. All errors are checked for in these routines, e.g. parity error, framing error, etc, and any which are found are signalled to the user. All these routines will handle I/O from/to either the terminal or the host. However there are two specific routines to handle the host so that a file can be listed by the host and captured by the SM or vice-versa. These routines include sending the appropriate command string to the host to list or create the relevant file. The ability to send data to the host and have it captured as a file is particularly useful to the SM due to its lack of hard disk.

Intermodule Data Transfer

As has been said the MC68010 TRAP #15 exception is reserved for transferring data between the SM and other SMP modules over C-bus. This function is reserved as a system function since it would be unwise to allow user programs to write data to other modules without due control and supervision by the SM system. Hardware also supports this in that the SM cannot perform transactions via C-bus except when the SM MC68010

processor is in supervisor mode (this is signalled by the MC68010 functions code lines). Similarly the SM PI/T which governs the control of the PRIV* and CONTROL* lines on C-bus is only accessible in supervisor mode. The TRAP #15 function will eventually be extended to the monitor kernels of all SMP modules so that all C-bus accesses are controlled by the system.

Embedded in the SM monitor software is a table giving details of the modules which can be physically present within the system. This table, the ***Module Identification Table (MIT)*** has a 96 byte entry for each possible module and allows any details concerning a module to be identified to the SM, e.g the MID, differences in the address map of any of the modules, any specific hardware features which they may have, etc. During system initialisation the SM module determines which of these modules are actually present within the system and signals this in the MIT. The SM determines if a module is present by placing its MID on the map select lines of C-bus and then attempting to read from the modules memory. If the module is not present then the SM will receive a bus error signal. A bus error exception routine is temporarily set up for this function and if entered is alters the MIT to show that the module is not present in the system.

The TRAP #15 exception caters for physical reading and writing of data blocks to/from SMP modules. That is a user program can request to transfer data to a specific (physical) module by supplying the modules MID to the TRAP #15 function. The calling routine must also supply the source and destination addresses, the number of words to be transferred and a code specifying which function is being requested. At present three functions are catered for; writing a block of data to a module, reading a block of data from a module and bus broadcast transfers to the MCMs. The request for a transfer is checked first before being executed. Firstly, for non-broadcast transfers, the MIT is accessed to determine if the module is actually present in the system. The source and

destination addresses are also checked to determine that if they are valid areas of memory to transfer data to. If an error is found then the routine terminates and returns an error code to the calling routine. If no errors are found then the source/destination address on the module is translated to an offboard address by adding $800000, i.e. bit A23 set high.

The functions which are available at present via TRAP #15 although limited are all that are required for the SMP system. However they they can easily be extended to form the core of a multi-processor monitor environment. For example the bus broadcast routine could be extended to allow the user to request that certain modules be excluded from the transfer. Similarly extra functions could be added to allow the user to request the MID of a module of a specific type or with a specific hardware function available to it. The TRAP routine would then simply access the MIT table to determine the MID of the module which met the requirements of the user.

4.6.3 Supervisor Module SMP Software

There is currently available on the host computer a Pascal runtime library for the SM. This allows Pascal programs to be edited, compiled and linked on the host and then run on the SM. Most of the routines required to run on the SM for the purposes of the SMP have been written in Pascal with the remainder being written in assembler and called as subroutines to the main Pascal program.

The main task of the SM during any calculation is the initialisation of the MCMs and PG prior to the first iteration. This requires a number of basic tasks:

1/ generating all the look-up tables required by the Basis Generation function of the PG.

2/ generating all the tables required by the MCMs for determining the matrix element magnitude and sign.

3/ generating a table of matrix element magnitudes for use by the MCMs,

4/ transferring all of these tables into the memory of the relevant modules,

5/ sending commands to the PG and MCMs to tell them to commence processing.

A Pascal program TABLEBLD has been developed to run on the SM to function as the user interface to the SMP system. This program allows the user to set up the details of the nucleus under consideration and performs the system initialisation functions just mentioned as well monitoring the SMP system during runtime. Before a calculation can be carried out a file containing a number of data values and tables must be loaded into SM memory from the disk of the host computer. This file includes a table of energies for single particle states which allows the SM to build a table of Hamiltonian matrix element values required by the MCMs. A table detailing a default assignment of angular momentum values to the 24 active orbitals is also included. Once loaded into SM memory this latter table can be edited by the user to give any particular assignment of angular momentum values that is desired. When all the necessary tables have been built in SM memory they are transferred into the PG and MCM memories as required before the start of processing.

During each iteration the SM monitors the progress of the MCMs and MFG to watch for any errors which may be flagged by them. At present after each iteration the SM forms the complete final vector from the partial results held in the memory of each MCM and can then make it available for display at the terminal or send it to the host to be stored on disk.

We have only given a brief outline of the SM but it should now be apparent that it is a highly flexible, versatile, "stand alone" microcomputer, adequately capable of supervising the SMP system. We need not go into any further detail regarding its implementation since,

although it is crucial to the success of the SMP as a whole, it does not play an important part in actually determining the performance capabilities of the SMP. However due to the importance of the MCMs in processing each iteration we will devote the following chapter to a more detailed discussion and description of them. The details of their architecture are set out as well as the steps involved in processing the TSWs and so performing a Lanczos iteration.

CHAPTER 5

The Microcomputer Modules

5.0 Introduction

The SMP system is in effect made up of a two stage pipeline, consisting of the MFG and MMPU. The MFG is purely a data producer feeding a sole consumer, the MMPU, with TSWs to process. The relationship between these two subsystems is therefore governed by supply and demand. Should the demand for TSWs from the MMPU outstrip their supply then the MMPU will simply have to wait while the MFG catches up. Similarly if supply outstrips demand then the MFG will have to halt its work while the glut of data is reduced. Therefore one of these subsystems will impose an upper limit on the performance of the SMP as a whole. It is only advances in semiconductor and microprocessor technology that have enabled a high-performance MMPU to be designed, based on MCMII. The SMP system is therefore now in the position where the MFG is the system bottleneck whereas before the MFG would have had approximately twice the performance of the MMPU were it based on MCMI.

The most significant advances which have been incorporated into the MCMs are;

* the production of the 8 MHz MC68000 (before this the 8-bit MC6809 would have been used on the MCMs),
* the later introduction of the 16 MHz MC68000,
* the introduction of the National Semiconductor NS32081 floating-point unit,
* and finally a new design utilising two MC68000s per MCM.

These last three advances have all been incorporated onto MCMII, improving it by a factor of 9 over MCMI and thus giving the MMPU its current potential performance capabilities. The advantages of a modular system can therefore be clearly seen, in that advances in technology can be easily incorporated into the MCMs without having to remove or disturb in any way current components of the system, so long as new MCMs conform to the requirements of each of the SMP buses. Indeed the fact that the buses are asynchronous allows faster interfaces to be incorporated onto new MCMs, so long as they conform to the original bus protocol standards. This potentially allows faster transfer rates on any new MCMs without the need to change current, slower interfaces.

The MCMs should not be viewed as highly specialised, dedicated computer modules. They are in fact high performance microcomputers with a highly flexible architecture. There is very little that is specialised about their structure and the parts that are dedicated in no way interfere with their general purpose capabilities. The class of problems that the MCMs can be applied to is as varied as that of any microcomputer. Even the latest MCMII with its two processors is still a general purpose microcomputer, the user having the option of using the second processor as a high performance arithmetic processor, as a slave processor doing any tasks the master commands it to perform or indeed of simply ignoring it altogether. It is this flexibility of the MCMs that gives the whole MMPU its great power as a multi-processor system. However this non-specialised nature of the MMPU in no way compromises the ability of the system to carry out its intended function since none of the abilities required for this have been sacrificed to give it its current structure.

What follows is a description of the structure and architecture of the current MCMs. Although MCMI has been superseded it is still an important, working part of the MMPU. It was from the experience gained in designing, building and working with MCMI that the designs for MCMII

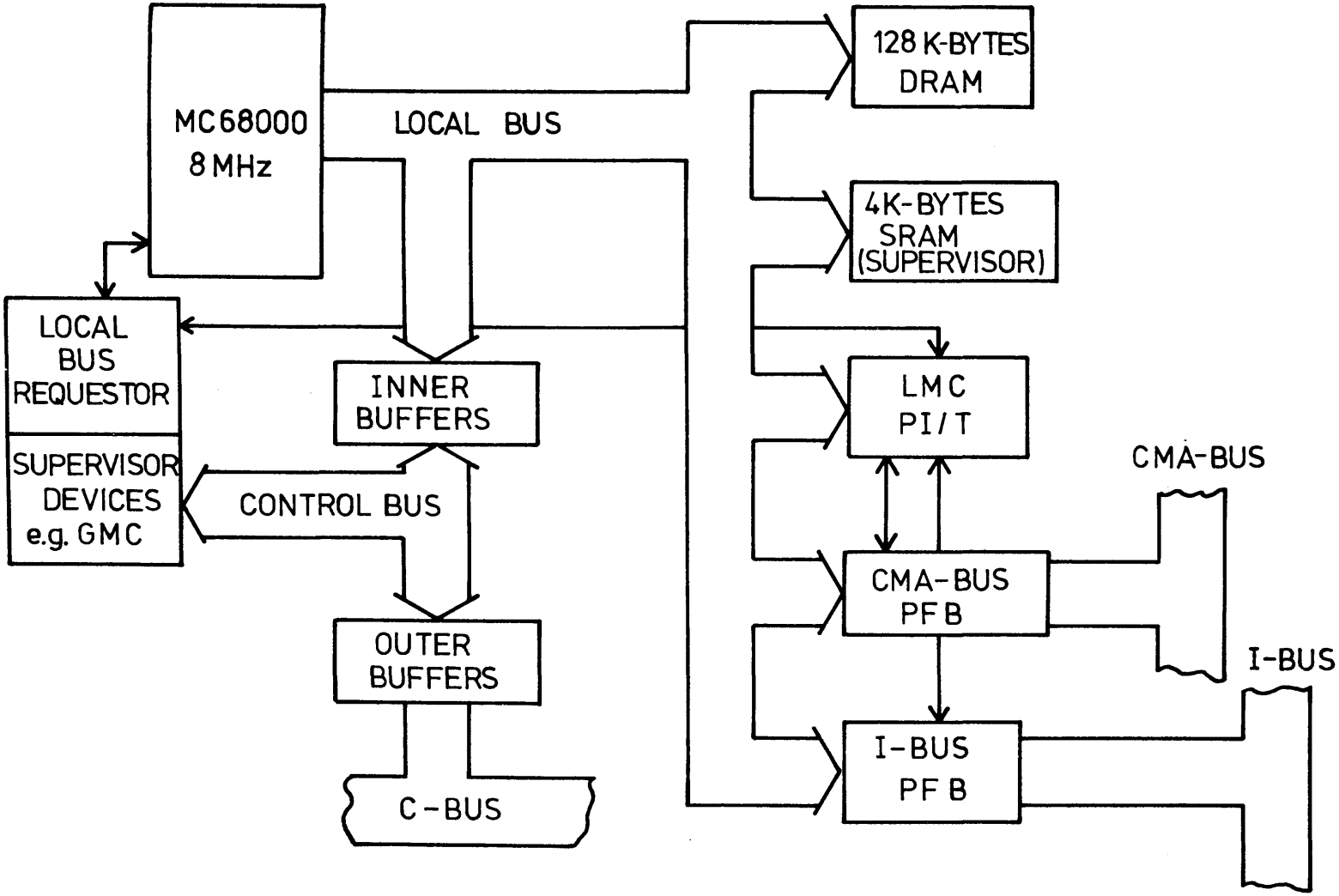


FIG. 5.1 MCMI OUTLINE

were developed and therefore an outline of MCMI is important to have before going on to discuss MCMII in greater detail.

5.1 MCMI Outline

A block diagram of MCMI is given in figure 5.1, showing its major components and their interconnection. The control bus can be seen between the inner and outer C-bus buffers, allowing C-bus masters access to the modules control map. The global decode logic is also situated on the control bus to intercept requests from C-bus to access the local bus. Any such requests are sent via the *Local Bus Requester (LBR)* to the local processors own bus request input. The MC68000's own bus arbitration control will automatically issue a bus grant signal and give up the local bus and only then can a C-bus master assume control.

It can be seen that all memory and devices on the local bus are completely dual-port with respect to C-bus. Thus all accesses to any of these devices are identical, regardless of whether they come from the onboard processor or from C-bus, except of course that all data transfer acknowledge signals are routed to C-bus in the latter case.

The memory requirements of the local processor are catered for by the 128K bytes of dynamic RAM and 4K bytes of static RAM. The DRAM although not large by todays standards is enough to hold the user program code and data for SMP processing and could be fairly easily updated to 512K bytes by using the 256K DRAM chips. The static RAM is placed in the lowest 4K of the processors address space. It is limited to supervisor mode accesses only, as decoded from the MC68000 function codes [Mot68000], and therefore only system functions can access this memory. The MC68000 reset and other exception vectors are placed in the lowest 1K bytes of memory and normally this would be implemented with ROM rather than RAM. The lack of ROM on the MCM presents no difficulties and indeed has significant advantages both for module development effort

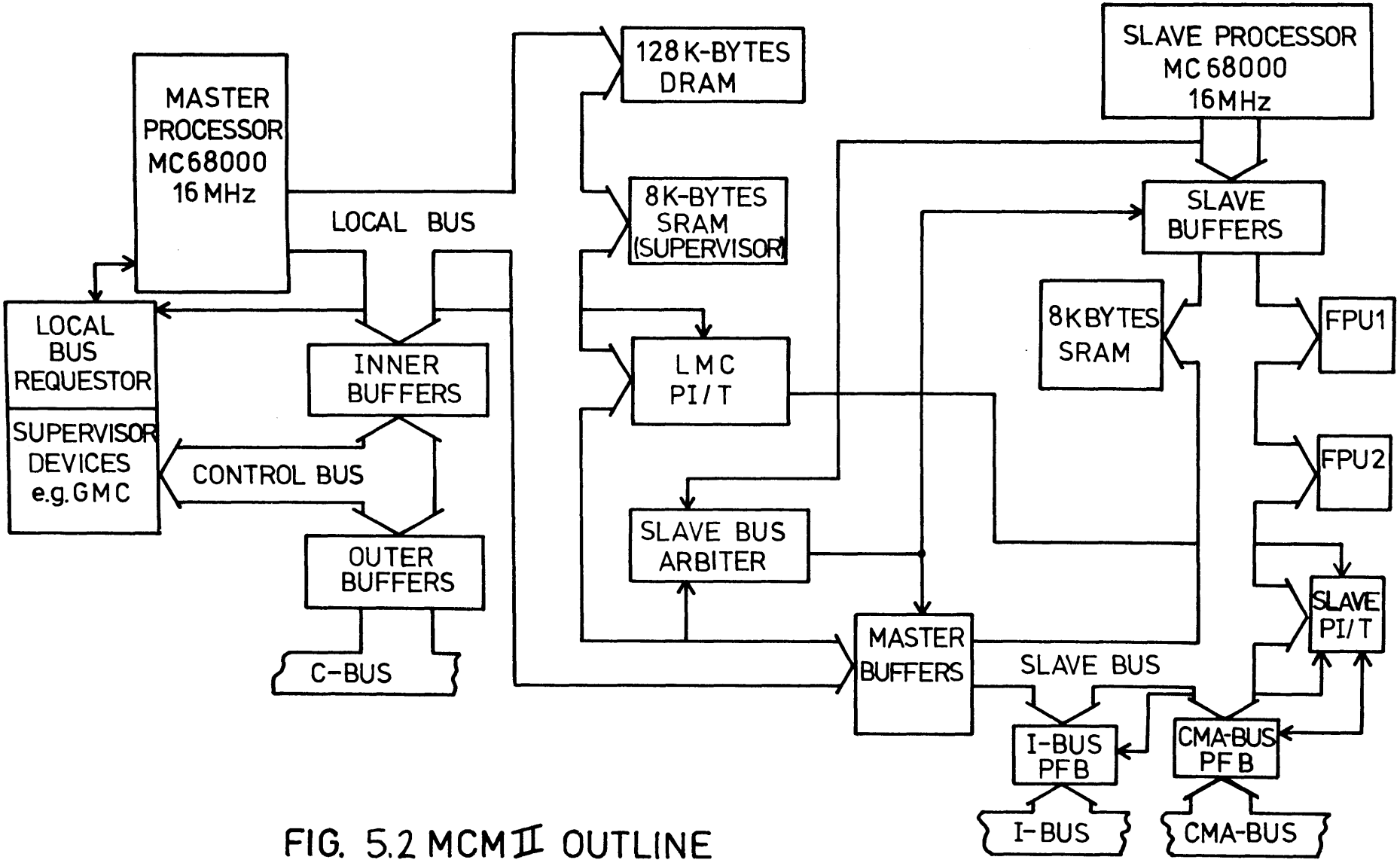


FIG. 5.2 MCM II OUTLINE

and operational flexibility. On power up the MCM processor is held halted by the GMC until released by the SM. However prior to doing this the SM should provide the processor with a reset vector, as well as any other necessary exception vectors and appropriate startup program code. Thereafter the module can be supplied with program code by the SM depending on what task(s) it requires the MCM to perform.

For shell-model processing the MCM is required to do significant amounts of floating-point arithmetic. However since MCMI has no hardware unit to perform this task all its arithmetic must be carried out in software. At present Motorola supplied MC68000 routines are used which implement the IEEE P754 floating-point format although they do not fully implement the arithmetic standard [IEEE81]. The approximate average times for addition and multiplication (of two non-zero numbers) are 80 μsecs and 100 μsecs respectively, for an 8 MHz processor with no wait states. Since the MCM has to do two multiplications and two additions much of MCMI's time is therefore taken up with arithmetic and therefore any consistent method of reducing this time is desirable. One possible solution is to use faster software routines which use a non-standard data format, e.g. Motorola Fast Floating-Point format routines, however this would be at the expense of the overall ease of use. In terms of speed, consistency and flexibility the best solution is to use one of the hardware arithmetic processors available today, e.g. the Motorola MC68881 or MC68882, National Semiconductor 32081 or AMD 29325 all if which implement the IEEE standard. It was therefore decided to include such a hardware device on MCMII.

5.2 MCMII Structure

Figure 5.2 shows the overall structure of MCMII. It is quite clearly an extended MCMI, with the basic structure of MCMI still being present but having the addition of a ***slave bus***. The master processor, which is now a

16 MHz MC68000, resides on the local bus. As with MCMI all devices are dual port with respect to C-Bus, including those on the slave bus.

The slave bus is the local bus for the slave processor by which it communicates with its local devices. The slave bus is completely accessible to the master processor so that all devices interfaced to it are dual port, i.e. can be accessed by both the master and slave processors. However the slave processor is confined to the slave bus and cannot access the local bus. Devices on the slave bus are; 8K bytes of fast static RAM, a PI/T for slave subsystem control, the I-bus and CMA-bus interfaces and two ***floating-point units (FPUs)***, although only one has as yet been included.

The arbitration scheme for the slave bus is fundamentally different from that used on the local bus. When a request is made for the local bus from C-bus it is submitted via the LBR logic to the master processor's bus request input. Only after the master processor has finished any current access is the bus granted, which is potentially a time consuming process. In contrast to this the arbitration for the slave bus happens independently of the slave processor and is completely transparent to him.

In essence the slave bus is a pool having two mutually exclusive access routes with neither processor having any particular right of ownership. That is there is not one privileged processor which grants the right of access to the slave bus to the other processor. Instead the processors compete on a cycle by cycle, first-come-first-served basis for the use of the slave bus. Thus the first processor to get its request to the arbiter will have other processor's buffers disabled.

However in acknowledgment of the fact that under normal operating conditions the master processor will use the slave bus only rarely, the slave processor's buffers are enabled by default to save him from being needlessly delayed while its buffers turn on. Only when the master processor is granted access to the slave bus are the slave buffers

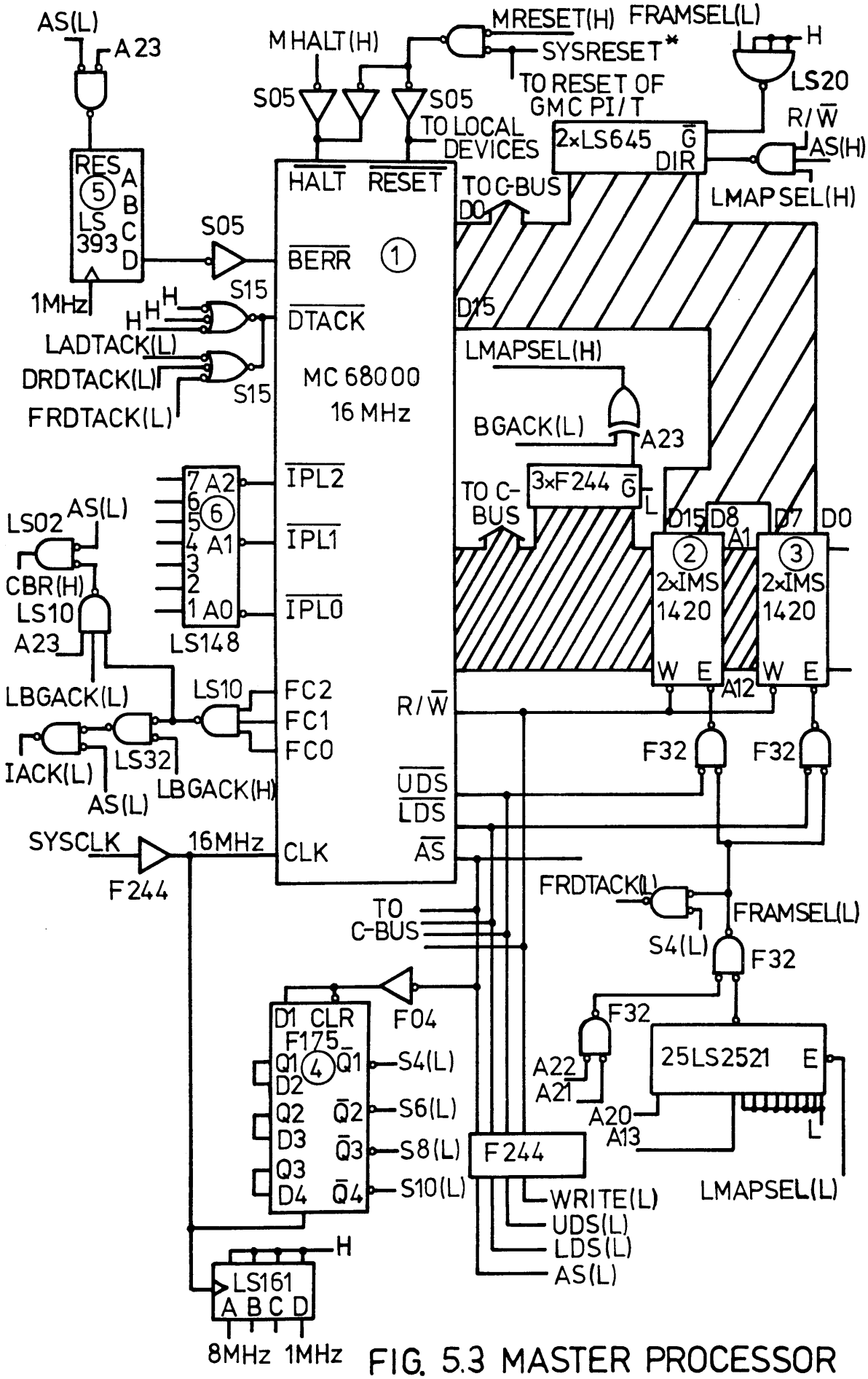


FIG. 5.3 MASTER PROCESSOR

disabled. In the case where the two processors do both require the slave bus at the same time then the loser in the arbitration contest will of course have its cycle delayed. However using this method the delay to either processor will never be as much as that obtained using the MC68000's own bus arbitration control logic.

The hardware to implement the slave bus sharing scheme and other components of the MCMII structure will now be discussed in more detail.

5.2.1 The Master Processor

Figure 5.3 gives the details of the master processor sub-system. At its heart lies a 16 MHz MC68000, with a minimum bus-cycle time of 250 ns. For a 16 MHz processor to be guaranteed to run with no wait states, static RAMs of at most 80-85 ns access time must be used. Dynamic RAMs have the extra delay of the controller to be considered and so have to be faster to achieve no wait state accesses. There are a number of static devices available today with this speed, the Inmos IMS1420-55, a 4K x 4 memory with an access time of 55 ns, being the one chosen for MCMII. Four of these are used to provide the master processor with 8K bytes of supervisor memory. This memory is used to hold the master processors exception vector tables, code for exception handler routines, the system stack, SMP program code and frequently used data. This amount of static RAM is more than adequate for SMP purposes but additional user memory can be easily added.

Using the AMD 25LS2521 8 bit comparator, 7, for decoding the memory select, allows for simple alteration of the position of memory within the address space. This can be done by changing the levels on the B inputs to the comparator. Indeed a rudimentary memory management could be implemented if the B inputs were tied to a PI/T port. This would allow the positioning of memory to be dynamically altered, although this would not be suitable for the supervisor memory.

The MHALT(H) and MRESET(H) are both driven by the module's GMC thus

allowing individual MCMs to be halted or reset by the SM or any other module capable of accessing the MCM control map. On power up the outputs of the GMC PI/T come on high thus assuring that each MCM is initially held reset and halted until released. The MCMs are also reset when the C-bus SYSRESET* line is activated. This signals a complete SMP system reset, and is the only method of reseting the GMCs.

In order to produce carefully timed data transfer acknowledge signals, DTACK(L), for the master processor an F175 (quad. D-type register with common clock and clear) is used, 4. When the masters AS(L) is activated, sometime within processor state S2 or S3 [Mot68000], the clear is removed from the F175 and the D1 input taken high. Q1 will then be brought high at the next positive edge of the clock, which is always at the start of state S4, Q2 will be brought high at the start of state S6 and so on. Thus for memory systems running with no wait states Q1 is used to produce DTACK(L). While for slower memories or devices requiring say 6 wait states Q4 would be used to produce DTACK(L).

When the processor attempts an access to a non-existant device then no DTACK(L) will be produced and instead a BERR(L) signal must be used to terminate the processor cycle. This is produced by the LS393, 5, which is clocked by a 1 MHz signal. On an attempted access to a local device, when both AS(L) and A23 are low, a BERR(L) signal will be produced after 8 usecs if the cycle is not terminated. For non-local accesses via C-bus the C-bus BERR* signal is monitored.

Interrupts to the MC68000 are delivered via the LS148, 6, which in the case of multiple requests will present the highest priority level to the processor. Seven interrupt levels are provided for with an MC68000. More than one interrupting device can be externally chained to the one level, allowing an unlimited number of devices to interrupt the processor. However with the MCMs no more than seven interrupting devices are foreseen.

The interrupt acknowledge cycle, signalled by IACK(L), is produced

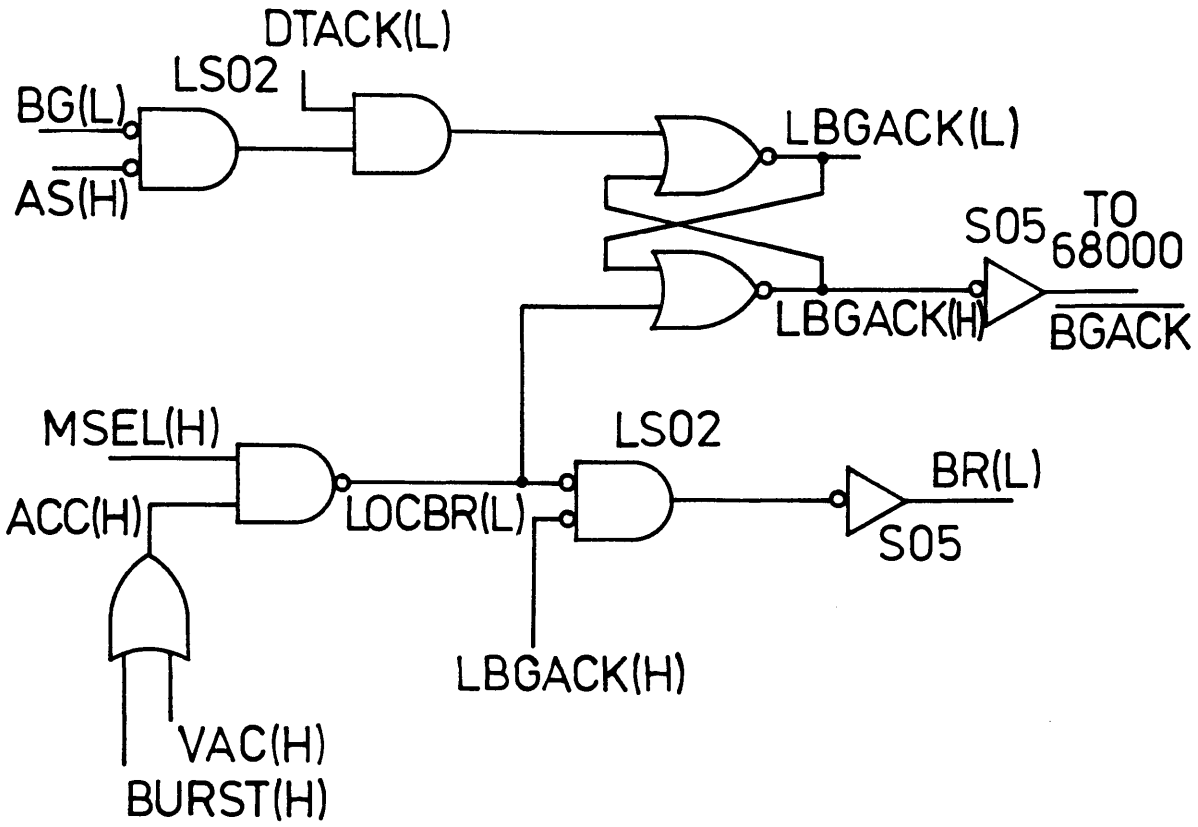


FIG. 5.4 LOCAL BUS REQUESTOR

in response to an interrupt to the processor and is used to determine the interrupt vector. Since only local devices can interrupt the processor no off board accesses are ever needed for this cycle i.e. MCMs are not C-bus interrupt handlers. C-bus requests, produced in response to CBR(H) active, are therefore only initiated in response to a normal processor cycle with A23 high.

In order to comply with MC68000 output loading requirements all the address lines, data lines and strobes are buffered before being used. However for the purposes of speed the MC68000 lines which are used for decoding the supervisor static RAM are unbuffered. All C-bus lines are attached directly from the inner C-bus buffers to the MC68000 lines, thus allowing C-bus masters to emulate the master processor and give all local devices dual-port access capabilities.

5.2.2 The Local Bus Requester

The MCM acts as a C-bus slave, in both the local and control modes although in the latter mode the local processors are unimpeded by the access. When the C-bus master wishes to access the MCM's local bus, then the local processor must be removed from it before the interface can be put in local mode.

A local bus request, LOCBR(L) figure 5.4, on an MCM is generated as soon as a C-bus cycle is in progress, signalled by VAC(H) active, and the modules local map is selected (either by the map-select lines or bus-broadcast line), MSEL(H) active. In the case of block transfers, signalled by BURST(H) active, then a LOCBR(L) will be generated just as soon as the local map is selected (see figures 4.6 and 4.7 for C-bus module select). LOCBR(L) active immediately generates a BR(L) signal to the local processor which will automatically generate a bus grant signal, BG(L), a minimum of 1.5 clock cycles and a maximum of 3 clock cycles later. The LBR then waits until the AS and DTACK signals on the local bus are inactive, signalling that the local processor has finished

with the bus, before asserting a bus grant acknowledge, BGACK(L), to the processor. This signal informs the local processor that its bus is being used and only once BGACK(L) is negated will it again assume mastership of the local bus. BGACK(L) is only negated once the external bus master has finished its access(es). Therefore during block transfers the local processor will be held off its bus until BURST(H) is negated. Note that during any period when the local (master) processor is removed from its bus the slave processor is not interfered with in any way, unless of course the C-bus master accesses the slave bus.

Although the local processor must be requested for the use of its bus, granting of the local bus to the requesting module is automatic and immediate. This is true no matter how important or crucial the task that the local processor is performing. Thus the local processor has lower priority use of its own bus in comparison with any C-bus master which is requesting access. However this situation is obviously more desirable than one where the local processsor has higher priority and could delay in giving up its bus to the C-bus master until a time when it so desired. In this case the C-bus master would be wastefully delayed waiting for the local processor, thus causing a reduction in C-bus bandwidth and therefore a global reduction in performance.

5.2.3 Dynamic RAM Subsystem

This subsystem is implemented using the MCM6665-15 64Kx1 dynamic RAM with an access time of 150 ns and cycle time of 300 ns. A National Semiconductor DP8409 Multi-mode Dynamic RAM controller/driver is used as the refresh and address multiplexing controller. This device is implemented in high speed Schottky TTL and is capable of driving up to 88 16K, 64K or 256K DRAMs, with only 25 ns (typical) propagation delay. With these speeds the 16 MHz MC68000 must run with 4 wait states on accesses to this memory giving a bus cycle time of 375 ns.

An external arbiter must be used to arbitrate between DP8409 DRAM

refresh requests and MC68000 access requests. When a refresh must take place and the local processor wants to access the DRAM the arbiter does not use the MC68000 bus arbitration control circuitry, but instead will hold off the DTACK(L) signal from the DRAM subsystem while the refresh goes ahead. Thus the processor access is delayed in a manner similar to the master/slave slave bus arbitration already mentioned. Since the 128 rows of the DRAM require refreshing every 2 ms, a refresh cycle must take place approximately every 16 μsec. This means that if the local processor were constantly accessing the DRAM only 1 access in 41 would be delayed by a refresh cycle. Since in most cases the local processor will not be accessing the DRAM this frequently then it will only very rarely be delayed.

At present no error checking or correcting is carried out on the DRAM system either via parity or a Hamming code. However it is envisaged that in the future at least a parity bit should be included on each byte to improve detection of errors.

The 128K byte DRAM subsystem is more than adequate for current SMP needs, since at present all data tables used by the MCMs only take up approximately 30K bytes.

5.2.4 Global and Local Module Controllers

The *Local Module Controller (LMC)* is implemented using an MC68230 PI/T. It controls various functions on the MCM as well as being used to drive the map-select lines during C-bus accesses. Due to its powerful capabilities the LMC has protected access rights, being only accessible in supervisor mode.

The LMC can also be used to interrupt the local processor at the request of the SM or PG in order to gain the attention and services of the MCM. This is performed by altering one of the control lines on the GMC. Thus although an MCM is not a C-bus interrupt handler it can still be interrupted by external modules capable of accessing the control bus.

PIN	I/O	Function
PA7		unused.
PA6	O	CMA-bus lock.
PA5	O	I-bus lock.
PA4	O	Bus broadcast lockout (BBLCK).
PA3	O	Master processor interrupt, level 7 (MINT).
PA2	O	Module reset (MRESET).
PA1	O	Module processor halt (MHALT).
PA0	O	Block/Cycle by cycle transfer select (BURST).
PB7 - PB0		Unused.
PC7		unused.
PC6	I	PIACK function.
PC5	O	PIRQ function (C-bus interrupt request).
PC4		unused.
PC3	O	TOUT function.
PC2	I	TIN (62.5 KHz).
PC1		unused.
PC0		unused.
H4		unused.
H3		Interrupt input, service request from MCM.
H2		unused.
H1		Interrupt input, processor halted.

Figure 5.5 Global Module Controller Pin Assignments

PIN	I/O	Function
PA7 - PA0	O	Map-select lines, MA7 - MA0.
PB7	I	I-bus buffers empty.
PB6	O	I-bus request lock and reset.
PB5		unused.
PB4	O	C-bus A23 polarity.
PB3	O	Slave processor interrupt.
PB2	O	Slave reset.
PB1	O	Slave processor halt.
PB0	O	SM service request.
PC7	I	TIACK function.
PC6	I	PIACK function.
PC5	O	PIRQ function, interrupts local processor.
PC4		unused.
PC3	O	TOUT function, interrupts local processor.
PC2		unused.
PC1	I	CMA-bus buffers empty.
PC0	O	CMA-bus request lock and reset.
H4		Interrupt input, slave service request.
H3		Interrupt input, C-bus grant.
H2		Interrupt input, local failure.
H1		Interrupt input, Supervisor service request.

Figure 5.6 Local Module Controller Pin Assignments

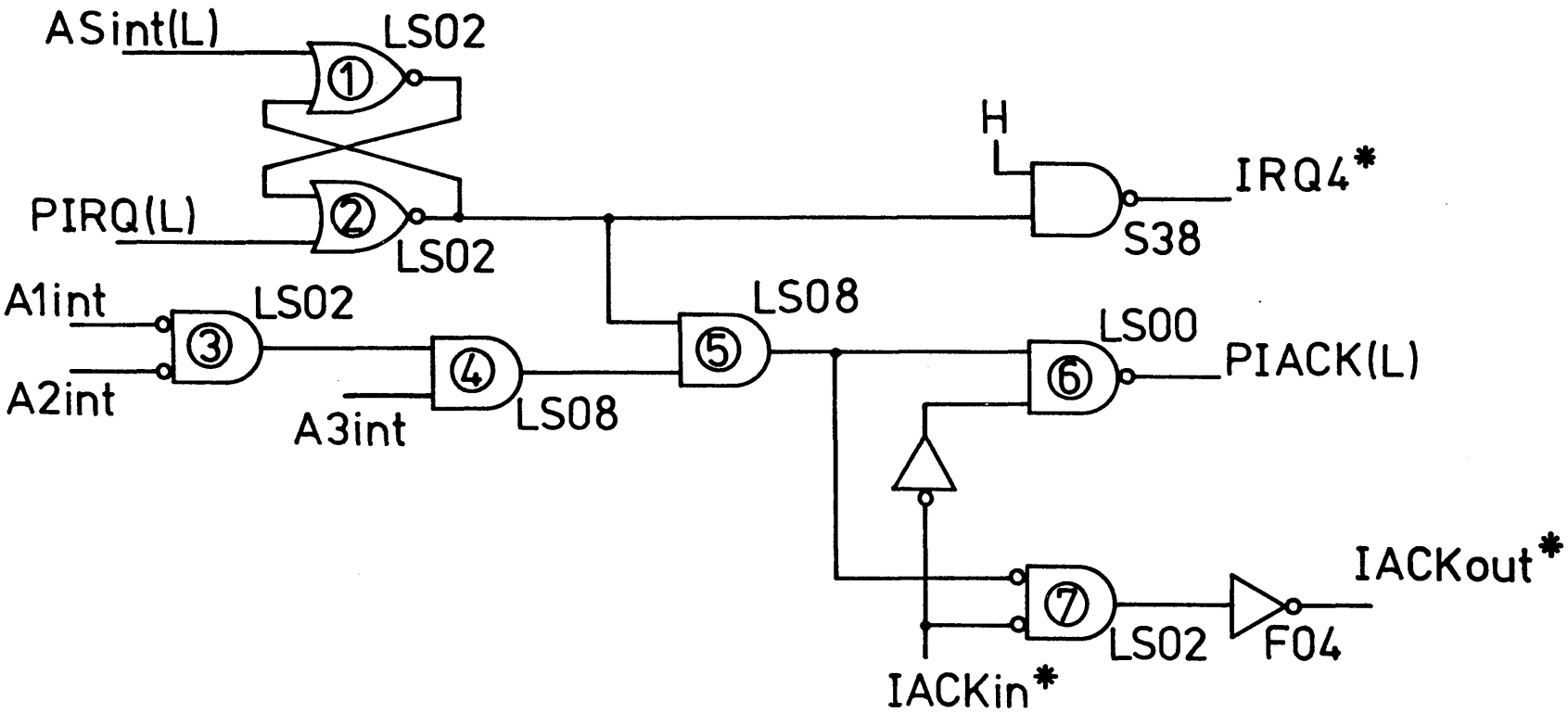


FIG. 5.7 C-BUS INTERRUPTER

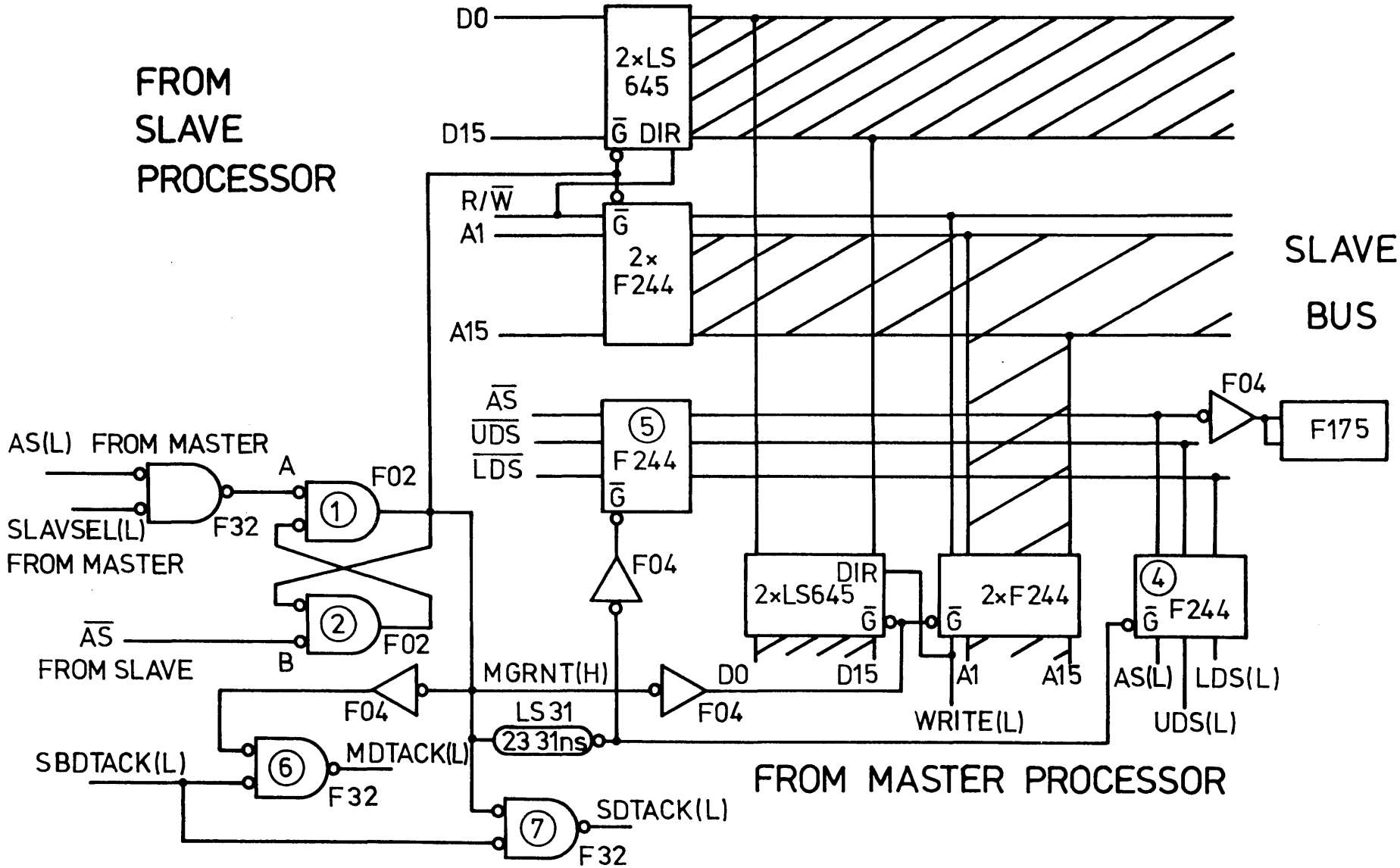


FIG. 5.8 SLAVE BUS

Interrupts can be sent to the local processor via the LMC for a number of other reasons, e.g. to warn of a local device failure (e.g. DRAM parity error), or as a request for attention from the slave processor. The timer on the LMC can also be used to generate interrupts to the local processor, although this is not required for shell-model processing. However this is a function which could prove useful if multi-tasking on the MCMs were ever envisaged. Current pin assignments for both the GMC and the LMC are shown in figures 5.5 and 5.6 respectively.

The MCM can request attention from the SM via the GMC. Thus the GMC is made a C-bus interrupter device on level 4. Interrupts to the SM from the GMCs can occur under two conditions, namely;

1/ a local processor halted condition occurring due to a double bus error for example,

2/ or an active request from the local processor via its LMC for attention from the SM.

Since each GMC will have a different set of interrupt vector numbers and since it modifies this number depending on which handshake line initiated the interrupt request, then the SM interrupt service routine will be aware of both the identity of the interrupting module and the cause. The vector numbers will themselves be supplied to the GMCs by the SM since it will be its task to initialise them. The MCM control bus must therefore be able to support interrupt acknowledge cycles from C-bus and so must monitor the IACK* daisy chain (figure 5.7). However since it is not intended that devices on the local bus should ever act as C-bus interrupters the local bus need not have this facility.

5.2.5 The Slave Bus

Details of the slave bus and its arbitration mechanism have already been given at the beginning of section 5.2. In this section we will go on to describe its hardware implementation (figure 5.8).

The key to the arbiter is the pair of NOR gates, 1 and 2. The slave bus appears as a 16K byte space within the masters local map and when it attempts to access this space, signalled by SLAVSEL(L) active, input A of, 1, is brought low. Since any cycle of the slave processor uses the slave bus then only the slaves address strobe, AS, need be used to bring input B low. The output of the F-F, MGRNT(H), is used to enable one of the two sets of buffers and will be in the low state, enabling the slave buffers, when neither processor is using the slave bus. The first processor to get its request into the F-F will win control of the bus by having its buffers enabled. However a delay is necessary in enabling the buffers which drive the data and address strobes in order to guarantee a setup time for the addresses. Therefore a delay of 23-31 ns, produced by an LS31, is introduced here.

The slave bus data transfer acknowledge, SBDTACK(L), which is timed in the same way as for the local bus, is sent to the appropriate processor by gates 6 and 7.

This arbitration mechanism is thus a simple, efficient method of allowing dual processor access to the slave bus, without unduly holding up either processor.

The slave processor subsystem itself is a much simplified version of the master processor system (section 5.2.1). The bus error and data transfer acknowledge signals are produced in much the same way, as are the interrupts. The reset and halt signals come from the master processors LMC, instead of the GMC, so that the master processor has complete control over the running of the slave. The slave is also reset whenever the master is.

The ***slave controller (SC)*** PI/T has a much more limited use than the LMC. Interrupts from the SC to the slave can come from four sources; the master processor, either of the FPUs signalling that they are finished processing, or the onboard timer. The master processors interrupt must be non-maskable and is therefore on level 7. The two FPU interrupts are

also on level 7, but can be disabled externally by the slave processor should it wish to ignore them. The PI/T timer interrupt is placed on level 4. Uses of the SC I/O port lines are limited to :

1/ I-bus buffers empty signal (input),

2/ I-bus requester reset and lock (output),

3/ CMA-bus buffers empty (input),

4/ CMA-bus requester reset and lock (output),

5/ Master processor attention request (output).

6/ FPU interrupt enable/disable (output).

We need not go into any detail concerning the slaves static RAM subsystem, which is implemented in the same manner as the master's using the Inmos IMS1420-55. However in the next section we will give some details concerning the FPUs.

5.2.6 The Floating-Point Units

The FPU used in MCMII is the National Semiconductor NS32081 (formerly the NS16081). This device is one of the slave processors to the NS32000 family of microprocessors, but can be used as a peripheral for other microprocessors [NS081]. It contains eight 32-bit internal data registers but only has a 16-bit external data bus. It can perform the normal arithmetic functions to 32 or 64-bit precision, i.e. single or double precision conforming to the IEEE standard, as well as format conversion instructions (e.g. integer to floating-point, single to double precision etc.) and floating-point comparisons. The NS32081 contains an internal floating-point status register (FSR) which is used to configure the FPU operational modes (e.g. IEEE rounding modes) and also records any exceptional conditions which were encountered during execution of an operation (e.g. underflow, inexact result etc.).

The 8 MHz part which is used is capable of performing a register to register multiplication in 6 μsec and a register to register addition in 9.4 μsec. Additional overheads are necessary for the MC68000 processor

to send the appropriate operation words and operands (if required) to the FPU at the start of each operation and to read back the FPU status and result (if there is one). If an error is detected by the FPU during operation then this will be signalled in the status word. It is then up to the processor to read the FSR from the FPU to determine more fully the nature of the error and take any necessary action.

At the time of designing MCMII the Motorola MC68881 floating point coprocessor was not available. Also early reports from Motorola indicated that it would not be possible to use the MC68881 as a peripheral to the MC68000 although this has now been changed. For these reasons the NS32081 device was chosen instead. However this proved to be more complicated and slower to interface to than at first was believed and although the device enhances the power of MCMII it is unsuitable, in terms of speed and ease of use, for the MC68000 based MCMII system.

5.3 MCM Task Look-up Tables

Having described the hardware of the MCMs we will now further elaborate on the main task which the MCMs perform and the data table resources that are available to them during normal shell-model processing. In H-mode operation the MCMs basic task involves the evaluation of the Hamiltonian matrix element $\langle e_m | H | e_n \rangle$ using one of equations 2.6-2.8 and then the evaluation of equations 2.12a and 2.12b by the following steps:

1/ Read a new TSW from the I-bus interface. Set-up the CMA-bus prefetch buffers to read from CM at the address given by the index, m, contained in the TSW.

2/ Using the annihilation and creation operators and the job-type bits, all contained in the TSW, as well as the prime state, determine the two-body matrix element and its sign. For zero and one jobs there will be more than one two-body element to determine, with each one

being added together to form the complete Hamiltonian element.

3/ Test to determine if the vector elements have arrived from CM. If so then read V_{im} and V_{fm} from the CMA-bus prefetch buffers. If not then wait until they have arrived. The L-bit for V_{fm} must be tested and if it is set then the element must be read again from CM (see section 4.4.1).

4/ Evaluate $V_{fm} = V_{in} \times H_{mn} + V_{fm}$ and

$V_{fn} = V_{im} \times H_{mn} + V_{fn}$.

5/ Write V_{fm} back to CM.

The task in 2 above of determining the Hamiltonian matrix element magnitude and sign is potentially very complex and demanding. In order to make this task easier for the MCMs a number of data tables are necessary. The tables required for these two stages will be discussed in the following two subsections.

5.3.1 The Matrix Element Magnitude

The value of each Hamiltonian matrix element, $\langle e_m|H|e_n\rangle$, is solely dependent on which particles are annihilated and which are created to transform the state $|e_n\rangle$ into the state $|e_m\rangle$ (eqn. 2.5). In a system of 24 active orbitals there would therefore be a potential maximum of $\left({}^{24}C_2\right)^2 = 76176$ two-body matrix elements, if there were no quantum mechanical constraints on the particles annihilated and created. Fortunately this is not the case and there are in fact two constraints on the annihilated and created particles which greatly reduces this number. These constraints are (for annihilated particles with indices k and l and created particles with indices i and j);

1/ **Isospin conservation**: if t(x) is the z-component of isospin of the single particle orbital with index x then we must have

$$t(i) + t(j) = t(k) + t(l) \qquad (5.1)$$

where t(x)= +1 or -1.

In other words, if two protons are destroyed then two protons must be

created, if two neutrons are destroyed then two neutrons must be created or if a proton and neutron are destroyed then a proton and neutron must be created.

2/ Angular momentum conservation: if m(x) is the z-component of angular momentum for the single particle orbital x then we must have

$$m(i) + m(j) = m(k) + m(l) \qquad (5.2)$$

These two constraints act to reduce the number of valid operator quadruples (k,l,i,j), and therefore the number of two-body matrix elements, to just 4196. Thus it is quite practical that all these elements should be stored as single precision floating-point numbers in local MCM memory in a ***Matrix Element Table (MET)***. The task of the MCM is then simplified to one of looking up the MET to find the appropriate element(s) with which to form the Hamiltonian entry $\langle e_m | H | e_n \rangle$. The MET must therefore be constructed in such a manner as to make the process of referencing it efficient. This process is performed using only the annihilation and creation operators since it is they which uniquely specify each two-body element.

The simplest and quickest method of referencing the MET would be to use the annihilation and creation operators directly to index into it. However this would necessitate that the MET be prohibitively large since it would contain mostly null and duplicated entries. Therefore in order to keep the MET to 4196 entries a system of look-up tables indexed by the four operators must be used.

To explain the structure of the MET we first define a set, S, containing all valid operator pairs (x,y), with $x < y$, and partition it by defining an equivalence relation on it. For the sd shell this set will have $^{24}C_2 = 276$ entries. We define the equivalence relation such that, for (i,j) and (k,l) both members of S then:

$$(i,j) \sim (k,l) \qquad (5.3)$$

if and only if $m(i) + m(j) = m(k) + m(l)$

and $t(i) + t(j) = t(k) + t(l)$

Example partition ;

S(M,t) = [(a,b) , (c,d) , (e,f) , (g,h)]

Matrix Element Table Structure

B(a,b)	(a , b , a , b) (c , d , a , b) (e , f , a , b) (g , h , a , b)
B(c,d)	(a , b , c , d) (c , d , c , d) (e , f , c , d) (g , h , c , d)
B(e,f)	(a , b , e , f) (c , d , e , f) (e , f , e , f) (g , h , e , f)
B(g,h)	(a , b , g , h) (c , d , g , h) (e , f , g , h) (g , h , g , h)

where the operator quadruple (a,b,c,d) represents the two-body matrix element H_{abcd}

Figure 5.9 Example of Blocks within MET

We have thus divided S into partitions, which we denote S(M,t), where;

$$M = m(i) + m(j) \quad \text{and} \quad t = t(i) + t(j) \tag{5.4}$$

i.e. t is either proton-proton (p-p), neutron-neutron (n-n), or proton-neutron (p-n). Therefore all pairs (x,y) in S fall into one and only one partition and so S is completely and uniquely partitioned by this equivalence relation.

The significance of these partitions lies in the fact that any pair of operators, (k,l), taken from a partition S(M,t) can only be joined with other operators, (i,j), within the same S(M,t) to form valid operator quadruples (k,l,i,j) which specify two-body matrix elements. In fact each of the operator pairs within S(M,t) will join with, and only with, all the pairs in S(M,t) to form these quadruples. Therefore if there are n pairs of operators in a partition then that partition will define n^2 two-body matrix elements.

We now define an order on the pairs in S(M,t) as follows:

for (x,y) and (x',y') both members of S(M,t)

$$(x,y) < (x',y') \iff 1)\ y < y' \quad \text{or} \tag{5.5}$$

$$2)\ y = y' \quad \text{and} \quad x < x' \ .$$

The MET can now be divided up into blocks, where all the two-body matrix elements in a block are specified by the same pair of annihilation operators (k,l) in the quadruple. These blocks are denoted B(k,l). All the two-body matrix elements in B(k,l) are then specified by taking each pair of operators in the partition S(M,t), where (k,l) belongs to the partition S(M,t), and forming a quadruple of operators. The elements in B(k,l) are given the same order as the pairs in S(M,t), figure 5.9.

Thus all the blocks in the MET whose annihilation operators (k,l) belong to the same partition S(M,t) will have the same set of creation operators (i.e. all those operators in S(M,t) itself) and so have the same number and order of elements. For example consider two blocks B(a,b) and B(e,f) in the MET, where (a,b) and (e,f) both belong to S(M,t), and consider another pair of operators (c,d) and (g,h) also in

S(M,t) (see figure 5.9),

i.e. $m(a)+m(b) = m(c)+m(d) = m(e)+m(f) = m(g)+m(h) = M$

and $t(a)+t(b) = t(c)+t(d) = t(e)+t(f) = t(g)+t(h) = t.$

That is if (c,d) and (g,h) are the 2nd and 4th pair in S(M,t) respectively then the quadruples

(c,d,a,b) and (g,h,a,b)

will define the 2nd and 4th two-body matrix elements in B(a,b) while the quadruples

(c,d,e,f) and (g,h,e,f)

will define the 2nd and 4th two-body matrix elements in B(e,f).

Therefore for any two-body matrix element its annihilation operators (k,l) can be used to specify the block within the MET where the element resides and its creation operators (i,j) can then specify its position in the block.

A number of auxiliary tables are now needed in the task of finding a matrix element in the MET using its annihilation and creation operators. These tables are;

1/ <u>The Global Offset Table (GOT)</u>: this table contains one entry for every pair of annihilation operators possible in the 32-bit word of the MFG and therefore has $^{32}C_2 = 496$ entries. Since there are only 24 active orbitals only 276 of these entries are valid and so almost half of the space in the GOT is unused. Each of the (valid) entries in the GOT contains a 16-bit address offset from the base of the MET to the start of a different block within the MET i.e. the entry for (k,l) in the GOT contains the offset to the block B(k,l) in the MET. The entries in the GOT are ordered according to equation 5.5, i.e. in the same way as the entries in S(M,t), and so the 2-byte entry for a pair (k,l) can be referenced by;

$$2[k + l(l-1)/2]$$
$$= 2k + l(l-1) \qquad (5.6)$$

where k and l are the annihilation operators (k < l).

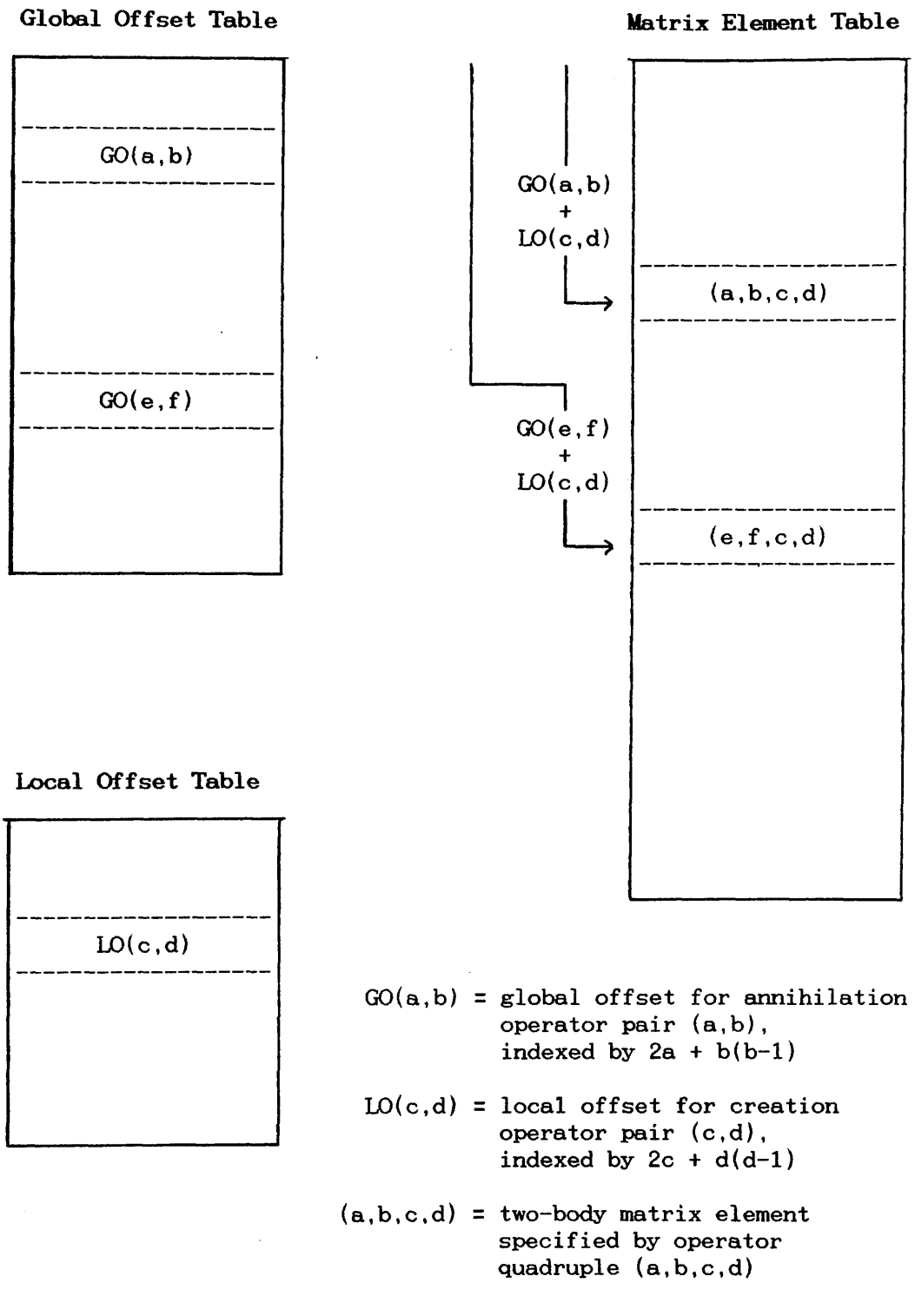


Figure 5.10 Global and Local Offset Tables

2/ The Local Offset Table (LOT): this table contains one entry for each pair of possible creation operators and so has the same number of entries as the GOT. Each entry in the LOT is a 16-bit address offset from the base of a block in the MET to the two-body matrix element specified by the the creation operators, figure 5.10. The creation operators are used with equation 5.6 to determine the 2-byte entry in the LOT to use. When the entry from the LOT is added to the entry from the GOT then the complete offset from the base of the MET to the vector element specified by the 4 operators is formed.

3/ The O-table: in order to aid in the evaluation of the offsets into both the Global and Local Offset tables using equation 5.6, a further table is added to give the value of x(x-1) for x equal 0 up to 31. Each entry for this table is 2-bytes and the entry at offset 2x from the base of the table gives the value of x(x-1).

The memory usage for these three tables plus the MET itself is

MET	4196 entries @ 4 bytes each =	16,784 bytes,
GOT	496 entries @ 2 bytes each =	992 bytes,
LOT	496 entries @ 2 bytes each =	992 bytes,
O-table	32 entries @ 2 bytes each =	64 bytes,

giving a total of 18,832 bytes (18.39K bytes).

It should be noted that the size of the MET can be reduced even further at no extra cost to the look-up process. This is achieved by first noting that the magnitude of each two-body matrix element remains unchanged by a transformation of neutron orbitals to proton orbitals (and vice-versa) with the same quantum numbers (n,l,m). Thus each block B(k,l) is the same as B(k',l'), (i.e. has the same two-body matrix elements in the same order), where

$$k' = (k+16) \bmod 16 \quad \text{and} \quad l' = (l+16) \bmod 16$$

in the representation given in figure 3.1.

Thus any neutron-neutron (n-n) block in the MET has its equivalent proton-proton (p-p) block and thus the GOT can map all p-p operators

onto their equivalent n-n blocks in the MET, or vice-versa. The p-p (or n-n) blocks can therefore be removed from the MET, thus reducing its size by 640 elements (= 15.25%).

The proton-neutron cases can also be reduced if the GOT maps (k,1), not onto B(k,1), but onto its equivalent block B(k',1') and so B(k,1) can also be removed from the MET. This removes a further 1362 entries in the MET, giving an overall reduction of 2002 entries (= 47.7%).

Other methods could be used to further reduce the size of the MET, however they would necessitate more complex look-up processes and are thus not considered. However although the above modification gives a fairly substantial reduction in the MET size at no extra cost to the MCM look-up process this has not been implemented for reasons which shall be explained later.

Clearly then there is a trade-off between the size of the MET and the ease with which it is referenced, with the most efficient method for finding an entry requiring far too much memory space for the MET. The final method must therefore be a compromise between speed and memory usage with speed being the greatest requirement, which the above solution offers.

5.3.2 The Matrix Element Sign

Before the final Hamiltonian matrix element, $\langle e_m|H|e_n\rangle$, is complete the sign of the two-body matrix element must be altered by the factor

$$(-1)^{[\sum(i+1,j-1) + \sum(k+1,l-1)]}$$

as given in equations 2.7 and 2.8. The power of -1 in this equation is simply the sum of the number of set bits (i.e. occupied orbitals) between k and l and between i and j in the slater determinant $R = a_k a_l |e_n\rangle$.

In order to determine this number we must first form R and then strip off the unwanted bits leaving only those set bits which are to be counted or in effect whose parity is to be determined. Forming R is

performed using the MC68000 bit-clear command, however stripping off the unwanted bits using this method would be time consuming. A simpler method is to have a table of 32-bit masks for each pair of operators (x,y), which consists of all zeros except for the bits between x and y. These masks are held in the ***Mask Table*** which is organised along the same lines as the GOT and LOT with 496 entries indexed by the operator pair using the function given in equation 5.6, although since the entries in the Mask table are 4 bytes instead of 2 the index gained from 5.6 must be doubled.

Two masks are retrieved from the Mask table, one for (i,j) and one for (k,l). Set bits which are common to both masks are eliminated, since they would otherwise have to be counted twice. To do this the masks are first XORed together and the resultant composite mask is then ANDed with R to leave only those set bits necessary.

The above method is the one currently used in producing the composite mask, however there is another method which is quicker although it requires the Mask table to be much larger. This other method comes as a result of noting that each composite mask is completely determined by the operator quadruple (k,l,i,j) just as the two-body matrix elements are. Therefore the Mask table could instead contain the 4196 possible composite masks, specified by the (k,l,i,j), and be referenced using the same index as used for retrieving the two-body matrix element from the MET. The Mask table would then be almost 9 time larger and could be incorporated into the MET with each entry containing a two-body matrix element and composite mask. The sign determination process would then be shortened since the composite mask would not have to be manufactured. It is for this reason that the MET is not reduced in size as described earlier so that in future the MET and Mask table can be referenced together.

Under normal operating conditions these increases in table size would be minimal compared to the 128K bytes available to the MCM at

present. However since there is no CM yet implemented, the initial and final vectors normally stored there must instead be stored locally on each MCM thus making space requirements more important. Therefore this improvement in manufacturing the composite mask has not been implemented in order to save local memory space for the storage of vectors.

The resultant word formed after R is ANDed with the composite mask must then have its parity determined and to do this we make use of the following result:

$$P(M1:M2) = P(M1 \oplus M2)$$

where P(M) is the parity of binary word M, M1 and M2 are words of equal length and M1:M2 is the word formed by the concatenation of M1 and M2. Thus to determine the parity of the 32-bit word we first XOR the two 16-bit words and then XOR the two bytes of the resultant word. The parity of the remaining byte, which equals the parity of the initial 32-bit word, is then found from a *Parity Table*, which is indexed directly by the byte. This 256 entry table gives at location x the parity of x, i.e. if the byte at location x is zero then x has an even number of bits otherwise x has an odd number of bits. Thus using the Mask table and Parity table the sign change for the two-body matrix elements can be determined.

The space requirement for these two tables is :

Mask Table -- 496 entries @ 4 bytes each = 1,984 bytes,

Parity Table -- 256 entries @ 1 bytes each = 256 bytes,

giving a total size of 2,240 bytes for these two tables and 21,072 bytes for all six tables. If the proposed changes to the Mask table were implemented it would be 16,784 bytes long, bringing the total figure to 35,872 bytes.

5.4 MCM Task Processing

All the hardware and software resources of the MCMs have now been

discussed and thus the foundations have been laid so that we can now describe the actual process, in terms of software, which the MCMs must follow through to perform a Lanczos iteration. All the software for the MCMs is written in MC68000 assembly language since much of the process involves manipulation of binary data which is less suited to high level languages. However more importantly than this assembly language can be tailored to meet high-performance requirements, which is a high priority for the MCM's task.

For the successful processing of an iteration by the MCMs a number of global software flags are required, in order that both the SM and PG can signal certain system-wide conditions to the MCMs. These conditions are:

1/ New Prime State: when there is a change in the prime state which is associated with the TSWs read from the MFG Buffer, the PG must inform the MCMs of the new prime state SD word and its index.

When the last TSW for a prime state is read from the MFG Buffer by the MCMs, the PG will be interrupted and the Buffer will be blocked by its onboard inhibit logic from any more reads (sec. 3.5.6). At this point the PG must send the new prime state details to a predetermined parameter passing area on the MCMs, using the C-bus bus-broadcast facility. The PG must then send a signal to the MCMs to inform them of the update and then remove the read inhibit from the MFG Buffer. However each individual MCM must be inhibited from reading from the Buffer until it has recognised that there is a new prime state. This is essential so that when an MCM reads a TSW from its PFB it knows which prime state the TSW belongs to. To this end the PG must also activate the local I-bus reset and lock signal on the LMC of each of the MCMs when it sends the new prime state details and before it enables the MFG buffer again. Thus if there is a TSW in an MCM's PFB after the I-bus lock has been set then it cannot refer to the new prime state. It is in fact this lock being activated that is used to

signal the presence of a new prime to the MCMs.

After finishing each task the MCM will test if its I-bus PFB is full or empty and if it is full then it will process the TSW as normal. It is quite possible that a new prime state has already been passed to the MCM at this point, however if there is a TSW in the PFB then it must belong to the old prime and so no action is taken by the MCM with regard to the new prime. Only if the PFB is empty does the MCM then test the lock signal in the LMC and if it has been activated by the PG's broadcast then the new prime state details are read into the appropriate workspace locations from the parameter passing area.

2/ Iteration start: after the MCM is released from the reset signal by the SM it will perform certain initialisation functions, e.g. set up the LMC. Once these tasks have been finished the MCM must wait for a start signal from the SM before reading from the MFG Buffer and commencing TSW processing. The MCM must also wait for this signal after finishing an iteration and before going on with the next. A word within the MCMs workspace is reserved for this and other signals from the SM, these signals being collectively called the ***global module code word (GMCODE)***.

3/ Iteration finished: when the PG finishes generating the basis list and the MFG Buffer empties, the PG will signal this to the SM who will in turn signal to the MCMs (via the GMCODE) that there are no more TSWs to be processed and so the iteration is finished.

The master processor on MCMII must also be able to give commands to the slave processor and so a location, the ***slave instruction code word (SICODE)***, is reserved in the slave's workspace for this purpose. These local commands are given to the slave in association with data that it is to process, e.g. two-body matrix elements for the slave to add together in zero or one job processing. Once the slave has read the data it will signal this to the master, via SICODE, and thus allow the master to pass more data.

In response to each of the above global signals the MCM will enter a different software routine. Similarly after reading a TSW from the I-bus PFB the MCM will enter one of three different routines depending on whether a zero, one or two job is indicated by the job-type bits. The details of these latter routines are now given in order to describe in more detail the task of the MCMs and explain their method of operation. Note that we describe here the routines assuming the presence of CM and two FPUs per slave processor, the actual routines currently implemented are therefore different since there is no CM and only one FPU on MCMII, we will however describe the differences this makes later.

5.4.1 Two-job Processing

Appendices A and B give listings of the two-job routines for the current and final versions of MCMII respectively.

Master processor:

Once the master processor has determined that the I-bus PFB is not empty it will read the first 4 bytes, which contain the job-type bits and secondary index, m, and store them in its workspace. The job-type bits, having been examined, are stripped off leaving only the index m. The 4 operators are then read from the PFB and stored.

In order to allow recovery from MCM errors, it is important that each TSW is stored in full in the MCM's workspace and not overwritten until the master has finished processing it. Thus in the event of a fatal error occurring on the master another MCM, or possibly the SM, can process the TSW and so the Hamiltonian associated with it is not lost.

Next the master uses the operators to fetch the offsets from the GOT and LOT. These are then added to form the offset into the MET and so the two-body matrix element can be retrieved. The sign change for the two-body element is then determined using the Mask table and Parity table, as previously described.

The master then tests SICODE to determine if the slave has read the

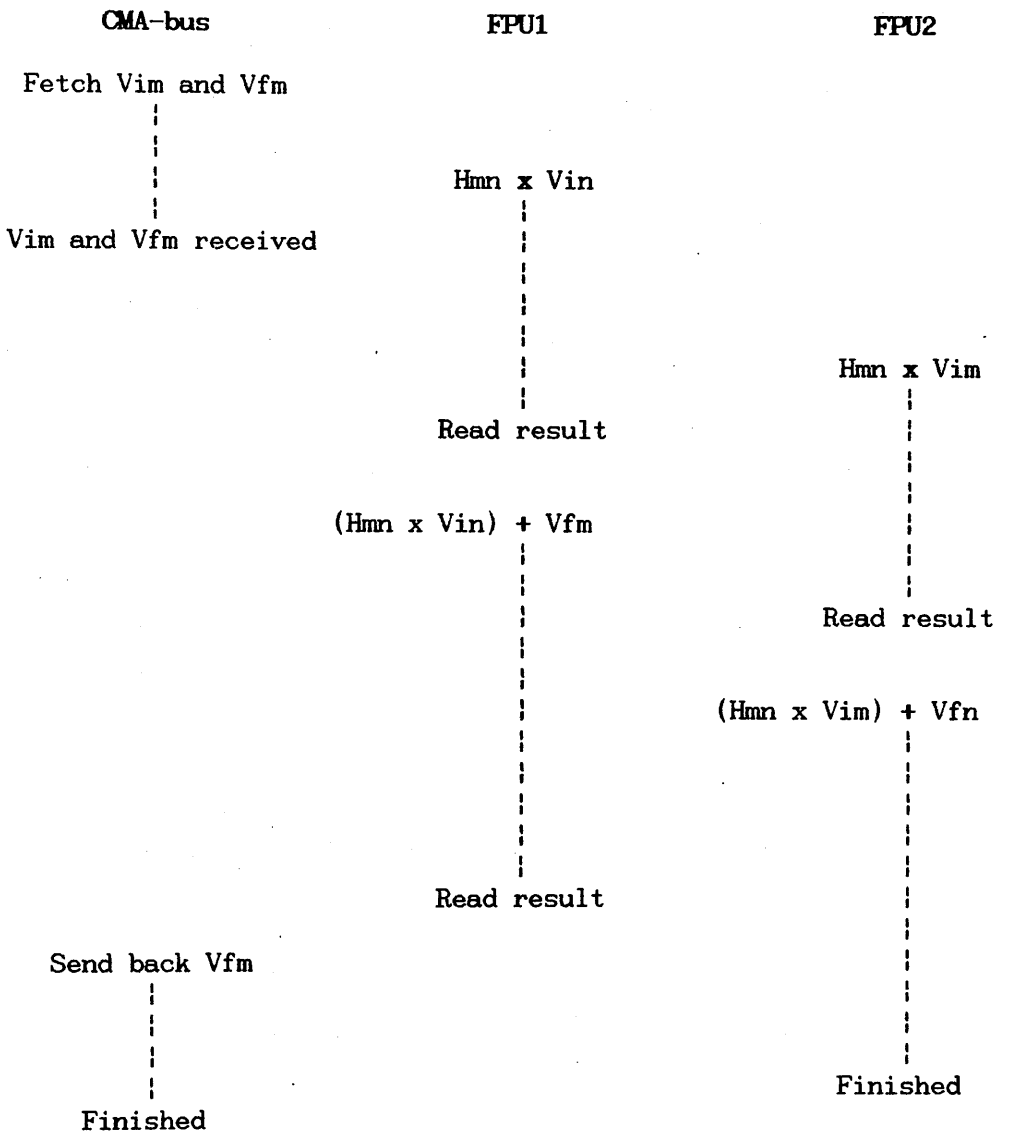


Figure 5.11 Concurrency During Slave Two-job Processing

last data which was sent. If the slave has then the parameter passing area in the slave's workspace is free to place new data in, otherwise the master must wait until the slave reads the data currently in it. When the parameter passing area is free then the master writes the Hamiltonian element just formed and the secondary index, m, into it and places the appropriate code (i.e. one which tells the slave that the data is for a two-job) in SICODE. The master then tests to see if there is a new TSW in the I-bus PFB.

Slave Processor:

Just as the master processor has its tasks initiated by data present in the I-bus PFB, so the slave has its initiated by a valid code word in SICODE and the associated data. The slave reads the code word and then branches to the appropriate routine.

Upon receiving the two-job code the slave will transfer H_{mn} and m from the parameter passing area into its local workspace and then signal to the master, via SICODE, that it has read the data. This then frees the parameter passing locations for more data. The slave will then immediately set off the CMA-bus PFB to fetch the initial and final vector elements indexed by m.

The first multiplication (H_{mn} x V_{in}) can then be started in FPU1 (V_{in} will be stored in one of the internal registers of FPU1). By this time V_{im} and V_{fm} should have arrived, i.e within 4 usec of setting off the CMA-bus PFB. The L-bit for V_{fm} is checked to determine if the value received is valid. If not then another request is issued to fetch the data from CM. The second multiplication (H_{mn} x V_{im}) can be started in FPU2, even if V_{fm} has not arrived. The result of the first multiplication (when it is available) can then be added to V_{fm} (when it is available), in FPU1. Then the result of the second multiplication can be added to V_{fn}, in FPU2 (V_{fn} is stored in one of the internal registers of FPU2). The new V_{fm} can then be returned to CM. Figure 5.11 details the concurrency of operation for the slave executing a two-job.

5.4.2 One-job Processing

Master Processor:

In this case there is only one annihilation operator, k, and one creation operator, i, held in the TSW. The annihilation operator is used to form the slater determinant $a_k|en\rangle$ and this determinant is then searched for all remaining occupied orbitals i.e. set bits. For each set bit that is found its index is used as the other annihilation and creation operator index so that a quadruple, (k,l,i,j) with l = j, is formed. Both operator pairs must then be ordered so that k < l and i < j and the quadruple is then used to fetch a two-body matrix element from the MET and determine the sign change as for the two-job case above.

Each element thus found is passed to the slave with the appropriate one-job code. When all the elements have been found the master sends the index m of the secondary state and the one-job termination code.

Slave Processor:

In this case the slave will take every two-body matrix element passed to it by the master and sum them in one of the FPUs. The slave again uses SICODE to signal to the master when it has read each of the elements. When the one-job termination code is received the slave then proceeds exactly as for the two-job case above.

5.4.3 Zero-job Processing

Master processor:

For a zero job no valid operators are present in the TSW and instead the master must search the slater determinant, $|e_n\rangle$, for all possible annihilation operator pairs (k,l), i.e. for all possible pairs of set bits. For each pair found the creation operator pair is set equal to it, so that k = i and l = j, and the resulting quadruple is then used to fetch the appropriate two-body matrix element from the MET. However this time in accordance with equation 2.6 no sign change need be

determined.

Each element found is sent to the slave with the zero-job code and when all elements have been found the zero-job termination code is sent. Note that in this case no index need be sent, since for a zero-job m = n and the slave already possesses the index n.

Slave processor:

Each two-body matrix element sent to the slave is added together, as in the one-job case, to form the final Hamiltonian entry. However since m = n only the new V_{fn} need be evaluated using

$$V_{fn} = H_{mn} \times V_{im} + V_{fn}$$

and V_{im} need not be fetched from CM since it is equal to V_{in}, therefore no references need be made to CM. The new value for V_{fn} is still held in an internal register of FPU2 and is not sent back to CM until later.

5.4.4 New Prime State Processing

Master processor:

When the master recognises that a new prime state has been received, by detecting that the I-bus lock signal in its LMC is active, it will first remove the signal and then transfer the new prime state and index into his workspace overwriting the old details. The master will then activate a new request to read from the MFG Buffer. If this request is not granted then it will again test to determine if another new prime state has been sent and if one has then it will repeat the above process. In this way the MCM can guarantee that no updates are lost.

When the PFB is filled with a new TSW the master will first write the new prime code to the slave along with the new prime state index and then proceed as usual by examining the job-type bits of the new TSW and branching to the relevant routine.

Slave processor:

Upon receiving the new prime code, the slave will make a copy of the old prime index and then transfer the new prime index to its workspace. It

will then activate its CMA-bus PFB to fetch V_{fn}, where n is the old prime index, so that its own local copy of V_{fn} can be added to it. Since each MCM must perform this operation and only one of them can hold a copy of V_{fn} at one time then they must all check the L-bit and wait in turn to receive V_{fn} from CM. When the slave does receive it, it adds on its local copy and then sends the result back to CM.

The slave then requests V_{in} from CM, where n is the new prime state index, (performing only a half-word read on CMA-bus). On receiving it the slave places it in an internal register of FPU1. The local copy of V_{fn}, held within an internal register of FPU2, is then zeroed.

5.4.5 Current Implementation

As we have said there is currently no CM installed in the SMP system and so the initial and final vector elements are stored locally on each MCM. However since they are stored in the master processor's DRAM the slave processor does not have direct access to them. Instead each time the master passes the slave an index number it must also pass the appropriate initial and final vector elements. For example during a two-job when the master passes H_{mn} and m it must also pass V_{im} and V_{fm} to the slave. Also at the start of prime state processing when the master passes the new index n to the slave it must also pass V_{fn} for the old index and V_{in} for the new index.

Similarly the slave must also pass the updated final vector entry to the master at the end of each job and at the end of each prime. With each of these elements the slave must also pass its index so that the master can store them back in the correct location in the final vector. To deal with this additional parameter passing locations are allocated within the slaves workspace to hold the extra parameters which are to be transferred. An additional code word, similar in nature to SICODE, is also assigned for the slave to signal to the master the nature of the data and for the master to signal to the slave that it has read the

data.

Since data is being transferred in both directions care must be taken to avoid deadlock between the two processors. That is the situation could arise where the master processor is waiting to send data to the slave but can't since the parameter passing area is full and similarly the slave is waiting to send data to the master. Therefore when the master or slave send any data they must also always check to determine if they have received any data and if so then read it.

At present the slave processor has only one FPU instead of the proposed two. This simply means that the none of the arithmetic operations to be performed by the slave can be pipelined. Instead they must be performed serially, therefore increasing the overall time taken by the slave.

5.5 Vector Processing

The task so far described of processing the TSWs produced by the MFG and thus producing a resultant vector is the main task of the MMPU but not its only one. The vector produced by the multiplication of the Hamiltonian matrix by the Lanczos vector must be converted into the next Lanczos vector in the sequence and then orthogonalised with respect to all the other Lanczos vectors (section 2.3). These operations demand the addition of two vectors and also the scalar multiplication of two vectors. For both these operations the two vectors will be stored (as usual) in CM in single precision floating-point format arranged so that two elements, one from each vector, can be read during one CMA-bus cycle.

For the addition of two vectors the SM can block partition the two vectors and then assign each MCM a block to add together. Each addition should take at most 20 usecs, with the writing and reading of vector elements to and from CM being pipelined with the FPU operation. Thus for

the largest vector of 93710 elements, 5 MCMs could perform an addition in 0.4 seconds.

The vectors can again be block partitioned for scalar multiplication with each MCM accumulating a partial result, which are all added together to form the final result. To reduce the accumulation of precision errors during a scalar multiplication the accumulations must be carried out to double precision, although the final result can be reduced to single precision. Each multiplication and addition should take at most 25 usec, with both FPUs being used by the slave and again the fetching of operands from CM being pipelined with slave activity. Thus scalar multiplication should take at most a total time of 0.5 seconds for 5 MCMs operating on the largest vectors.

Having now described the MCMs completely we can go on in Chapter 6 to give the details of performance achieved by the SMP system.

CHAPTER 6

Shell Model Processor Performance

6.0 SMP System Testing

With the completion of MCMI hardware and the MFG hardware and software in 1984 it was then possible to run and test the performance of the MFG subsystem on its own. Testing the correctness of operation of the MFG was done in two ways. The first was in essence simply to use a set of test vectors. That is with the channel memories of the SG loaded up with sample SD-byte chains a number of predetermined seed states, or test vectors, were sent to it. The resultant output could then be precalculated and compared with the actual output, which was read from the MFG buffer by MCMI. This proved a successful and useful means of testing and debugging the MFG hardware, i.e. the SG, PF, buffer and I-bus, and could eventually be incorporated into the SMP system as a means of self-testing.

The second test was to run the complete MFG system, software and hardware, using data for real nuclei. The Nuclear Theory Group at Glasgow University then supplied figures for the number of zero, one and two-jobs that should be found. MCMI was then used to read from the MFG buffer and to count the number of each type of job. The two sets of figures were then compared. This method provided a means of testing the MFG software and hardware system as a whole. By successfully passing both these tests the correct operation of the MFG could then be guaranteed with a very high degree of certainty.

The hardware and software for MCMII, along with the software for

Nucleus	Number of Basis States	T1	T2	T3	T4	Number of states produced by SG for T3.
28 Si 14 m = 0 Np = 6 Nn = 6	93,710	3	117	436	396	3.4833×10^{9}
27 Al 13 m = 5/2 Np = 5 Nn = 6	64,299	<2	66	214	197	1.66618×10^{9}

All timings in seconds.

T1 = time taken with no SG driver & no interrupts i.e. time taken to produce the basis list and associated driver tables.

T2 = time with no interrupts and no waiting for SG seed requests i.e. seeding SG as fast as possible.

T3 = time with no H-mode interrupt.

T4 = total time per iteration for the MFG.

All the above times are for the MFG in H-mode, with the MMPU inactive and output from the PF ignored.

Table 6.1 MFG Measured Iteration Times

both MCMI and the SM, were all finished by summer 1985. The SMP system, minus the CM, could then be tested and evaluated. Using the DRAM of the MCMs in place of the CM, nuclei with up to 13,500 basis states could be tested. The Nuclear Theory Group supplied the resultant vectors after one iteration for a number of small nuclei. These nuclei were then run on the SMP system and the results compared. On all the nuclei which were tested the results were in complete agreement.

6.1 MFG Performance

Table 6.1 details certain timing characteristics for the MFG alone. All these timings were taken with the MFG in H-mode and clocked at 112 MHz. The MCMs were inactive, with the MFG buffer simply being allowed to overflow.

The first two timings give an indication of the PG performance. The first figure shows the speed at which the PG can generate the basis of states and the SG driver tables, i.e. run the Basis Generation Function and SG Control Function but without the SG Driver routine. The second figure gives the time for the full PG task including the SG Driver routine, i.e. the sending all the seed states to the SG, but assumes the SG is fast enough to keep up. It is thus obvious that the majority of the PG's time is taken up with seeding and controlling the SG.

The third timing figure is for the MFG as a whole performing an iteration, but without the H-mode interrupts generated by the SIC. The SG would thus produce too many states since some would go beyond the diagonal element of the matrix. Running in this mode it was possible to count the number of states produced by the SG using the SIC. The final column shows how many states were produced by the SG for this third case. From this figure we can calculate the amount of time the SG should have taken, in theory, under those conditions for one iteration with these nuclei. For example, at 112 MHz, producing 1 new state every 13

clock cycles, the amount of time to produce 3.4833×10^9 states for the ^{28}Si nucleus should be approximately 404 seconds. Comparing with the measured time of 436 seconds this shows an overhead of 32 seconds, i.e. an extra 8%. Similarly for the ^{27}Al nucleus the overhead is approximately 11%. This overhead is due to the time that the SG has to wait to be serviced by the PG. It is to be expected that the overheads due to PG servicing will increase for smaller nuclei, since M-partitions will in general be smaller and the SG will thus spend more time waiting for seed states.

The last timing figure is with the H-mode interrupts enabled and thus gives a true time for a complete iteration by the MFG as a whole. This last set of figures therefore represents a lower limit on the iteration time for the SMP system.

We can also from the last three figures obtain an estimate for the saving produced by searching only connected N-partitions (section 3.2). For the ^{28}Si nucleus the final iteration time (T4) is 91% of figure T3. It can therefore be assumed that approximately only 91% of the states counted for T3 are actually produced once the H-mode interrupt is active. That is 3.1698×10^9 states are generated compared with a possible maximum of 4.39×10^9 for half the matrix, a saving of almost 28%. For the ^{27}Al nucleus the saving is almost 26%. It is to be expected that the smaller the nucleus then the smaller the saving, since in general there will be fewer N-partitions and so proportionately less of the nucleus will be excluded from the search.

6.2 MCMII Performance

Processing two-jobs is by far the most common task for the MCMs during any iteration, making up between 88% (for ^{18}F m=0) to 96% (for ^{28}Si m=0) of the total number of tasks processed. Therefore the time taken to process a two-job will in general be the most predominant in determining

the overall time for the MCMs to complete an iteration.

With the current implementation of MCMII, i.e. a single FPU and using local RAM to store initial and final vectors, the program code for a two-job on the master processor should take approximately 46 μsecs to run and 55.5 μsecs on the slave processor, allowing time for the FPU (see Appendix A for current two-job listing). This implies that an MCM should process a two-job in about 55.5 μsecs with the master processor being delayed by the slave. The total time for a one-job or zero-job will depend on the number of two-body matrix elements to be found and added together, which will in turn depend on the number of occupied orbitals and therefore on the nucleus under consideration. Examination of the program code for zero-jobs and one-jobs shows that currently MCMII processes each two-body matrix element in approximately 18 μsecs and 38 μsecs respectively. We can use these figures to estimate iteration times as follows:

For the ^{31}P nucleus (m=11/2) with 8 protons and 7 neutrons there are;

13,327 zero-jobs with 105 two-body matrix elements to be found and added for each,

52,091 one-jobs with 14 two-body matrix elements to be found and added for each,

1,174,180 two-jobs.

The current MCMII will thus take

13327 x 105 x 18 μsecs = 25.2 secs for zero-job processing,

52091 x 14 x 38 μsecs = 27.7 secs for one-job processing,

1174180 x 55.5 μsecs = 65.2 secs for two-job processing.

Giving a total estimated time of 118.1 seconds for a complete iteration.

For the ^{24}Mg (m=10/2) nucleus with 4 protons and 4 neutrons there are;

10,026 zero-jobs (28 two-body elements each),

37,758 one-jobs (7 two-body elements each),

829,643 two-jobs.

Estimated time is therefore

Nucleus	Estimated MCMII Time	Measured MCMII Time	Measured MCMI Time	Measured Combined MCMI + MCMII Time
^{31}P m=11/2	118.1	114	755	99
^{24}Mg m=10/2	61.1	62	427	54
^{23}Ne m=1/2	33.6	34	238	30

All figures for H-mode operation.

Table 6.2 MCM Timings For Sample Nuclei

5.1 secs for zero-job processing,

10 secs for one-job processing,

46 secs for two-job processing,

giving a total of 61.1 seconds for a complete iteration.

And for the ^{23}Ne (m=1/2) nucleus with 2 protons and 5 neutrons there are;

6,457 zero-jobs (21 two-body elements each),

22,166 one-jobs (6 two-body elements each),

469,974 two-jobs.

Estimated time is therefore;

2.4 secs for zero-job processing,

5.1 secs for one-job processing,

26.1 secs for two-job processing,

giving a total of 33.6 seconds for a complete iteration.

The actual measured times for a complete iteration with these nuclei, using only the current version of MCMII, are;

^{31}P 114 seconds,

^{24}Mg 62 seconds,

^{23}Ne 34 seconds,

with the above estimates giving very good agreement.

Table 6.2 summarises these figures and adds additional figures for measured iteration times using MCMI on its own and then using MCMI and MCMII together. It can be seen that currently MCMII is approximately 7 times faster than MCMI.

The value of producing and verifying these estimates lies in our ability now to extrapolate forward and make estimates for the time the final MCMII will take. With the addition of CM and a second FPU, as well as implementing the changes to the software look-up tables already mentioned, we can expect the two-job processing time to be reduced to approximately 29 μsecs for the master processor and 37 μsecs for the slave processor (see Appendix B for final two-job listing). The one-job

Nucleus	Number of Basis States	Number of One-jobs	Number of Two-jobs	Measured MFG Time (seconds)	Estimated Final MCMII Time (seconds)
^{28}Si m=0	93,710	414,848	12,165,224	396	711
^{27}Al m=1/2	80,115	349,824	10,089,502	294	568
^{27}Al m=5/2	64,299	279,102	7,802,290	197	444
^{29}Si m=7/2	51,421	221,704	6,002,244	126	380
^{29}Si m=9/2	37,971	162,247	4,200,906	76	271
^{23}Ne m=1/2	6,457	22,166	469,974	5	24

All figures for H-mode operation.

Table 6.3 Timings For Sample Nuclei

processing time will be reduced to 30 μsecs per two-body matrix element and the zero-job time to 16 μsecs per two-body matrix elements. We now have an estimated time for the above nuclei of;

for ^{31}P ; 22.4 secs for zero-job processing,
21.9 secs for one-job processing,
44.6 secs for two-job processing,
giving a total of 88.9 seconds.

For ^{24}Mg ; 4.5 secs for zero-job processing,
7.9 secs for one-job processing,
30.7 secs for two-job processing,
giving a total of 43.1 seconds.

While for ^{23}Ne ; 2.2 secs for zero-job processing,
4.0 secs for one-job processing,
17.9 secs for two-job processing,
giving a total of 24.1 seconds.

Thus the final MCMII will be approximately 30 % faster than the current limited version, making it in total a factor of 9 faster then the original MCMI.

Table 6.3 shows the estimated time that one of the final MCMII modules would require to process all the TSWs for a selection of nuclei. Also shown is the measured time for the MFG to produce all the TSWs. It can be seen that for the largest sd shell nuclei 2 MCMs would out perform the current MFG, while for the smaller nuclei at most 5 MCMs would be necessary. The increase in efficiency of the MFG for smaller nuclei is due to the fact that the Hamiltonian is less sparse the smaller the nuclei. For example the Hamiltonian for the ^{28}Si nucleus shown has only 0.29% non-zero entries, while the ^{23}Ne nucleus has 2.38%.

Table 6.3 shows that for the largest nuclei the average rate of production of TSWs is 32,000 per second, while for the smaller nuclei 100,000 TSWs are produced on average per second. Clearly neither I-bus nor CMA-bus are overloaded with the data transfer rates this produces.

Nucleus	Number of basis states	IBM (minutes)	MFG (minutes)
$^{28}_{14}Si$ m=0	93,710	-	6.60
$^{27}_{13}Al$ m=5/2	64,299	1.56	3.28
$^{29}_{14}Si$ m=7/2	51,421	1.05	2.17
$^{25}_{12}Mg$ m=1/2	44,133	0.79	1.65
$^{29}_{14}Si$ m=9/2	37,971	0.68	1.27
$^{25}_{12}Mg$ m=9/2	20,007	0.28	0.42
$^{23}_{10}Ne$ m=1/2	6,457	0.08	0.08

Table 6.4 Comparative Timings For A Single Iteration

In fact I-bus could easily support an increase of over 100 fold in the rate of production of TSWs for the larger nuclei, while CMA-bus, which requires two transfers per task, could support an increase of 65 fold. However the current low usage of these two buses means that the MMPU is not yet near its saturation point and thus any increase in the number of MCMs should give a linear increase in its performance.

6.3 Conclusion

The figures given in table 6.3 show quite clearly that the MFG is currently the SMP system bottleneck. Table 6.4 shows MFG iteration times compared with equivalent timings on an IBM 360/195 system (figures obtained from [WWCM77]). Since the SMP system iteration time is the same as the MFG processing time when using only five MCMs then these figures show the final performance capabilities of the SMP system. As can be seen the performance is very respectable, being at worst only a factor of 2 slower than the IBM.

CHAPTER 7

The Extended SMP System

7.0 Introduction

The original aims of building a dedicated computer for nuclear structure calculations have been largely fulfilled in the SMP system. The system has been built at a low cost (less than £5000 for materials) and can carry out any sd shell calculation in a reasonable time. However as has been seen the limitation on the performance of the SMP system is the MFG. While this does not present a problem for the current system working on the sd shell it does severely limit the ability of the design to be extended to a system which will perform pf shell calculations. Such calculations, which would require an MFG which is 4 times the size, could generate basis lists of 10 to 100 times the size of sd shell lists. This would impose an impossible burden on an MFG of equivalent design to the current one. Thus new designs or new methods are required for the function of determining the Hamiltonian for an extended system.

7.1 Matrix Determination

During any shell model calculation it is only the Lanczos vectors which change between iterations, the Hamiltonian matrix being constant. There is therefore no reason, in theory, why the non-zero matrix elements should not be generated only once and then stored and read back during each iteration. Each matrix element would require only two entries; one being the 32-bit matrix value and the other being its 24-bit column

index. As with the current system the row index, which changes infrequently, can be broadcast to all MCMs only when it changes. Thus 7 bytes of information require to be stored per matrix element. The task of each MCM becomes much simpler with such a system since H_{mn} is read directly and does not have to be found by the MCMs from a look-up table. For sd shell calculations the maximum number of non-zero elements is approximately 12.5 million for ^{28}Si (m=0), requiring 87.5 Mbytes of storage. For the pf shell this figure could easily increase by a factor of 100, requiring almost 10 Gigabytes. Obviously this requires some form of disk storage to be used. Parallel disk assemblies of 1.5 Gbytes capacity and sustained transfer rates of 4 Mbytes/sec are available at under $12,000 [Mo87]. The use of such an assembly would be feasible for sd shell calculations, increasing performance by a factor of up to 15 for large nuclei. However multiple disk assemblies would be required even for medium sized pf shell calculations. These could be read in parallel increasing performance even further, but expense would become the limiting factor. For large pf shell calculations storage of the matrix would probably become impractical.

Matrix generation, as opposed to matrix storage, has much more potential for application to large pf shell calculations. It is quite feasible to construct a new MFG with a similar architecture to the current one but with a 6 to 10 fold increase in performance, i.e. a secondary state production rate of at least 50 MHz. This can be achieved with a much simplified SG channel design, a simplified timing control unit and a new pipelined OEC design, all implemented using 100K series ECL logic and using only a 50 MHz clock [Mac83]. A high performance PG/SG interface would also be used with a dedicated hardware controller responsible for transferring seed states to the SG. Such an interface could almost entirely eliminate SG overheads due to waiting for seed states.

The performance of the matrix generation function can be increased

even further by using an array of parallel MFGs. The whole MFG need not be duplicated but certainly the SG and PF functions would be and these could feed either their own private buffers or a single shared buffer. Each SG would then work on different columns of the matrix. New columns could be assigned sequentially on demand to each SG so that at any one time all SGs were working on columns in the same N-partition. In this case the seed state table would be common to all SGs. Alternatively the matrix could be block partitioned so that each SG has its own section of sequential columns to process. Each SG would have its own individual DMA controller which would transfer the seed states from the seed state table memory. If after the first iteration it were found that the processing load was spread too unevenly between the SGs then the MFG controller could redistribute the workload, and thus attempt to maximise the performance of the total MFG system.

However a problem is introduced with multiple SGs in that the MMPU must know which column any TSW it reads belongs to. One solution would be to tag each TSW to identify which SG produced it. Each MCM would then require a list giving the details of the column that each SG was processing. As before the MCMs must be informed when any SG finishes processing a column to allow them to take appropriate action, e.g. accumulate the previous V_{fn} (if working in H-mode) and read the new V_{in}.

Considerable amounts of hardware would be required even for one SG/PF in an extended system. In order to make multiple SG/PFs feasible custom gate or cell arrays would be necessary. However there is a different method for generating the Hamiltonian matrix elements which could remove the necessity for a separate MFG altogether. This approach would be to use an ***element-placement algorithm*** [MMBW88], which is a hybrid of the method used by the original Glasgow Program [WWCM77] combined with the SMP basis partitioning techniques.

The method used by the Glasgow Program first computes each element of the basis list, in numerical order, and stores the complete list in

primary memory. Each state in the basis list is then operated on directly to produce a secondary state, such that the two states form a non-zero matrix element. This is done by selecting pairs of set bits and clearing them, while a pair of cleared bits are then set, such that the appropriate quantum numbers are conserved. In this manner only the non-zero matrix elements are generated. However the index of the secondary element must then be determined and this is done by a binary search of the basis list, hence the reason why it must be stored in primary memory in the first place. The obvious limitation of this method is the necessity to store the complete basis list, requiring 16 bytes per element for pf shell calculations. For a multiprocessor architecture this list would have to be shared and would inevitably become a system bottleneck, since an exceptionally high bandwidth would be required in order that each processing element could perform its binary search. However by using an element-placement algorithm which would structure the basis list in a manner similar to the SMP system it is possible to remove the requirement to store the complete basis list. Instead the index of the secondary element can be calculated, due to the structure which has been imposed on the list, with the aid of functions and look-up tables. While this approach makes the task of generating each matrix element more complicated, it does remove the burden of generating large amounts of unwanted zero entries as with the current MFG. Matrix generation using element-placement algorithms would be best suited to being performed on the MCMs themselves, rather than on a dedicated hardwired module, so that the MFG function is effectively absorbed into the MMPU.

7.2 The Multiple Microprocessor Unit

Whatever method is used to increase the performance of the matrix generation function the MMPU will also have to have increased

capabilities, especially if element-placement algorithms are used. The MMPU can contain at most 16 MCMs and using MCMII this would not be enough to cope with an increased performance MFG or element-placement approach. Similarly CM and CMA-bus, which impose a limit of 2 million tasks per second will also require increased capabilities, since even before this limit is reached the system would begin to saturate so that increasing the number of MCMs would have a less than linear improvement upon system performance.

7.2.1 The Microcomputer Modules

The performance of any new MCM can be significantly improved upon by the use of new microprocessors. For example the recently introduced Motorola MC68030 is twice as powerful as the MC68020 and its floating point coprocessor the MC68882 is 4 times more powerful than the original MC68881. Also the introduction of the newer ***Reduced Instruction Set Computer (RISC)*** architectures [Wa85, Pa85] could bring significant improvements to the MCMs. The RISC philosophy, which advocates the simplification and optimisation of a computers instruction set and internal architecture, has recently been applied to a number of new microprocessors, e.g. the Intel 80960 series, the AMD 29000 and the Motorola M88000 family. The Motorola MC88100 RISC microprocessor has four fully concurrent, independent execution units (including a floating point unit) and two separate external buses for program and data (Harvard architecture) [Mot881]. The 20 MHz part boasts a sustained 14 to 17 MIPS (million instructions per second) and 7 MFLOPS (million floating point operations per second) processing rate, while being able to transfer data at a rate of up to 80 Mbytes/sec. In addition each MC88100 processor can support up to 4 MC88200 16-Kbyte cache/memory management units on each of its external buses. These provide full speed memory caching and demand-paged memory management as well as support for shared-memory multiprocessing [Mot882].

The MC88100 in conjunction with the MC88200 would seem an ideal processor on which to base an updated MCM, due to its optimised data movement and manipulation capabilities as well as its integral floating point unit. In order to condense as much processing power as possible into the MMPU each MCM could contain four MC88100s and 4 Mbytes of shared DRAM. Each processor would be equipped with 4 MC88200s and a small amount, perhaps 32K bytes, of fast static RAM, used to store initialisation software and supervisor mode functions. The DRAM would be shared on an equal priority basis between all processors and would be used to hold program code, look-up tables etc. With 4 MC88200s per processor, 2 for the data space and 2 for the program space, contention for the shared memory would be minimal for most applications, allowing each processor to run at full speed for most of the time.

However with such an architecture there arises the problem of how to handle the dedicated communications net interfaces. With a "live" bus such as C-bus where the processor itself controls all the bus accesses there is little problem. In this case each request by the processors to use such a bus can simply be arbitrated on a cycle by cycle basis by an onboard arbiter just as the shared memory would be arbitrated for. However the dedicated PFB interfaces of I-bus and CMA-bus require more control. For example with the CMA-bus interface a processor must be able to claim ownership of the PFB before using it. This enables a processor to start the CM access by writing to the PFB as normal and then keep ownership until the transaction is complete. This could be resolved by a number of methods;

1/ ***Software semaphores***; as with any shared resource where lockout must be provided semaphores could be used in the shared memory. These are accessed via indivisible read-modify-write accesses and are used to signal that the relevant interface is in use.

2/ ***Dedicated I/O processor***; a fifth processor could provide communications services for all the other processors. This processor

would have sole charge of and access to the MCM communications interfaces. A software image of the interfaces could be maintained for each processor in the shared memory. The I/O processor could poll each of these and when required provide the necessary servicing.

3/ ***Hardware image***; each processor could essentially have its own copy of each of the PFB registers. In the case of I-bus, when a processor emptied the contents of its image PFB register by reading it, the image PFB hardware would arbitrate for ownership of the real PFB. If the real PFB had current data then it would be transferred into the image, otherwise the image PFB simply waits until data arrives. The CMA-bus interface would be similar in that when the processor wrote a CM address to its image PFB, the hardware would arbitrate for ownership of the real CMA-bus PFB and once ownership was obtained it would be held until the CMA-bus transaction was complete. In this manner each processor would appear to have its own personal subnet interface PFB since all arbitration would be transparent.

All of these methods would provide an efficient means for sharing the MCM interfaces, with the last method providing the highest performance but at the expense of a considerable amount of additional hardware. Overall the proposed architecture would make each MCM a tightly coupled MIMD system and should give it approximately 20 to 30 times the performance of MCMII.

7.2.2 The Communications Subnet

In any new MMPU C-bus would be uprated to the full 32-bit data and address bus as allowed by the VME-bus specification, but would retain the enhancements which have been added in the SMP system (section 4.3). As has been said CM and CMA-bus must be improved beyond their current limit of 4 million accesses/second. This could be achieved by providing multiple CM modules, with the CM address space interleaved between the modules. Each CM module would also require the ability to queue incoming

requests which would be held and serviced in sequence. In addition the CMA-bus data and address buses can be split to allow independent transfers operating concurrently. Thus an MCM can send an address to a CM module in parallel with data being transferred to/from another MCM. Thus the transfer rate is no longer limited by the cycle time of the CM. CMA-bus transfer rates of over 30 MHz can be envisaged using fast bipolar TTL or even ECL interfaces and with 16 CM modules the CM system should be able to sustain this rate. This would provide a bandwidth of over 240 Mbytes/s, an increase of 8 fold.

7.3 Conclusion

The current SMP system demonstrates the processing power which can be readily available to scientists through the use of dedicated computing systems. By the application of parallel processing techniques and the use of modern VLSI devices such systems can be put together at a low cost and in a much reduced size compared to conventional computer installations.

Utilising the algorithms and architectural enhancements discussed it should be possible to build an extended shell-model processor with 100 times the power of the current system. Such a system, based on the basic architecture of the current SMP and using the experience gained in its design, would allow nuclear theorists to perform pf shell calculations which hitherto have not been feasible.

References

[AMD86]
AMD Bipolar Microprocessor Logic and Interface,
1986 Data Book.

[Ba76]
Jean-Loup Baer,
"Multiprocessing Systems",
IEEE Trans. Computers, Vol C-25, No 12, December 1976, pp 1271-1277

[Ba80]
Jean-Loup Baer,
"Computer Systems Architecture",
Computer Science Press, 1980.

[Bat80]
K.E. Batcher,
"Design of a Massively Parallel Processor",
IEEE Trans. Computers, Vol C-29, No 9, 1980, pp 836-840.

[Bat88]
R.T. Bate,
"The Quantum-Effect Device: Tomorrow's Transistor ?",
Scientific American, March 1988, pp 78-82.

[BCMW83]
I. Barron, P. Cavill, D. May, P. Wilson,
"Transputer does 5 or more MIPS even when not used in Parallel",
Electronics, November 17, 1983, pp 109-115.

[Ch86]
K. Chan,
"ECL Technology Suits High-Speed Logic Systems",
EDN, January 23, 1986, pp 153-158.

[ER74]
R. Eisberg, R. Resnick,
"Quantum Physics of Atoms, Molecules, Solids, Nuclei
and Particles",
John Wiley and Sons, 1974.

[FAST83]
Fastbus : A Modular High Speed Data Aquisition System for High
Energy Physics and Other Applications,
ESONE/FB/01, ESONE Committee, May 1983.

[Fi85]
W. Fischer,
"IEEE P1014 - A Standard for the High-Performance VME Bus",
IEEE Micro, February 1985, pp31-41.

[Fl66]
M.J. Flynn,
"Very High Speed Computing Systems",
Proc. IEEE, Vol 54, No 12, December 1966, pp 1901-1909.

[Fl72]
M.J. Flynn,
"Some Computer Organisations and Their Effectiveness",
IEEE Trans. Computers, Vol C-21, No 9, September 1972, pp 948-960

[FK83]
E.T. Fathi, M. Kreiger,
"Multiple Microprocessor Systems: What, Why and When",
IEEE Computer, March 1983, pp 23-32.

[FM77]
L. Fox, D.F. Mayers,
"Computing Methods for Scientists and Engineers",
Clarendon Press, Oxford, 1977.

[FO84]
G.C. Fox, S.W. Otto,
"Algorithms for Concurrent Processors",
Physics Today, May 1984, pp 50-59.

[Fu78]
S.H. Fuller, et al,
"Multi-microprocessors: An Overview and Working Example",
Proc. IEEE, Vol 66, No 2, February 1978, pp 216-228.

[GT83]
D.B. Gustavson, J. Theus,
"Wire-OR Logic on Transmission Lines",
IEEE Micro, June 1983, pp 51-55.

[Gu84]
D.B. Gustavson,
"Computer Buses - A Tutorial",
IEEE Micro, August 1984, pp 7-22.

[HB87]
K. Hwang, F.A. Briggs,
"Computer Architecture and Parallel Processing",
McGraw-Hill, 1987.

[Hi84]
W.D. Hillis,
"The Connection Machine: A Computer Architecture Based on Cellular Automata",
Physica, 10D, 1984, pp 213-228.

[HJ81]
R.W. Hockney, C.R. Jesshope,
"Parallel Computers",
Adam Hilger Ltd, 1981.

[HLSM82]
L.S. Haynes, R.L. Lau, D.P. Siewiorek, D.W. Mizell,
"A Survey of Highly Parallel Computing",
IEEE Computer, Jan. 1982, pp 9-24.

[Hw87]
K. Hwang,
"Advanced Parallel Processing with Supercomputer Architectures",
Proc. IEEE, Vol 75, No 10, October 1987, pp 1348-1379.

[IEEE81]
IEEE Computer Society,
"A Proposed Standard for Binary Floating-Point Arithmetic.
IEEE Draft 8.0 of Task P754",
IEEE Computer, March 1981, pp 51-62

[KT80]
W. Kozdrowicki, D.J. Theis,
"Second Generation of Vector Supercomputers"
IEEE Computer, November 1980, pp 71-

[Mac83]
L.M. MacKenzie,
"The Application of Microelectronics to Nuclear Physics Research",
Ph.D. Thesis, Dept of Physics, Glasgow University, 1983.

[MB76]
R.M. Metcalfe, D.R. Boggs,
"Ethernet: Distributed Packet Switching for Local Computer Networks",
Communications of the ACM, Vol 19, No 7, July 1976, pp 395-403.

[MBMW85]
L.M. MacKenzie, D.J. Berry, A.M. MacLeod, R.R. Whitehead,
"A Dedicated Lanczos Computer for Nuclear Structure Calculations",
The Recursion Method and its Applications,
eds D.G. Pettifor & D.L. Weaire, Springer-Verlag Berlin, 1985, p165

[MECL83]
"MECL System Design Handbook",
Motorola Semiconductor Products Inc, 4th Edition, 1983.

[MECL86]
"MECL Device Data Book",
Motorola Semiconductor Products Inc, 2nd Edition, 1986.

[MM83]
D. MacGregor, D. Mothersole,
"Virtual Memory and the MC68010",
IEEE Micro, June 1983, pp 24-39.

[MMB87]
L.M. MacKenzie, A.M. MacLeod, D.J. Berry,
"A Multiple Microprocessor System for CPU-bound Calculations",
The Computer Journal, Vol 30, No 2, 1987, pp 110-118.

[MMBW88]
L.M. MacKenzie, A.M. MacLeod, D.J. Berry, R.R. Whitehead,
"Concurrent Algorithms for Nuclear Shell Model Calculations",
Computer Physics Communications,
Vol 48, No 2, February 1988, pp 229-240.

[MMM84]
D. MacGregor, D. Mothersole, B. Moyer,
"The Motorola MC68020",
IEEE Micro, August 1984, pp 101-118

[Mo87]
N. Mokhoff.
"Parallel disk assembly packs 1.5 Gbytes, runs at 4 Mbytes/s",
Electronic Design, November 12, 1987, pp 45-46.

[Mot82]
Motorola MC68000 16-bit Microprocessor User's Manual
Prentice-Hall Inc, Third Edition, 1982.

[Mot010]
Motorola MC68010 Virtual Memory Processor,
Product Preview, Motorola Semiconductors, 1982.

[Mot230]
Motorola MC68230 Parallel Interface/Timer,
Advance Information, Motorola Semiconductors, 1981.

[Mot451]
Motorola MC68451 Memory Managment Unit,
Advance Information, Motorola Semiconductors, 1982.

[Mot452]
Motorola MC68452 Bus Arbitration Module,
Advance Information, Motorola Semiconductors, 1982

[Mot881]
Motorola MC88100, 32-bit Third-Generation RISC Microprocessor, Technical Summary, Motorola Semiconductors, 1988.

[Mot882]
Motorola MC88200, 16-Kilobyte Cache/Memory Management Unit (CMMU), Technical Summary, Motorola Semiconductors, 1988.

[Mot68000]
Motorola MC68000 16-bit Microprocessor Unit, Advance Information, Motorola Semiconductors, 1982.

[NS081]
"Interfacing the NS32081 as a Floating-Point Peripheral", Application Note, National Semiconductor Corporation, Microprocessor Applications Engineering.

[Pa72]
C.C. Paige, "Computational Variants of the Lanczos Method for the Eigenproblem", J. Inst. Maths. Applics, 10, 1972, pp 373-381

[Pa85]
D.A. Patterson, "Reduced Instruction Set Computers", Communications of the ACM, Vol 28, No 1, January 1985, pp 8-21.

[Pr79]
D. Prener, "Large Multimicroprocessor Systems", Microprocessors and Microsystems, Vol 3, No 6, July/August 1979, pp 271-276.

[PRT85]
R.B. Pearson, J.L. Richardson, D. Toussaint, "Special-Purpose Processors in Theoretical Physics", Communications of the ACM, Vol 28, No 4, April 1985, pp 385-389.

[RT88]
M. Reece, P. Treleavan, "Computing from the Brain", New Scientist, 26 May, 1988, pp 61-64.

[Ro69]
S. Rosen, "Electronic Computers: A Historical Survey", Computing Surveys, Vol 1, No 1, March 1969, pp 7-36.

[SG79]
E. Stritter, T. Gunter,
"A Microprocessor Architecture for a Changing World:
The Motorola 68000",
IEEE Computer, February 1979, pp 43-52.

[Se85]
C.L. Seitz,
"The Cosmic Cube",
Communications of the ACM, Vol 28, No 1, January 1985, pp 22-33.

[SM84]
C.L. Seitz, J. Matisoo,
"Engineering Limits on Computer Performance",
Physics Today, May 1984, pp 38-45.

[Ta84]
D.M. Taub,
"Arbitration and Control Aquisition in the Proposed
IEEE 896 Futurebus",
IEEE Micro, August 1984, pp 28-41.

[VME82]
VMEbus Specification Manual,
Revision B, August 1982,
VMEbus Manufacturers Group (Motorola, Mostek, Signetics/Philips).

[Wa85]
P. Wallich,
"Toward Simpler, Faster Computers",
IEEE Spectrum, August 1985, pp 38-45

[Wh72]
R.R.Whitehead,
"A Numerical Approach to Nuclear Shell-Model Calculations",
Nuclear Physics, A182, 1972, pp 290-300

[Wi87]
T. Williams,
"Optics and Neural Nets: Trying to Model the Human Brain",
Computer Design, March 1987, pp 47-62

[WWCM77]
R.R. Whitehead, A. Watt, B.J. Cole, I. Morrison,
"Computational Methods for Shell-Model Calculations",
Advances in Nuclear Physics, Vol 9, Chapter 2, Plenum Press 1977.

List of Abbreviations

			Section No.
BAM	--	Bus Arbitration Module	(4.3.4)
BBFC	--	Buffer Block Finished Comparator	(3.5.6)
BCSR	--	Buffer Control and Status Register	(3.6.1)
BRAC	--	Buffer Read Address Counter	(3.5.6)
BWAC	--	Buffer Write Address Counter	(3.5.6)
BWC	--	Buffer Word Counter	(3.5.6)
CCM	--	Channel Clocking Memory	(3.5.3)
CIT	--	Channel Information Table	(3.7.1)
CM	--	Central Memory	(2.5.2)
CU	--	Control Unit	(1.1)
DTB	--	Data Transfer Bus	(3.5.9)
DTC	--	Driver Table Constructor	(3.7.3)
DUART	--	Dual Universal Asynchronous Receiver/Transmitter	(4.6)
FNP	--	Final N-Partition	(3.7.1)
FPU	--	Floating Point Unit	(5.2)
GMC	--	Global Module Controller	(4.3.1)
GOT	--	Global Offset Table	(5.3.1)
INP	--	Initial N-Partition	(3.7.1)
JSW	--	Job Status Word	(3.7.1)
LBR	--	Local Bus Requestor	(5.1)
LMC	--	Local Module Controller	(5.2.4)
LOT	--	Local Offset Table	(5.3.1)
MCM	--	Micro-Computer Module	(2.5.2)
MFG	--	Matrix Format Generator	(2.5)
MFLOP	--	Millions of Floating Point Operations/sec	(1.1)
MID	--	Module ID	(4.3.2)
MIT	--	Module Identification Table	(4.6.2)
MMPU	--	Multiple Microprocessor Unit	(2.5)
MMU	--	Memory Management Unit	(4.6)
MPC	--	M-Partition Controller	(3.7.2)
NPC	--	N-Partition Controller	(3.7.2)
OEC	--	Operator Encoder Channel	(3.3)
PE	--	Processing Element	(1.1)
PF	--	Pair Filter	(2.5.1)
PFB	--	Prefetch Buffer	(3.7.2)
PG	--	Primary Generator	(2.5.1)
PIC	--	Primary Index Counter	(3.7.1)
PI/T	--	Parallel Interface/Timer	(3.6)
PSR	--	Prime State Register	(3.6.1)
RDB	--	Runtime Data Block	(3.7)

SC	--	Slave Controller	(5.2.5)
SD	--	Slater Determinant	(2.2)
SDWS	--	Slater Determinant Word Sequencer	(3.7.2)
SG	--	Secondary Generator	(2.5.1)
SGCSR	--	SG Control and Status Register	(3.6.1)
SGDR	--	SG Driver Routine	(3.7.3)
SIC	--	Secondary Index Counter	(3.2)
SM	--	Supervisor Module	(2.5.2)
SMP	--	Shell Model Processor	(2.0)
STB	--	Seed Table Builder	(3.7.3)
TCU	--	Timing and Control Unit	(3.5.1)
TSW	--	Task Setup Word	(2.5.1)

Appendix A

Two-job software listing for current MCMII.

```
*************************************************************************
*                                                                       *
*                MASTER PROCESSOR TWO-JOB ROUTINE                       *
*                                                                       *
*************************************************************************
*
* Address register contents;
*         A0 ---- Base address of Slaves workspace,
*         A1 ---- Base address of initial and final vectors,
*         A2 ---- Workspace,
*         A3 ---- Local directory,
*         A4 ---- Matrix Element Table,
*         A5 ---- Global Directory,
*         A6 ---- Mask table.
*
* NOTE that all instructions which reference tables held in DRAM require
* 2 extra  clock cycles per word length access due to wait states.  This
* affects all references via address registers A1-A6.
*
* Data register contents;
*         D5 ---- Prime state.
                                                              Number of
                                                             clock cycles
* Read words 0 and 1 from I-bus PFB.
* Contains job-type bits and SIC.
TESTJOB  MOVE.L  I_BUS,D4                                        20
         MOVE.L  D4,SEC_IN(A2)          Save in workspace.       16+4
* Test job-type bit and branch if not a
* two-job to determine what type it is.
         BCLR.L  #23,D4                                          14
         BNE     P.JOBTYPE              Branch if not a two-job  12
* Job is a two-job.
* Mask off unwanted bits in long word to leave only the SIC.
         ANDI.L  #SICMSK,D4                                      16
* Get the four operators from I-bus PFB.
         MOVE.L  I_BUS+4,D0                                      20
         MOVEP.L D0,OPERATORS+1(A2)     Separate operators       24+8
         MOVE.L  D5,D7                  Copy prime state into D7  4
* Place the four operators i,j,l,k
* in registers D0 to D3 respectively.
         MOVEM.W OPERATORS(A2),D0-D3                             32+8
* Get base address of O-table in A0.
         LEA     OTABLE(A2),A0                                   8
* Get the local offset using i and j.
         ADD.W   D1,D1                  2j                       4
         MOVE.W  (A0,D1.W),D1           Get j(j-1) from O-table  14+2
         ADD.W   D0,D0                  2i                       4
         ADD.W   D0,D1                  2i + j(j-1)              4
         MOVE.W  (A3,D1.W),D6           Get local directory entry 14+2
         ADD.W   D1,D1                  4i + 2j(j-1)             4
```

```
* Annihilate appropriate bits in prime state.
        BCHG.L  D3,D7                   Annihilate kth orbital      8
        BCHG.L  D2,D7                   Annihilate lth orbital      8
* Get global offset using k and l and
* add to local offset.
        ADD.W   D2,D2                   21                          4
        MOVE.W  (A0,D2.W),D2            Get l(l-1) from O-table     14+2
        ADD.W   D3,D3                   2k                          4
        ADD.W   D2,D3                   2k + l(l-1)                 4
        ADD.W   (A5,D3.W),D6            Get global offset and add   14+2
        ADD.W   D3,D3                   4k + 21(l-1)                4
* Get matrix element value from MET.
        MOVE.L  (A4,D6.W),D6                                        18+4
* Determine the sign of the matrix element.
        MOVE.L  (A6,D1.W),D1            Get i,j mask                18+4
        MOVE.L  (A6,D3.W),D3            Get k,l mask                18+4
        EOR.L   D3,D1                   Form composite mask         8
        AND.L   D1,D7                   Mask prime state word       8
        MOVE.L  D7,D1                   Copy resultant word into D1 4
        SWAP    D1                      Swap words in D1            4
        EOR.W   D1,D7                   Exclusive-OR the two words  4
        MOVE.W  D7,D1                   Copy resultant word into D1 4
        LSR.W   #8,D1                   shift byte 1 into byte 0    22
        EOR.B   D7,D1                   Exclusive-OR the two bytes  4
* Use resultant byte to address Parity table. If parity
* byte is not equal zero then matrix element is negative.
        LEA     PTABLE(A2),A0           Parity table address in A0  8
        MOVE.B  (A0,D1.W),D1            Read parity byte           14+2
        BEQ.S   POS                     Branch if parity byte zero  8
        BCHG.L  #31,D6                  Change matrix element sign  12
* Reinitialise slaves workspace address in A0.
POS     MOVE.L  ASLWSPC(A2),A0                                      16+4
        LSL.L   #3,D4                   Multiply SIC by 8.          14
*
* Got matrix element so send to slave when he is ready.
WAIT2   TST.W   (A0)                    Test if slave ready for job 8
        BPL.S   WAIT2                   If not then wait            8
        MOVE.L  (A1,D4.L),SLT.NVM(A0)   Send new Vim                30+4
        MOVE.L  4(A1,D4.L),SLT.NFM(A0)  Send new Vfm                30+4
        MOVE.L  D6,SL.NHMN(A0)          Send Hmn                    16
        MOVE.L  D4,SL.NSI(A0)           Send SIC                    16
        MOVE.W  #T2JOB,(A0)             Send two-job code           12
* Test to see if slave has any final vector
* elements to send back to be stored in vector table.
        TST.W   SLT.CODE(A0)            Test slave code word        12
        BMI.S   NEXTJOB                 If no data then pass        8
        MOVE.L  SLT.IN(A0),D0           Otherwise read vector index 16
        MOVE.L  SLT.FI(A0),4(A1,D0.L)   Read and store Vfm          30+4
        MOVE.W  #NOVAL,SLT.CODE(A0)     Signal that data read       16
* Test I-bus interface control to determine if
* a new TSW has arrived, if so then go back to start.
NEXTJOB TST.B   MCNTRL                                              16
        BPL     TESTJOB                 Back to start               10
        ...

                                        Total   =   731   clocks

                                                =   45.7  μsecs
```

```
*************************************************************************
*                                                                       *
*                SLAVE PROCESSOR TWO-JOB ROUTINE                        *
*                                                                       *
*************************************************************************
*
* Address register contents;
*       A0 ---- Pointer to SICODE and base address of workspace,
*       A1 ---- unused,
*       A2 ---- FPU - address to send ID word,
*       A3 ---- FPU - address to send operands and read results,
*       A4 ---- FPU - address to read status,
*       A5 ---- unused,
*       A6 ---- unused.
*
* Data register contents;
*       D7 ---- ID-byte for FPU.
*
* FPU operation times;
*       Multiply   =   6     µsec,
*                  =   96    processor clocks.
*       Addition   =   9.375 µsec,
*                  =   150   processor clocks.
*
                                                          Number of
                                                        clock cycles
* Test SICODE word to determine if got another
* task and what its type is.
TSTJOB  TST.W   (A0)                                            8
        BEQ.S   START2                                          10
        BMI.S   TSTJOB
        ...

* Start a two-job.
START2  MOVE.W  #T2JOB,SL.RCIC(A0)    Save job type in workspace    16
        MOVE.L  SL.NHMN(A0),D2        Read new Hmn                  16
        MOVE.L  SL.NSI(A0),D5         Read new index                16
        MOVE.L  SLT.NVM(A0),D3        Read new Vim                  16
        MOVE.L  SLT.NFM(A0),D1        Read new Vfm                  16
        MOVE.W  #NOVAL,(A0)           Signal that data read         12
* Save Hmn and index in workspace.
        MOVE.L  D2,SL.HMN(A0)         Save Hmn                      16
        MOVE.L  D5,SL.SI(A0)          Save index                    16
        SWAP    D2                    Swap Hmn, ready for FPU       4
*
* Multiply Hmn by Vin (Vin is held in FPU).
        MOVE.W  D7,(A2)               Send ID word to FPU           8
        MOVE.W  #MULFF0A,(A3)         Send operation code word      12
        MOVE.L  D2,(A3)               Send Hmn                      12
*
        SWAP    D1                    Swap Vfm, ready for FPU       4
        MOVE.W  #ADDFIA,D6            Get next op. word ready       8
*                                                       Wait       84
* Read back FPU status word.
        MOVE.W  (A4),D0               Read back FPU status          8
        BEQ.S   FP2                   Branch if OK                  10
        TRAP    #1                    Otherwise TRAP if error
        DC.W    FPUER                 FPU error signal to TRAP routine
```

```
* Read result from FPU.
FP2     MOVE.L  (A3),D0             Read multiplication result   12
* Add result of multiplication to Vfm.
        MOVE.W  D7,(A2)             Send ID word                  8
        MOVE.W  D6,(A3)             Send FPU op. word             8
        MOVE.L  D0,(A3)             Send previous result         12
        MOVE.L  D1,(A3)             Send Vfm                     12
* These instructions pipelined with FPU operation
        SWAP    D3                  Swap Vim, ready for FPU       4
        MOVE.W  #MULFIA,D6          Get next FPU op. code word    8
*                                                      Wait     138
* Read back FPU status.
        MOVE.W  (A4),D0             Read FPU status               8
        BEQ.S   FP3                 Branch if OK                 10
        TRAP    #1
        DC.W    FPUER
* Read result from FPU.
FP3     MOVE.L  (A3),D1             Read result                  12
*
* Multiply Hmn by Vim.
        MOVE.W  D7,(A2)             Send ID word                  8
        MOVE.W  D6,(A3)             Send FPU op. word             8
        MOVE.L  D2,(A3)             Send Hmn                     12
        MOVE.L  D3,(A3)             Send Vim                     12
* This instruction pipelined with FPU operation
        MOVE.W  #ADDFIF1,D6         Get next FPU op. code word    8
*                                                      Wait      88
* Read back FPU status.
        MOVE.W  (A4),D0             Read FPU status               8
        BEQ.S   FP4                 Branch if OK                 10
        TRAP    #1
        DC.W    FPUER
* Read result from FPU.
FP4     MOVE.L  (A3),D0             Read result                  12
*
* Add result to Vfn (Vfn held in FPU register).
        MOVE.W  D7,(A2)             Send ID word                  8
        MOVE.W  D6,(A3)             Send FPU op. word             8
        MOVE.L  D0,(A3)             Send previous result         12
* Send back updated Vfm to master.
* These instructions pipelined with FPU operation
WAIT2   TST.W   SLT.CODE(A0)        Test if master ready         12
        BPL.S   WAIT2               If not then wait              8
        SWAP    D1                  Swap Vfm back to normal       4
        MOVE.L  D1,SLT.FI(A0)       Send back Vfm                16
        MOVE.L  D5,SLT.IN(A0)       Send back index              16
        MOVE.W  #T2JOB,SLT.CODE(A0) Signal that data sent        16
*                                                      Wait      78
* Read back status for FPU operation, no result to be read back.
        MOVE.W  (A4),D0             Read FPU status               8
        BEQ     TSTJOB              Branch if OK to start        10
        TRAP    #1
        DC.W    FPUER
        ...

                                    Total   =   886  clocks

                                            =   55.4 µsec
```

Appendix B

Two-job software for final version MCMII.

```
*************************************************************************
*                                                                       *
*                MASTER PROCESSOR TWO-JOB ROUTINE                       *
*                                                                       *
*************************************************************************
*
* Address register contents;
*         A0 ---- Base address of Slaves workspace,
*         A1 ---- O-table , Parity table +64,
*         A2 ---- Workspace,
*         A3 ---- Local directory,
*         A4 ---- Matrix Element Table,
*         A5 ---- Global Directory,
*         A6 ---- Mask table.
*
* NOTE that all instructions which reference tables held in DRAM require
* 2 extra  clock cycles due to wait states. This affects all  references
* via address registers A1 and A3-A6.
*
* Data register contents;
*         D5 ---- Prime state.
                                                          Number of
                                                        clock cycles
* Read SIC and job-type bits from I-bus PFB
* and store.
TESTJOB  MOVE.L  I_BUS,D4                                      20
         MOVEL.L D4,SEC_IN(A2)                                 16
* Test job-type bits
* and branch if not a two-job
         BCLR.L  #23,D4                                        14
         BNE     P.JOBTYPE                                     12
* Is a two-job, mask off unwanted bits to leave only SIC, read and store
* operators, make copy of prime state in D7.
         ANDI.L  #SICMCK,D4                                    16
         MOVE.L  I_BUS+4,D0                                    20
         MOVEP.L D0,OPERATORS+1(A2)                            24
         MOVE.L  D5,D7                                          4
* Get operators into data registers.
* i into D0, j into D1, l into D2, k into D3.
         MOVEM.W OPERATORS(A2),D0-D3                           32
* Get local offset
         ADD.W   D1,D1                 2j.                      4
         ADD.W   D0,D0                 2i.                      4
         ADD.W   (A1,D1.W),D0          2i + j(j-1).          14+2
         MOVE.W  (A3,D0.W),D6          Get local offset.     14+2
* Annihilate particles in prime state.
         BCHG.L  D3,D7                                          8
         BCHG.L  D2,D7                                          8
```

```
* Get global offset.
        ADD.W   D2,D2                   21.                             4
        ADD.W   D3,D3                   2k.                             4
        ADD.W   (A1,D2.W),D3            2k + l(l-1).                    14+2
        ADD.W   (A5,D3.W),D6            Add on global offset.           14+2
* Mask state.
        AND.L   (A6,D6.W),D7                                            20+4
* Get two-body matrix element.
        MOVE.L  (A4,D6.W),D6                                            18+4
* Determine sign change.
        MOVE.L  D7,D1                   Copy result.                    4
        SWAP    D1                                                      4
        EOR.W   D1,D7                   EOR two halves.                 4
        MOVE.W  D7,D1                                                   4
        LSR.W   #8,D1                                                   22
        EOR.B   D7,D1                   EOR two bytes.                  4
* Get parity byte and if not zero then change sign.
        MOVE.W  64(A1,D1.W),D1          Get parity byte.                14+2
        BEQ.S   WAIT2                   Branch if zero.                 8
        BCHG.L  #31,D6                  Otherwise change sign.          12
* Pass parameters to slave if ready to take more.
WAIT2   TST.W   (A0)                    Test if slave ready             8
        BPL.S   WAIT2                   If not then wait.               8
        MOVE.L  D6,SL.NHMN(A0)          Pass matrix element.            16
        MOVE.L  D4,SL.NSI(A0)           Pass secondary index.           16
        MOVE.W  #T2JOB,(A0)             Pass two-job code.              12
* Test if I-bus ready with another TSW,
* if so then do again.
NEXTJOB TST.B   MCNTRL                  Test if I-bus ready.            16
        BPL     TESTJOB                 If so then do again.            10
        ...

                                        Total   =  464  clocks

                                                =  29   µsecs
```

```
*************************************************************************
*                                                                       *
*                SLAVE PROCESSOR TWO-JOB ROUTINE                        *
*                                                                       *
*************************************************************************
*
* Address register contents;
*        A0 ---- Pointer to SICODE and base address of workspace
*        A1 ---- CMA-bus PFB
*        A2 ---- FPU1 - address to send ID byte,
*        A3 ---- FPU2 - address to send ID byte,
*        A4 ---- FPU1 - address to send operands and read result,
*        A5 ---- FPU2 - address to send operands and read result,
*        A6 ---- SCNTRL control register for CMA-bus.
*
* Data register contents;
*        D7 ---- ID-byte for FPUs.
                                                          Number of
                                                        clock cycles
* Read data from parameter passing area.
PS.STRT2 MOVE.L  SL.NHMN(A0),D2        Read new Hmn.            16
         MOVE.L  SL.NSI(A0),D5         Read new secondary index. 16
* Signal to master processor (via SICODE) that data has been read.
         MOVE.W  #NOVAL,(A0)                                    12
* Activate CMA-bus to read Vim and Vfm.
         MOVE.L  D5,(A1)               Activate CMA-bus.        12
* Save data in workspace.
         MOVE.L  D5,SL.SI(A0)                                   16
         MOVE.L  D2,SL.HMN(A0)                                  16
* Start off FPU1 ---- Hmn x Vin
         MOVE.W  D7,(A2)               Send FPU ID-byte.        8
         MOVE.W  #MULFF0A,(A4)         Send operation code word. 12
         SWAP    D2                    Swap Hmn.                4
         MOVE.L  D2,(A4)               Send Hmn.                12
* Read vector elements from CMA-bus PFB, if ready.
WAITC    TST.B   (A6)                  Test if ready.           8
         BPL.S   WAITC                 If not then wait.        8
         MOVE.L  4(A1),D3              Read Vim.                16
* Start of FPU2 ---- Hmn x Vim
         MOVE.W  D7,(A3)               Send ID-byte.            8
         MOVE.W  #MULFIA,(A3)          Send operation code word. 12
         MOVE.L  D2,(A5)               Send Hmn to FPU2.        12
         SWAP    D3                    Swap Vim                 4
         MOVE.L  D3,(A5)               and send to FPU2.        12
* Read Vfm from CMA-bus PFB.
         MOVE.L  8(A1),D1              Read Vfm from CMA-bus.   16
         SWAP    D1                    Swap Vfm.                4
* Read back status from FPU1 ( = Hmn x Vin)
         MOVE.W  8(A2),D0              Read status word.        12
         BEQ.S   FP1                   If zero then OK.         10
         TRAP    #1
         DC.W    FPU1ER
* Read result from FPU1.
FP1      MOVE.L  (A2),D0               Read result.             12
```

```
* Start off FPU1 (Hmn x Vin) + Vfm
        MOVE.W  D7,(A2)             Send FPU ID-byte.          8
        MOVE.W  #ADDFIA,(A4)        Send operation code word.  12
        MOVE.L  D0,(A4)             Send previous result.      12
        MOVE.L  D1,(A4)             Send Vfm.                  12
* Read back status from FPU2 (= Hmn x Vim)
        MOVE.W  8(A3),D0            Read status word.          12
        BEQ.S   FP2                 If zero then OK.           10
        TRAP    #1
        DC.W    FPU2ER
* Read result from FPU2.
FP2     MOVE.L  (A5),D0             Read result.               12
* Start off FPU2 (Hmn x Vim) + Vfn.
        MOVE.W  D7,(A3)             Send ID-byte.              8
        MOVE.W  #ADDFIF1,(A5)       Send operation code word.  12
        MOVE.L  D0,(A5)             Send previous result.      12
*
* Wait for FPU1;                                 wait approx.  102
*
* Read back status from FPU1.
        MOVE.W  8(A2),D0            Read status word.          12
        BEQ.S   FP3                 If zero then OK.           10
        TRAP    #1
        DC.W    FPU1ER
* Read result from FPU1.
FP3     MOVE.L  (A4),D0             Read result (=Vfm)         12
        SWAP    D0                  Swap                       4
        MOVE.L  D0,8(A1)            Send to CMA-bus PFB.       16
        MOVE.L  D5,(A1)             Activate CMA-bus.          12
*
* Wait for FPU2;                                 wait approx.  8
*
* Read back status from FPU2 (no result to read back).
        MOVE.W  8(A3),D0            Read status word.          12
        BEQ.S   FP4                 If zero then OK.           10
        TRAP    #1
        DC.W    FPU2ER
* Test SICODE from master processor for next job.
FP4     CMPI.W  #T2JOB,(A0)         Test for next job.         12
        BEQ     PS.STRT2            If two-job then do again.  10
        ...

                                    Total   =   588    clocks

                                            =   36.75  µsecs
```

Computer Physics Communications 48 (1988) 229–240
North-Holland, Amsterdam

CONCURRENT ALGORITHMS FOR NUCLEAR SHELL MODEL CALCULATIONS

L.M. MACKENZIE

Dept. of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland

A.M. MACLEOD, D.J. BERRY and R.R. WHITEHEAD

Dept. of Physics and Astronomy, University of Glasgow, Glasgow, Scotland

Received 30 July 1987

The calculation of nuclear properties has proved very successful for light nuclei, but is limited by the power of the present generation of computers. Starting with an analysis of current techniques, this paper discusses how these can be modified to map parallelism inherent in the mathematics onto appropriate parallel machines. A prototype dedicated multiprocessor for nuclear structure calculations, designed and constructed by the authors, is described and evaluated. The approach adopted is discussed in the context of a number of generically similar algorithms.

1. Introduction

Physicists have been investigating the structure of the atomic nucleus since its discovery in the early years of this century. The difficulties are considerable: not only is the nucleus a quantum many-body system, but it is governed, moreover, by an interparticle potential (the nucleon–nucleon interaction) which is not fully understood. From empirical observations, it has long been known that there are indications of a shell structure, resembling the well-understood model of the atom. Yet, despite basic similarities, there are major fundamental differences of character between the atomic and nuclear cases: the existence of two distinct types of nucleon; the more exotic nature of the nuclear force; and the absence of any heavy centre of this force.

The basic assumption of the Shell Model [1] is, that to a first approximation, each nucleon moves independently in a potential that represents the average interaction with the other nucleons. The solutions of the single-particle Schrödinger equation in an approximation to this potential reveal a fundamental shell structure. Upon inclusion of the spin–orbit term, it is then possible to make predictions in encouraging agreement with experiment, in cases where two-body forces are not effective.

In most nuclei, however, we must assume that the Hamiltonian has a two-body nature:

$$H = \frac{-\hbar^2\nabla^2}{2m} + \sum_{1=i<j} V(i, j).$$

The crucial problem of diagonalising this operator may be tackled by choosing as a basis for the configuration space, wave functions with definite values of "good" quantum numbers like J (total angular momentum), T (isospin) etc., thereby permitting a decomposition into subspaces where these quantum numbers are conserved. Although this reduces the magnitude of the problem somewhat, the technique is not without its drawbacks: for example in the form of the elaborate algebra of coupling central to its mathematical development.

Within the last decade or so, theorists at Glasgow have explored a fruitful alternative, the *m-scheme*, in which a Slater determinant basis is employed (Slater determinants are eigenvalues of the single-particle state occupation operators). Al-

0010-4655/88/$03.50

though very large configuration spaces result from this strategy, the Hamiltonians can be efficiently dealt with by proper attention to numerical techniques. The real strength of the method lies in the natural way in which it can be mapped onto an extremely simple and elegant digital representation, making it ideal for computer manipulation. This mapping is central to the considerable success of the Glasgow Shell Model Program [2].

However, even light *m*-scheme systems can generate a substantial computational load (enough to tax available time on conventional mainframes), and the machine resources required by nuclei of higher mass number are enormous. The authors believe that only a parallel solution to this problem is realistic, but, to pursue such a course, certain difficulties must first be overcome. The current Glasgow Program is, in essence, a uniprocessor algorithm, which, for various reasons is not directly suitable for a concurrent machine. Further, the scale of the potential CPU demand is so large that even were a suitable algorithm identified and successfully mapped onto a general purpose parallel computer architecture, any existing or planned machine would still be unable to meet it satisfactorily. It seems logical therefore, that any search for such an algorithm should be undertaken with a view to its possible implementation as a dedicated system. This paper describes a recent project at Glasgow to identify a class of concurrent shell model algorithms, and investigate the feasibility of mapping this class onto real machine architectures. As part of this project, a pilot *Shell Model Processor*, has been constructed. This machine illustrates the principles applicable to a much larger system, although its own capabilities are necessarily limited.

2. Review of shell model theory

The computer-oriented representation of the Glasgow Shell Model Program is developed from the traditional *occupation number* formalism. In any *n*-particle system we can, given the single-particle states ordered by some arbitrary means, form a basis for the system as a whole from the Slater determinants:

$$|n_1 \cdots n_i \cdots\rangle$$

with n_i particles in state i. These are eigenfunctions of the number operators, representing *n*-particle states with definite values for the occupancy of each single-particle state [3]. For a nuclear system, of course, the Pauli principle demands that $n_i = 0$ or 1, for each i. The Glasgow formalism involves assigning each single-particle nucleon state of a given system to a different bit of a computer word. The values, 0 or 1, which the bit can assume, indicate whether the state is empty or full. Thus the Slater determinant $|10010100\rangle$, describing a 3-particle system with 8 possible single-particle states, can be represented by an 8-bit word 10010100.

Much of the initial effort expended on the *m*-scheme approach, has concentrated on light nuclei, with an active sd-shell (fig. 1). The sd-shell consists, in fact, of 3 sub-shells ($1d_{5/2}$, $2s_{1/2}$ and $1d_{3/2}$) with a total of 12 single-particle states for protons and an equivalent 12 states for neutrons. For calculations involving sd-shell nuclei, the closed 1s and 1p shells are considered only as contributing to the overall single-particle potential, while the outer pf-shell is deemed to be inaccessible (although this constraint is sometimes slightly relaxed). Hence only the 24 sd-shell

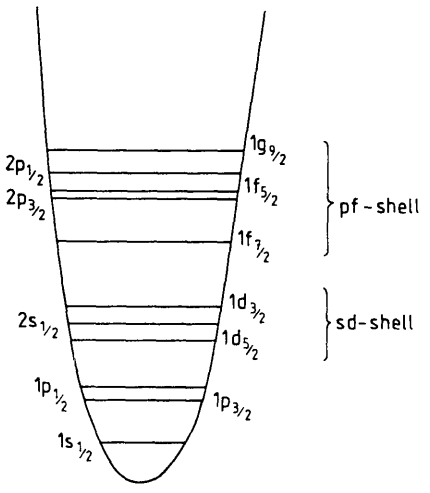


Fig. 1. Energy levels in the shell model.

orbitals need be considered and the computer word used for representation requires 24 bits.

In the m-scheme there is a trade-off between the flexibility of the representation and the size of the configuration space. Despite the indeterminate values of most of the helpful quantum numbers, however, Slater determinants are eigenfunctions of the M_J operator (z-component of total angular momentum), so conservation does still allow some reduction in the scale of the problem. The first step is to generate a complete set of Slater determinants with prescribed quantum numbers (e.g. number of protons, number of neutrons, total M_J and parity), forming a basis for the section of the nuclear space under consideration. Once this is achieved, the fundamental problem, both from the physical and computational points of view, is that of evaluating the Hamiltonian matrix and thereupon diagonalising it. Many other calculations, such as for example, deriving the density matrix for a given state, or determining the expectation values of other observables, are essentially less demanding variations of this basic task.

The magnitude of the eigenvalue determination problem is a consequence of the iterative nature of the diagonalisation process, performed on what is likely to be a very large matrix. The largest pure sd-shell calculation generates a space with a dimension of about 10^5, but, if pf-shell orbitals are introduced, there is no realistic upper bound. Obviously, therefore, no machine can perform totally unconstrained Shell model calculations involving shells above the sd. The question is: how large a subset might become feasible in the foreseeable future?

If the Hamiltonian is treated as a two-body operator, a typical element $\langle f | H | i \rangle$ is zero whenever the basis states $|i\rangle$ and $|f\rangle$ differ by the position of more than two particles (i.e. if the representing digital words have a Hamming distance of more than four). Otherwise matrix entries can be computed by taking linear combinations of the empirically determined *uncoupled two-body matrix elements*. H_{ijkl}, of which there are a reasonably small number, and which contain the quantitative description of the nuclear force. The *zero-condition*, as the Hamming criterion will be referred to henceforth, does, however, imply a Hamiltonian matrix which is irregularly sparse, a feature which can be exploited to some extent, but which does not admit any real labour-saving mathematical devices. The Hamiltonian is, therefore, generated by determining which pairs of basis elements are linked by a non-zero matrix entry, and, for each such pair, by computing a real value determined by a simple evaluation theorem. For example, if exactly two bits differ between the representations of $|i\rangle$ and $|f\rangle$, the value is a single two-body element H_{ijkl}, apart from sign. The diagonalisation itself is accomplished by means of the *Lanczos* method [4]. This has the great advantage that, for any given $n \times n$ matrix A, it is only necessary to diagonalise its upper $m \times m$ corner for $m \ll n$, to obtain convergence for the dominant eigenvalues of A. Since this can be done after only $m - 1$ iterations, the Lanczos method is ideally suited to the problem of finding the lowest energy eigenstates of large Hamiltonian matrices.

The Lanczos algorithm itself, starting within a trial vector v_1, generates a sequence of mutually orthogonal vectors (y_i) such that:

$$AY = YT \quad \text{where} \quad Y = [y_1, \ldots, y_n],$$

and T is tridiagonal. During this process, a moderate amount of vector manipulation is involved, but by far the most computationally intensive step is the multiplication of A into the current iteration vector y_i. For matrices the size of the larger shell-model Hamiltonians this is a very heavy arithmetic load indeed.

The multiplication problem can be tackled in two distinct ways: the matrix can be generated once, then stored and retrieved when required; or it can be generated afresh, in real time, for every iteration. The former approach reduces the computational load, but requires enormous amounts of secondary storage (hundreds of gigabytes for relatively modest pf-shell calculations). Further, any attempt to exploit high-performance matrix–vector multipliers must confront the significant data retrieval problems which such a secondary storage scheme would face. Although the authors do not, by any means, consider that these difficulties are insurmountable, the present work is confined to development of the second alternative,

which seems to offer a more flexible, and at least as cost-effective, solution.

3. Parallelism in Shell Model calculations

Seeking a suitable parallel algorithm for Shell Model calculations, a natural first step is the explicit identification of any fundamental concurrency in the logic of the *m*-scheme, typified by the following sequence:

1) Generate an ordered Slater determinant basis for the space involved. This basis $(e_1, \ldots, e_N)$ serves as an index for the rows and columns of the Hamiltonian, and for the rows of the state vectors.
2) Find all pairs of basis elements (e_i, e_j) which fail the zero-condition test. If a pair passes the test, the corresponding (i, j)th entry of the Hamiltonian is, as discussed above, automatically zero.
3) For each contributing pair found, use the evaluation rules and the uncoupled two-body matrix elements to compute the real value of the (i, j)th Hamiltonian entry, say, H_{ij}.
4) For each nonzero H_{ij}, multiply by the jth element of the initial vector for the iteration and accumulate the product into the ith element of the product vector. When this has been achieved for all nonzero H_{ij}, the matrix multiplication is complete.

The most obvious parallelism arises at steps 3 and 4. The evaluation of individual matrix entries is independent unless tables of two-body elements are shared, and there is no great inhomogeneity in the amount of work associated with different entries. Furthermore, matrix multiplication is itself also inherently parallel, for clearly two arithmetic processors can proceed independently to compute contributions to a final vector given two distinct entries of the multiplier matrix. However, considering the potentially very large dimensions of the configuration spaces, the initial and final vectors for an iteration will inevitably be shared random access data structures, creating a possible limit to the maximum practical degree of concurrency.

The parallelism in tracking down the contributing pairs is less easy to exploit efficiently. Since the matrix is irregular, there is no way of, say, block-partitioning it, to share work equally amongst several processing elements. However, given that any subdivision may be unfair, it is possible to introduce concurrency here also by, for example, allocating different rows to different searching elements.

The ultimate aim is to provide an algorithm which allows maximal parallelism. To achieve this, any potential bottlenecks must be identified and their effect minimised. As is well known, such bottlenecks arise when processes are forced to communicate in such a way that the communication medium, or *subnet*, becomes saturated. The above analysis suggests that the presence of shared data structures is liable to engender just such a situation, and, clearly, minimising access to any such structures will be vital to the success of a practical system.

4. The Glasgow Program

As a salient starting point, the standard Glasgow Program algorithm [2] is analysed as applied to the sd-shell, providing a simple yet concrete example. Fig. 2 shows a typical assignment of single-particle states to a 24-bit word, divided in two, with the most significant half chosen (arbitrarily) to represent proton orbitals. Each bit, i, has associated with it, a value, m_i, representing the contribution to the z-component of total angular momentum from the ith orbital (if occupied). Of course:

$$M_J = \sum_i m_i .$$

Suppose a calculation involves a nucleus with n_p protons, n_n neutrons and total $M_J = M$. Generation of the basis with the orbital assignment of fig. 2, amounts to producing all 24-bit words with n_p ones in the upper 12 bits, n_n ones in the lower 12 bits, and with a total M_J contribution of M from these set bits. This is achieved by, first, filling the leftmost n_p proton bits and the leftmost n_n neutron bits with ones, then successively moving the rightmost set bit one place to the right, checking, as each new word is produced, whether its M_J value is M. Words satisfying the

Bit Number	l	j	m_j	Nucleon
23	d	5/2	+5/2	proton
22	d	5/2	+3/2	proton
21	d	3/2	+3/2	proton
20	d	5/2	+1/2	proton
19	d	3/2	+1/2	proton
18	s	1/2	+1/2	proton
17	d	5/2	-1/2	proton
16	d	3/2	-1/2	proton
15	s	1/2	-1/2	proton
14	d	5/2	-3/2	proton
13	d	3/2	-3/2	proton
12	d	5/2	-5/2	proton
11	d	5/2	+5/2	neutron
10	d	5/2	+3/2	neutron
9	d	3/2	+3/2	neutron
8	d	5/2	+1/2	neutron
7	d	3/2	+1/2	neutron
6	s	1/2	+1/2	neutron
5	d	5/2	-1/2	neutron
4	d	3/2	-1/2	neutron
3	s	1/2	-1/2	neutron
2	d	5/2	-3/2	neutron
1	d	3/2	-3/2	neutron
0	d	5/2	-5/2	neutron

Fig. 2. Typical assignment of single particle states to a 24-bit word.

conditions are stored, producing a basis list in descending numerical order.

Once the basis has been computed, each element, e_r, is used to begin a search along a row of the Hamiltonian, seeking non-zero entries. This is done by selecting a pair of set bits, say k and l, and resetting them. The indices k and l are used to locate a block of main store containing all two body matrix elements H_{ijkl}, for all i, j such that: $m_i + m_j = m_k + m_l$.

Proceeding through this list, those elements are selected which are such that when i and j are set, the new Slater determinant has valid n_p, and n_n (M is guaranteed by the above condition), so that it is a member of the basis, say e_s. The pair e_r, e_s now represent row and column indices for a Hamiltonian entry, $H_{r,s}$, which can be passed for evaluation: in fact its value, apart from sign will frequently be just H_{ijkl}. However, to use this element in the matrix multiplication, the value s must be explicitly known. This may be found by conducting a binary search on the stored basis list (which, recall, is in numerical order). Once s is found, $H_{r,s}$ can be multiplied by the sth element of the initial vector for the iteration, and accumulated into the rth element of the final vector. The process is repeated for all k and l in e_r, and then for each r, until the basis is exhausted.

To discover the limitations of this scheme, it is necessary to examine problems which may arise as it is applied to larger calculations. Firstly, note that there is inevitably a loss of efficiency in primitive manipulation, as the size of the Slater determinant representation exceeds that of a CPU word on the host machine. However, a rather more serious problem, is the requirement that the entire generated basis must reside in primary store throughout the calculation. For an sd-shell nucleus, assuming 32-bits per Slater determinant, this demand will not exceed 1/2 Mbyte, which is acceptable, given that the two iteration vectors will occupy twice this space. On the other hand a pf-shell representation word can be up to 128-bits long, so that a space of dimension, say 10^6, would require 16 Mbytes to store its basis alone.

If the algorithm were implemented on a parallel machine, presumably the initial and final iteration vectors, together with the stored basis would be shared data structures. In a typical pf-shell calcu-

lation, perhaps 20 access would be required by each element evaluation: a heavy load on the available shared bandwidth (a single access to the initial vector is inevitable anyway). Since this shared bandwidth is always a possible bottleneck, the potential degree of parallelism is consequently reduced substantially, when compared with an algorithm which could avoid this search.

Although other search algorithms could be employed, there is no ideal alternative candidate. For example, there is no obvious hash function which would efficiently map the basis into a hash table, without both requiring substantially more primary memory and significantly increasing the computation load of the search. Although a hashing approach would reduce the number of shared memory references per evaluation, the cost would be appreciable.

The ideal solution is one which allows the value of the column index to be determined without any shared memory references at all. It transpires that just such a solution is possible, eliminating not only the search process, but, in addition, requiring no basis storage whatsoever. The authors have investigated algorithms with this property, which depend on imposing a structure on the Slater determinant basis, and have designed and constructed a prototype parallel dedicated system, The Shell Model Processor (SMP), to test the method out in practice.

As will emerge shortly, the efficiency with which critical sections of an algorithm can be hardwired into high-performance hardware "accelerators" is an important consideration in the design of cost-effective special purpose computing machines. The Glasgow Program was not designed with these considerations in mind, and consequently, as might be expected, the techniques it employs (e.g. for basis generation) are not very suitable for such translation.

5. The Shell Model Processor project

The Shell Model Processor was developed to demonstrate the feasibility of implementing *structured-basis algorithms* as dedicated parallel architectures. The scope of the project has been limited to the development of a machine capable of performing sd-shell calculations, but with an architecture extensible to the pf-shell.

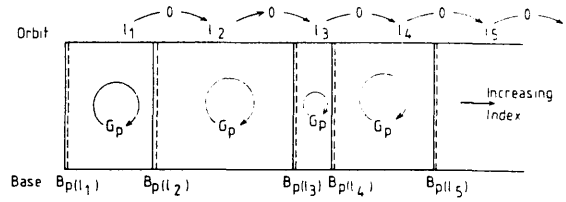


Fig. 3. Structured Basis organisation.

These algorithms generate an ordered basis for a given nuclear configuration space as a sequence of sublists. The process depends on defining an equivalence relation, $\sim$, on a Slater determinant basis, L, for the configuration space. As is well known, such a relation induces a 'partition' of L, the subsets of which can be ordered by some arbitrarily chosen order relation, $<_R$, to form a sequence, $(l_1, l_2, \ldots, l_n)$. A sequence of this kind is referred to henceforth as an *O-partition* of L (fig. 3). To facilitate discussion the following formal definitions are introduced.

1) Let $P(L) = (l_1, l_2, \ldots, l_n)$ be an O-partition of L. A permutation operator on $\{l_1, l_2, \ldots, l_n\}$ is called the *O-function* of P(L) if, for $i = 1, \ldots, n$,

$$O(l_i) = l_{i+1}, \quad \text{where} \quad l_{n+1} = l_1.$$

2) Let $P(L) = (l_1, l_2, \ldots, l_n)$ be an O-partition of L. A is called an *orbit-generating function*, Gp, for P(L) if it is a permutation operator on L such that the l_i are precisely the orbits of A [5] i.e. if e_i is an element of l_i, then:

$$l_i = \left(e_i, Ae_i, A^2e_i, \ldots, A^{(n-1)}e_i\right) \quad \text{and}$$

$$A^n e_i = e_i.$$

If one element (a *base*) is selected from each of the l_i, then Gp (together with P(L)) defines a total order on L converting it to an ordered basis $L_{P,G}$. The function mapping l_i onto its base element is called the *base function* of P(L), $B_p(L) \rightarrow L$.

3) The function mapping l_i onto the index of its base element, in the ordering of $L_{P,G}$, is called the *index function* of P(L), denoted Ip: $P(L) \rightarrow N$.

4) Let e_i be an element of L, The function mapping each e_i onto the unique l_j containing e_i is

called the *characteristic function* of P(L) and will be denoted Cp: L → P(L).
5) The function mapping e_i onto the offset into $C_p(e_i)$ is called the *offset function* of $L_{P,G}$, Off[P,G]: L → *N*.

The success of a generating algorithm, depends crucially on the choice of ~ and thereupon on selecting an ordering relation such that functions O, G and B can be found which are suitable for efficient evaluation in low-level software, or even in hardware. This criterion is far from a demand that these functions should have simple analytical forms. On the contrary, evaluation may involve any appropriate technique including, for example, interpolation of values in pre-calculated look-up tables.

Once a partition for the basis has been chosen, the generation algorithm is:

```
const null = −1;
type Orbit = record
                OrbitIndex: −1..MAXINT;
                Descriptor: OrbitDescriptor;
             end;
var CurrentOrbit: Orbit;
  Basis_Elt: Slater Determinant;
begin
  CurrentOrbit := FirstOrbit;
  repeat                          (*outer loop*)
    if CurrentOrbit.OrbitIndex <> null then
    begin
        Basis_Elt := Base(CurrentOrbit);
        OrbitComplete := false;
        repeat                    (*inner loop*)
          Output(Basis_Elt);
          Basis_Elt := G(Basis_Elt);
        until Basis_Elt = Base(CurrentOrbit);
    end{if}
    CurrentOrbit := O(CurrentOrbit);
  until CurrentOrbit = FirstOrbit;
end;
```

where OrbitDescriptor is a complex type representing the l_i, and FirstOrbit a predefined variable of type Orbit. The roles of G and O in this are clear.

Structured basis algorithms are distinguished by the means employed to produce the structured Slater determinant basis. However, subdivisions in the class can be identified according to how the Hamiltonian search is conducted. The prototype Shell Model Processor is designed to work with a subclass which will be called *Dual-Basis* (DB) algorithms, characterised by the generation of a second basis for each element of the original basis produced (fig. 4). This corresponds to treating the latter as the indexing element of a row of the Hamiltonian, which is searched element by element. As each element of the secondary basis is produced, it is compared against the primary element to check the Hamming distance zero condition. If a column counter is maintained against the second basis, the index of any secondary element which fails need not be computed, since it will be available directly.

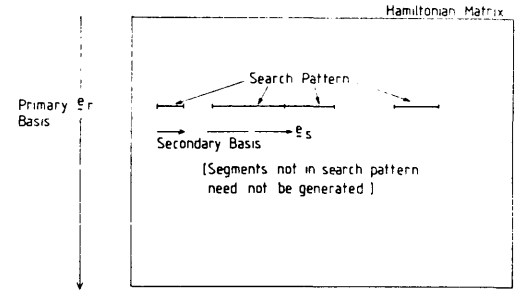


Fig. 4. Dual Basis generation.

The advantages of this approach hinge on the success with which the functions O, and especially G, can be implemented. If a relatively simple high-speed hardware engine can be constructed to realise G, then an efficient search can be conducted. The algorithm is of course optimal with respect to accesses to shared storage, which are only required for retrieval of Lanczos vector coefficients. However, the wastefulness of the search is equally evident. Unlike the Glasgow Program, all Hamiltonian entries are tested: there is no means of selecting only pairs of basis elements which fail the zero condition test, without explicitly conducting that test. Fortunately the test can be carried out in hardware at very high speed. If A and B are two representation words for basis Slater determinants, then the Hamming distance between them is the number of ones in the exclusive-OR of A

and B. If the number is 0, 2 or 4, the test is failed and the corresponding Hamiltonian entry must be evaluated; otherwise, that entry is zero.

The pilot SMP system is designed to investigate parallel implementations of DB algorithms. The machine can handle Slater determinants with up to 32 single-particle states, although it has, as yet, only been tested on sd-shell iterations. It is intended to act as an experimental prototype for a much larger system (Phase II) which would have a similar architecture, but might use other structured basis algorithms. As part of the project, the authors have defined a multiprocessor architecture (the MMPU [6]) which is capable of supporting any structured-basis algorithm (and, for that matter, algorithms of the Glasgow type).

The present SMP consists of a prototype MMPU connected to a Hamiltonian *Matrix Format Generator* (MFG). This is a dedicated subsystem which uses a dual-generation algorithm to search the Hamiltonian, identifying pairs of basis elements which fail the zero-condition test and passing them on to the MMPU for evaluation and multiplication. The MFG generates the primary basis by software running on an internal microcomputer *the MFG Controller*, but the production of the secondary basis and the subsequent zero-condition testing is implemented almost entirely in hardware. There are 4 functional units (fig. 5).

1) *The MFG Controller* is a Motorola MC68000-based microcomputer, designed and developed by the authors, responsible for overall supervision of the MFG as well as execution of the Primary Basis Generator software.

2) *The Secondary Basis Generator*, a high-speed (ECL technology) synchronous hardware accelerator, generates the secondary basis in response to each new basis state produced by the Controller. In fact the Secondary Generator can itself produce, unaided, only a single orbit of the chosen partition, and must, therefore, also be supplied with a search pattern by the Controller. Sometimes it is possible to eliminate entire orbits from the search because it can be shown in advance that none of their Slater determinants can possibly fail the zero condition test with a Slater determinant in the current primary orbit (fig. 4). The MFG's dual-generation algorithm allows very easy identification of such inter-orbit incompatibility and Secondary Generator search patterns are chosen accordingly. Early simulations and subsequent experiments with the full system have shown that the total search can be reduced by about 25%, using this technique.

3) *The Pair Filter* accepts each pair of Slater determinants ($|i\rangle$, $|f\rangle$) output from the Primary and Secondary Generators, performing a hardware zero-condition test, discarding pairs which pass, but synthesising job-packets for pairs which fail. These packets, each 42-bits long are passed to the MMPU where each will initiate an independent concurrent evaluation/multiplication process.

4) *The MFG Buffer* is a high-speed FIFO which evens out inhomogeneities in the production rate of pairs by the MFG and their consumption by the MMPU. The current Secondary Generator operates on a minor cycle of 118 MHz, producing one Slater Determinant about every 100 ns (Major cycle about 10 MHz). Of these, in a typical large sd-shell calculation, less than 1% will fail the zero-condition test, and initiate an MMPU process. The buffer must be capable of accepting job-packets from the MFG at the sustained maximum rate, while still allowing (asynchronous) reads from MMPU processors which have become idle.

The three hardware modules form a sequential fully pipelined machine and operate in asynchronous parallelism with the Primary Generator software.

The MFG algorithm creates a basis O-partition as follows. Each 32-bit Slater determinant is divided into four bytes, which are indexed:

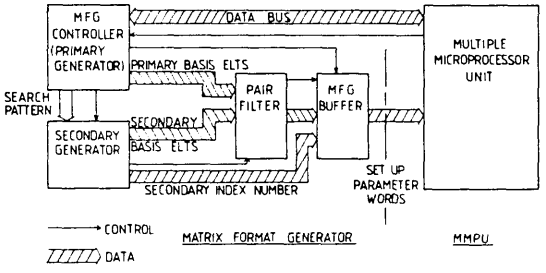


Fig. 5. Matrix Format Generator (logical block diagram).

Bits 0–7 are designated 0;

Bits 8–15 are designated 1;

Bits 16–23 are designated 2;

Bits 24–31 are designated 3.

Bytes 0 and 1 contain the neutron orbitals, while bytes 2 and 3 contain the proton orbitals. The equivalence relation chosen is:

Let A and B be Slater determinant representations. Define ~ by:

$$A \sim B \Leftrightarrow$$
$$n_i(A) = n_i(B)$$

and

$$m_i(A) = m_i(B) \quad i = 0, \ldots, 3,$$

where $n_i(A)$ is the number of set bits (occupied orbitals) and $m_i(A)$ the contribution to M_J from the ith byte of A.

The Secondary Generator implements a generating function G for this relation as follows. To each byte of the Slater determinant representation word, there corresponds a *channel* in the Generator. If the ith channel is seeded with a particular pattern of 8-bits, then the channel will produce, in sequence, all patterns with the same n_i and m_i as the seed. A complete orbit of the above relation can be produced by assigning a significance weight to each channel (e.g. 3 the most significant) so that the whole unit acts like a counter, channel 0 running through its entire chain before the output of channel 1 moves on etc.

The O-function itself is implemented (in software) by a combination of shifting set bits and searching for 4-way partitions (in the number theoretic sense) for the total M_J of the configuration space. This performs well, but, at least in its present form, would not be suitable for implementation in high-speed hardware. This could be a disadvantage for very large calculations, where the combinatorial size of the problem could overwhelm a single processor (although a multiprocessor solution is possible, the speedup is not, in the general case, predictable), forcing the Generator to wait. For this reason, the authors have been concerned to develop alternatives which would be more amenable to a hardware realisation.

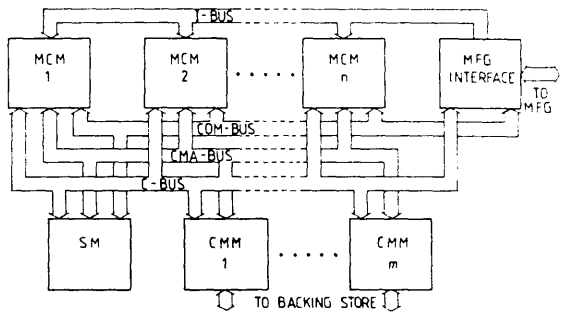


Fig. 6. Block diagram of MMPU.

The MMPU is described in detail in ref. [6]. Multiple shared buses connect a number of general-purpose processing elements called *Microcomputer Modules* (MCMs), *Central Memory Modules* (CMMs) which provide bulk storage for global data such as Lanczos vectors, the MFG Controller and a *Supervisor Module* which coordinates the activities of the system as a whole (see fig. 6). Referring to fig. 6, the MCMs, which are responsible for executing the evaluation–multiplication processes setup by the MFG, read asynchronously from the MFG buffer using the high-speed dedicated I-bus. All interactions with the shared Lanczos vectors are handled by the 64-bit CMA-bus, which is used only for accesses to CMMs. Driven by specialised interfaces, both these buses are optimised to permit maximal sharing of centralised resources, and, in particular to support the greatest possible throughput for Hamiltonian element processes.

Although a full MMPU has not been implemented (the prototype has only two of the buses installed at present), the authors have been able to run complete sd-shell iterations using the pilot system. The results are in accordance with expectations.

1) The MMPU shared resources and communication structures would not present a bottleneck unless demand were increased by some two to three orders of magnitude over that required by the present sd-shell system. This can be considered the ultimate limit of the MMPU structure as presently defined.

2) MCMs have no rigidly defined internal architecture (although they have rigidly defined func-

tional specifications) and are intended to facilitate replacement by technologically superior units as these become available. The most powerful MCM installed to date is capable of handling about half the average output of the prototype MFG. Two such modules would give the prototype processor about half the speed of an IBM 360/195 executing the Glasgow program. It would be feasible already to construct MCMs with 5 to 10 times the processing capacity of these using standard microprocessors (a preliminary design already exists based on the Motorola MC68020 in a multiprocessor configuration).

From this it is apparent that the current system bottleneck lies within the MFG rather than the MMPU. The MFG is of course the only part of the SMP which is still sequential. The authors estimate that state-of-the-art ECL technology would allow the construction of an MFG no faster than ten times the speed of the prototype. The solution, clearly, is to introduce parallelism into this part of the algorithm as well.

5. Increasing the parallelism

The most obvious solution to the bottleneck presented by the MFG is to attempt to introduce parallelism by dividing the work of the dual-basis search into several concurrent parts which can be tackled simultaneously by an array of dedicated machines. This is obviously possible, for example, at the row level, by assigning different rows of the matrix to different Secondary Generators, operating together. What is not immediately clear is how the MFG Controller task (primary generation plus search pattern computation) can be divided up, should it, as appears likely in very large calculations, exceed the capacity of a single microprocessor. One way in which this might be achieved would be to separate the primary basis generation and Secondary support functions (see e.g. the design of fig. 7). Although the latter may seem the more substantial of these tasks, search patterns are identical for all Slater determinants in a given primary basis orbit, so once a pattern is computed, if it can be stored it can be reused repeatedly.

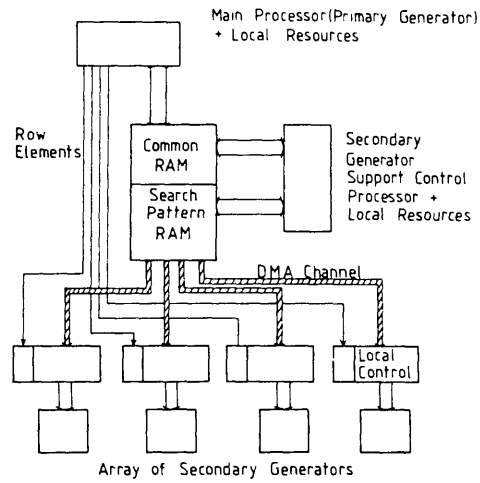


Fig. 7. Separation of Primary Generator and Secondary Generator Support functions in parallel MFG using dual-basis algorithm.

Despite the apparent simplicity of this extension, there are some outstanding issues. Firstly, the division of the task is not homogeneous, so performance improvements are not very predictable over a range of calculations. Secondly, the requirement for specialised high-speed hardware is substantial, and may only be feasible if VLSI custom devices are fabricated (ideally in CMOS). Although these problems are certainly solvable, there is another approach, based on a related but distinct class of algorithms called *element-placement* algorithms.

Element placement algorithms use the same Hamiltonian generation techniques as the original Glasgow Program, but in the context of a structured basis. The primary basis is generated using an O-function and orbit generator. As each primary basis element is produced, however, connecting secondary elements are computed exactly as in the Glasgow scheme. When such an element, e_j, is discovered, its index is established by evaluating

$$\mathrm{Ip}(\mathrm{Cp}(e_j)) + \mathrm{Off[P,G]}(e_j).$$

It is obvious that to be suitable for such an algorithm, a partition must be found which not only has suitable O, G and I, but also $\mathrm{I} \circ \mathrm{C}$ and Off[P,G]. This latter requirement is much more

demanding than the former, and indeed the SMP basis generation algorithm does not satisfy it particularly well. However, recent work has indicated that algorithms can be found which meet the more stringent conditions satisfactorily.

A variation of the SMP basis generation algorithm, called *fixed-occupancy* generation, has revealed some encouraging properties which appear to indicate its suitability for either a dual-basis or element-placement approach. In fixed occupancy, the Slater determinant representation word is divided into unequal groups of orbitals (bits), each group comprising all single particle states with the same value of m (z-component to angular momentum). The advantage of this is that once n_i is fixed for each group, m_i is automatically predetermined. In the sd-shell for example there are 6 groups each for protons and neutrons (fig. 8). The basis is divided into orbits completely characterised by twelve occupancy figures. The first advantage of this is that the G-function can be implemented very easily. The set of possible ways of distributing n_p particles amongst the 6 proton groups has at most 106 members (where $n_p = 6$); likewise for n_n. If two lists are maintained in memory corresponding to each of these sets, it is a simple process to cross-couple them in such a way as to generate the base of every basis

$1d_{5/2}$ $m_J = +5/2$

$1d_{5/2}$ $m_J = +3/2$
$1d_{3/2}$ $m_J = +3/2$

$1d_{5/2}$ $m_J = +1/2$
$1d_{3/2}$ $m_J = +1/2$
$1s_{1/2}$ $m_J = +1/2$

$1d_{5/2}$ $m_J = -1/2$
$1d_{3/2}$ $m_J = -1/2$
$1s_{1/2}$ $m_J = -1/2$

$1d_{5/2}$ $m_J = -3/2$
$1d_{3/2}$ $m_J = -3/2$

$1d_{5/2}$ $m_J = -5/2$

Fig. 8. Division of single-particle proton orbitals into 6 fixed-occupancy groups.

orbit. The orbit generating function G can be implemented even more easily than in the byte-splitting SMP case.

Fixed-occupancy is in many ways a more natural algorithm than that used in the SMP. Because its characteristic and offset functions are relatively easy to implement on a real computer, it is suitable for use in the SMP-like dual-basis, or the more efficient element-placement matrix generators. Element-placement, however, presents sufficient difficulties to a hardware accelerator designer to make a cost-effective MFG less likely. It is probable that such an approach would be best served by increasing the number of MCMs (with eventual paralleling of entire MMPUs) to cope with the increased CPU load of matrix generating processes.

The SMP algorithm is ideal in applications involving the sd-shell, or allowing access to a restricted subset of the pf-shell. In more substantial calculations, however, parallel Secondary Generators would be necessary, and the load on the MFG Controller would be greatly increased. Using fixed-occupancy generation, the Controller itself could be largely replaced by hardware, with a processor needed only for the most high-level supervision. However, for really large calculations, element-placement allows a pure multiple processor architecture, like the MMPU, to be employed without an MFG. The CPU load would, of course, be heavier, but with an MMPU based architecture this is certainly a real alternative.

6. Conclusion

A dedicated processor, would appear to give nuclear theorists a real chance to conduct calculations which would otherwise not be practical. Such a system can, as demonstrated by the Glasgow Shell Model Processor project, be constructed at relatively low cost, in a modular fashion, so that its capacity can be extended when required. A large scale pf-shell calculator could be based purely on the MMPU architecture, using fixed-occupancy element-placement matrix generation, or, given the availability of a VLSI implementation of a fixed-occupancy dual-basis MFG, by a similar

configuration to that now used in the pilot system. A choice between these approaches would depend on detailed simulations on very large configuration spaces, but both appear suitable for adoption in cases where the dimensions involved are of the order of 10^6–10^7.

References

[1] P.J. Brussard and P.W.M. Glaudemans, Shell-Model Applications in Nuclear Spectroscopy (North-Holland, Amsterdam, 1977).

[2] R.R. Whitehead et al., in: Advan. Nucl. Phys., vol 9, eds. M. Baranger and E. Vogt (Plenum Press, New York, 1977) p. 123.

[3] S.S. Schweber, An Introduction to Relativistic Quantum Mechanics (Row and Peterson, Evanston, 1961).

[4] J.H. Wilkinson, The Algebraic Eigenvalue Problem (Oxford University Press Oxford, 1965).

[5] W. Lederman, Introduction to Group Theory (Longman, London, 1973).

[6] L.M. Mackenzie et al., Computer J. 30 (1987) 110.

[7] L.M. Mackenzie et al., in: The Recursion Method and its Applications, eds. D.G. Pettifor and D.L. Weaire (Springer-Verlag, Berlin, 1985) p. 165.

A Multiple Microprocessor System for CPU-bound Calculations

L. M. MACKENZIE,* A. M. MACLEOD AND D. J. BERRY
Department of Natural Philosophy, University of Glasgow, Glasgow G12 8QQ

This paper describes a multiple microprocessor system, under development at Glasgow University, for application to calculations arising in the theory of the Nuclear Shell Model. It is the intention of the authors to discuss the architecture rather than the operation of this machine, and to concentrate particularly on design features which will allow future expansion of both capability and applicability within the range of such computations.

Received September 1985

1. INTRODUCTION

In recent years there has been a growing awareness of the potential provided by massively parallel systems to bridge the frustrating gap between computer performance and the computational demands of present-day scientific research. While performance limitations have tended to set fairly tight constraints on the applicability of integrated microprocessing units to highly CPU and memory-intensive concurrent computations,[1] VLSI fabrication techniques have increased the processing power of such devices by up to two orders of magnitude in the last decade. In consequence, many of the microelectronics manufacturers are now acutely aware of the potential of their latest products to influence the designs of high-performance computer architectures, as witnessed, for example, by the marketing of the Inmos IMS T424 'transputer', a 32-bit single-chip microcomputer proclaimed by its designers as an ideal building block for extensive multiprocessor assemblies.[2]

While significant national programs are currently directed at the development of general purpose 'fifth-generation' parallel architectures, the performance of 'state-of-the-art' VLSI technologies can be brought to bear on intractable numerical or logical calculations by means of more specialised, but relatively low-cost, modular multiple microprocessor systems, dedicated to the solution of particular classes of problem. This approach has several important advantages as follows.

(1) The performance attained can be very high, even in terms of absolute comparison with contemporary supercomputers, and is subject to incremental improvement when required.

(2) Since the machine has the character more of a laboratory-based super-calculator than a computer installation, comparisons of absolute performance are in any case grossly pessimistic. The effective processing capacity (power/availability product) at the disposal of a research group can be several orders of magnitude greater than that provided by an annual allocation on a centralised supercomputer installation.

(3) Running and maintenance costs of a well-designed machine are so low that it should be possible to recoup the initial construction outlay rapidly in saved mainframe time. Modularity, in particular, if effectively exploited, facilities rapid repair of hardware failures.

The difficulties involved in an undertaking of this kind, however, are not negligible. There is a fundamental requirement for a flexible and extensible hardware design, optimally adaptable within often rigid financial and operational constraints. Adaptability is important since future perturbations or extensions to a method cannot always be foreseen. There is, in addition, the potential bonus that an architecture with a sufficient degree of inherent generality might form the basis of similar dedicated systems devoted to other, perhaps quite unrelated, computational problems, thus reducing research and development effort in future undertakings.

The idea of machines dedicated to specific problems or classes of problems is not, of course, new, and has found favour especially in theoretical physics.[3] The authors are interested in the design of systems of this kind and have, in particular, been concerned with calculations of the type arising in the theory of the Nuclear Shell Model. The remainder of this paper will outline a design for a *Nuclear Shell Model Processor* which exemplifies the above approach, and which has, in fact, already been partially implemented in an ongoing development project.

2. THE SHELL MODEL PROCESSOR: WHY A MULTIPLE MICROPROCESSOR SYSTEM?

In quantum mechanics, each observable quantity (position, momentum, energy, etc.) is represented as a linear operator acting on a *configuration space* of state vectors, corresponding to the allowable 'states' of the target system. The Nuclear Shell Model involves a study of such quantum mechanical configuration spaces of very large dimension. State vectors with as many as 10^6 elements, and matrix operators with 10^{12} entries are generated by nuclei of only medium mass number. An ideal Nuclear Shell Model Processor should be capable of performing a range of relevant computations including the determination of the eigenvalues and eigenvectors of quantum operators, density matrix elements of state vectors, expectation values of observables, etc. The calculation of the energy eigenstates (the eigenvectors of the energy operator) of a given nucleus, in particular, is at the same time both crucial to the theoretical development and exceptionally computationally demanding, involving the evaluation and diagonalisation of a symmetric matrix operator, the *Hamiltonian*, acting on the nuclear configuration space. The *Lanczos* algorithm is now accepted as the standard method of

* Now at Department of Computing Science, University of Glasgow.

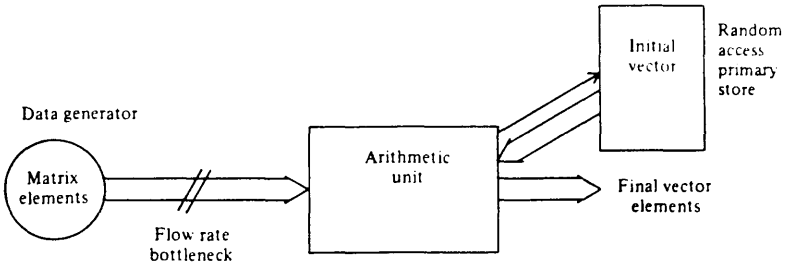


Figure 1. Multiplication of a large dimensional vector by an irregularly sparse matrix

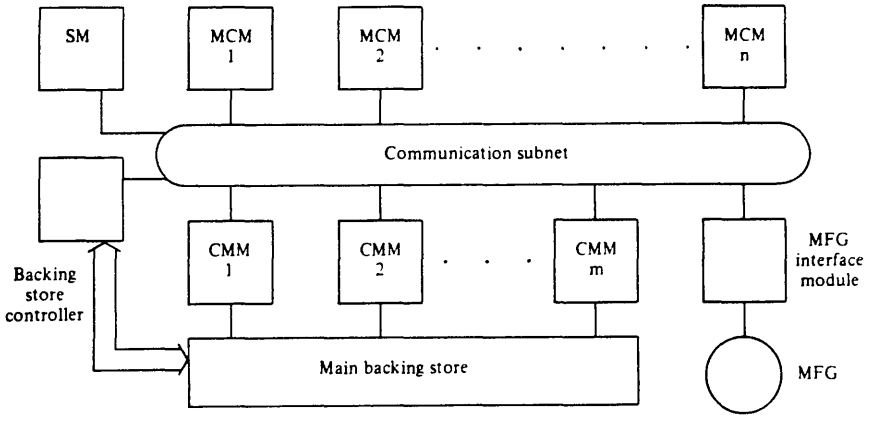


Figure 2. Shell model processor station (MMPU). Note: subnet provides a communication service between any pair of modules interfacing to it.

tri-diagonalising the Hamiltonian matrix in an angular-momentum uncoupled representation, since the approximations to the lower eigenvalues converge after only relatively few (say 100) iterations.[4] The capacity to execute this algorithm is, therefore, a necessary, but by no means a sufficient, condition for a successful Shell Model processor.

A Lanczos iteration involves several matrix/vector operations, of which the most time consuming is the multiplication of a nuclear state vector by the Hamiltonian. While, at least in principle, modern VLSI technology makes the construction of a powerful and dedicated parallel matrix-vector multiplier a fairly straightforward undertaking, the capability of a machine of this kind is severely constrained when the arrays involved are very large and, as in this case, irregularly sparse. The matrix, with perhaps more than a thousand million real entries, cannot be held in primary storage, so that there is a practical limit to the rate at which operands may be fed to an arithmetic processor (see Fig. 1).

There are two alternative approaches to this problem.

(1) *Matrix storage*. The matrix can be computed once and held on disk, being retrieved and fed to the arithmetic unit during each iteration.

(2) *Matrix generation*. The matrix can be generated in real time during each iteration, without ever being actually stored.

Since the number of elements is so large, the former approach would require some tens of gigabytes of on-line, fast secondary storage, and the technique is inevitably extremely expensive; indeed for large calculations it is probably not feasible. Matrix generation, on the other hand, appears to have a greatly superior overall ratio of performance to cost, but requires substantial additional computational power. The authors have developed and tested a prototype generator, the MFG (Matrix Format Generator), which combines a high-performance MC68000 microcomputer and a dedicated ECL hardware accelerator, to produce in real time partial descriptions of the Hamiltonian matrices of sd-shell nuclei (i.e. those with between 9 and 20 protons and between 9 and 20 neutrons) identifying the positions, but not the values of all non-zero elements. The problem of evaluating these elements is highly parallel in nature, but has an asynchronous heterogeneous nature which demands the versatility of a multiple CPU machine rather than, say, an array processor. A multiple microprocessor system of the kind discussed above is an ideal solution in this situation. Although it does not exclude a storage approach for smaller calculations, it can provide the flexibility and performance, at suitably low cost, to run generation algorithms.

The Shell Model Processor project is seen as consisting of two phases. Phase I, now approaching completion, is a practical feasibility study, involving the construction of a 'pilot' multiple processor, driven by the MFG, and capable of handling calculations with up to 32 single-particle nuclear orbitals. Phase II will require the production of a significantly more ambitious machine with up to 4 times this orbital capacity. The Phase II system, as currently envisaged by the authors, is

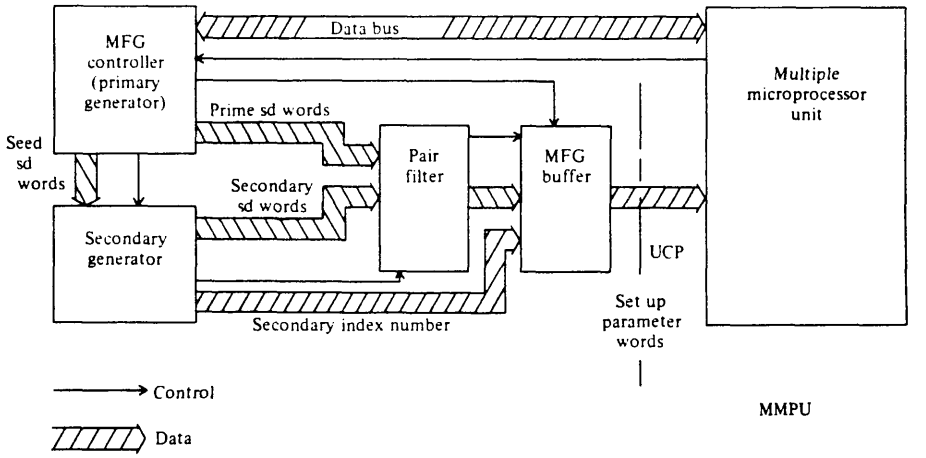


Figure 3. Matrix format generator. Logical block diagram

essentially an extension of the existing Phase I processor; in the following discussion the complete extended architecture will be described, indicating those areas not applicable to the prototype.

3. GENERAL ARCHITECTURE

The multiple microprocessor system developed by the authors for application to Nuclear Shell Model calculations is based on the modular architecture illustrated in Fig. 2. The fundamental building block is a self-contained station called a *Multiple Microprocessor Unit* (MMPU) which has stand-alone capability but can be linked to other stations, providing scope for horizontal expansion should it ever be desired. The present work will be (even in Phase II) restricted to the construction of a single MMPU which should have performance characteristics more than adequate for projected requirements. Within an MMPU, control resides with a single *Supervisor Module* (SM), which coordinates the activities of a number of general-purpose processing elements called *Microcomputer Modules* (MCMs), each an independent computer in its own right. All the MCMs may randomly access the shared *Central Memory Modules* (CMMs), which provide bulk storage for global data such as the vectors in a Lanczos iteration. Additionally they may obtain parameters from an external generator which can act as a data driver for internal MCM processes. This generator, which could be a massive secondary storage facility or a front-end processor, acts as the source of matrix elements during the Lanczos matrix-into-vector step. The present arrangement uses the prototype MFG in this role (Fig. 3), and is expected to continue until the system is required to execute calculations involving nuclides with active pf shells.

The fundamental feature of any multiple processor system is its communications *subnet* to which all its constituent processors (*hosts*) interface. The subnet's properties are defined by the system interconnection topology and, for many applications, determine the absolute limits of performance. Conventional multi-microprocessors fall squarely into Flynn's MIMD category:[5] systems consisting of many processors running what are essentially autonomous but, in general, intercommunicating processes. It is now widely accepted that, for large machines in this class to be successfully implemented, individual processors must be endowed with local resources (especially memory) so that the global subnet is loaded only when necessary. In particular, CPU references to the instruction stream and to local variables can be removed from the subnet altogether, significantly reducing the *utilisation ratios* of individual CPUs (i.e. the ratio of subnet bandwidth required by a processor to total bandwidth required by that processor).

Many structures have been proposed for multiple processors: for example the crossbar switch in Carnegie Mellon's C.mmp;[6] shared memory in UMIST's CYBA-M;[7] a binary 'n'cube' in Caltech's COSMIC CUBE;[8] linked buses in Cm*, etc.[6,9] The MMPU subnet is based on a simple multiple shared bus. Despite, or perhaps because of, their simplicity, bus-oriented subnets have several significant and desirable natural properties.

(1) The subnet does not require internal 'intelligence'. The routeing, congestion and flow control problems characteristic of, for example, packet-switched networks, are eliminated or, more precisely, are reduced to the level where they can be handled by fast hardware. Bus hardware, in general, is simple, fast, reliable and relatively easy to debug.

(2) The subnet is itself symmetrical in the sense that any node can reach any other directly with no routeing delay. The symmetry makes it particularly easy to interface special devices to the system, such as, for example, shared memory modules or special processors.

(3) The subnet is flexible in that total available bandwidth can be divided in any desired way amongst hosts. Thus a specialised host requiring, say, heavy bursts of traffic (e.g. an array processor) can be allocated as much bandwidth as arbitration protocols permit, up to, of course, the total available limit.

(4) The structure makes not only point-to-point but also broadcast transfers extremely easy to effect. The latter are often very useful where globally significant information has to be transmitted, or where multi-host synchronisation is desired.

Although these advantages are clear, bus structures have tended to be regarded as rather restrictive. The total

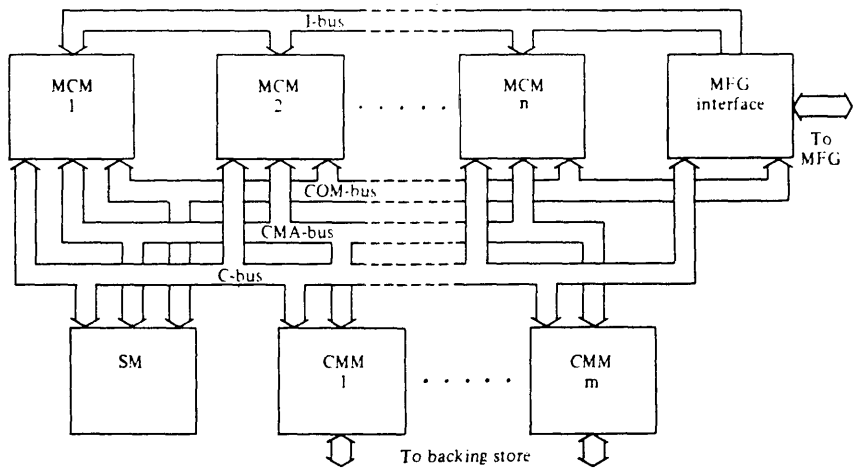


Figure 4. Block diagram of MMPU (Phase II)

available bandwidth, B, on a given bus, is a characteristic upper limit determined by the technology. No matter how large B is, there is a point beyond which the system cannot grow without encountering overloading (below this point the modularity of bus systems is excellent). If B is high enough this objection is more a matter of aesthetics than a serious practical worry (any parallel machine designer would like to believe that his architecture is infinitely extensible), but until recently the problem has been precisely that values of B have been too small.

Modern bus specifications, however, offer bandwidths of between 30 and 40 megacycles/s: notably the IEEE 896 *Futurebus*[10] and the NIM *FASTbus*,[11] now being adopted as IEEE Standard 960-1984. Using advanced ECL drivers it is probably already feasible to achieve a transfer rate of 50 MHz, so that, say, a 32-bit bus with a bandwidth of 150–200 Mbytes/s is not by any means inconceivable. If restricted to essential data transfers (i.e. no code or avoidable data transfers) such a bus can satisfy the peak requirements of at least 50–100 high-performance microprocessors (e.g. NS32032 or MC68020). Hence, although the objection remains valid in the sense that a pure bus-based system is not feasible for massive parallel systems with thousands of processors, an extremely powerful flexibly coupled multiprocessor based on message passing, shared memory or both, can be constructed. Such machines could, of course, form 'supernodes' on a more extensive 'supersubnet'.

The MMPU uses four buses (Fig. 4) to provide the necessary interconnection between its host components. (As yet only two have been implemented in the pilot machine, but a reduced CMA-Bus will be added eventually.) Before discussing these individually, a general comment might be helpful. The Phase II MMPU subnet is intended to provide bandwidth requirements well in excess of those currently projected as necessary for the most powerful Phase II module designs, which might each have, say, 20–30 times the performance of a Motorola MC68000L8. The Shell Model application requires a fairly low subnet utilisation ratio for each processor, so that in fact, in this case, the communications structure described below is powerful enough to support processing technology almost an order of magnitude faster than the best available today. It is therefore feasible that a future (Phase III?) Shell Model Processor could use virtually the same subnet, but support, say, 20 modules, each with sufficient processing power to execute 100 million operations/sec.

(1) C-Bus (Command Bus) is the primary MMPU communication highway, connecting all modules together and intended to carry system-level command and control messages. It is also used in the pilot machine to transmit bulk data and process code, although this function will probably be largely subsumed by COMbus in the Phase II implementation. In order to obtain access to a pool of available off-the-shelf hardware, it was decided to base C-Bus on the now widely accepted Motorola/Mostek/Signetics/Thomson VMEbus.[12] Although the performance of this standard is moderate by comparison with the structures discussed above (< 40 Mbytes/s), it was felt that the C-Bus function could be adequately supported and that compatibility with an industry standard was consequently a more important consideration. VMEbus includes 32-bit data and address buses, four levels of daisy chained arbitration and seven levels of interrupt. Data transfer occurs via a fully interlocked asynchronous handshake.

The authors have augmented the standard VMEbus specification in two ways intended to enhance multiprocessor support.

(*a*) A *Bus Broadcast* facility has been included, enabling a suitably privileged master to write data simultaneously to any subset of MCMs.

(*b*) The lowest bus request/grant priority level BR0* BG0IN*/BG0OUT* uses a decentralised daisy-chain grant protocol which removes the position-dependent prioritisation inherent in the normal VMEbus system (none the less retained for levels BR1* – BR3*). MCMs, which are by definition isomorphic modules, share this line and thus have virtually equal priority on C-Bus.

Despite these changes, upward compatibility with VMEbus is maintained by identifying C-Bus-specific accesses using the VME Address Modifier lines. Thus a standard VME card is entirely compatible with custom-

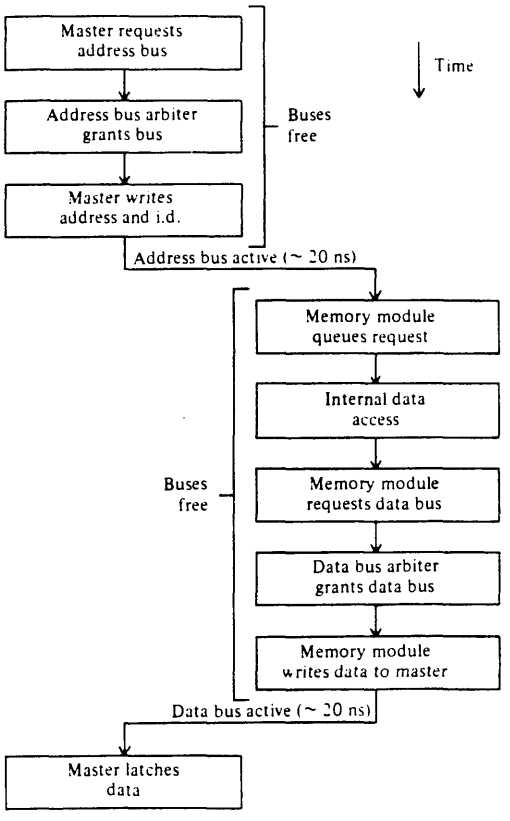


Figure 5. Concurrent address/data transfer on CMA-bus

ised MMPU modules designed to accord with the C-Bus enhancements.

(2) I-Bus is a dedicated data bus which provides MCMs with a high-speed communications route to as many as 32 single-address devices and, in particular, to the Shell Model Processor's MFG front end. Advanced Schottky bus drivers enable transfer rates in excess of 120 Mbytes/s to be attained on a 56-bit data pathway. This is considerably in excess of requirements, in keeping with the philosophy outlined above: in fact a fully populated Phase II machine would require an I-Bus bandwidth of no more than 20 Mbytes/s. Data transfer is again asynchronous, governed by a four-edged handshake; arbitration is single priority and is accomplished by means of the same decentralised daisy-chain protocol employed for C-Bus priority level 0.

(3) CMA-Bus is a bidirectional 64-bit shared data pathway designed to support fast random read/write cycles, over a 2 Gbyte range, for transfer of operands between MCMs and CMMs. The incorporation of a bus devoted to such transactions is necessary to free the system from the constraints of a conventional shared-bus architecture. The specification allows the data and address buses to operate concurrently on independent transfers so that very high performance is attainable (Fig. 5). With ECL interfaces, bus cycle times of less than 30 ns, and fully pipelined arbitration a bandwidth potentially in excess of 300 Mbytes/s would be attainable (Phase II requirements are projected as < 50 Mbytes/s). To support these access rates the CMM address space would need to be interleaved between several (say 16) independently accessible memory banks distributed over a number of CMMs: clearly, an ability to queue incoming read and write requests for later service would be required. Data determinacy during multiple read–modify–write operations on CMA-Bus will be preserved by means of hardware-driven *lockout* flags which will protect each CMM location.

(4) COM-Bus is a high-speed message-passing inter-MCM link proposed for the Phase II machine, with a maximum bandwidth of up to 200 Mbytes/s, shared on a cycle-by-cycle basis, in such a way as to allow many concurrent private conversations (a broadcast facility could easily be accommodated). High bandwidth communication between MCMs is not required during any sd-shell applications, and anticipated needs in the pilot system can easily be satisfied by C-Bus alone or, exceptionally, by means of a Central Memory 'mailbox' system.

The MMPU modules are inevitably subject to design constraints imposed by the requirement that they interface consistently to the protocols of the subnet. In the next section the three major categories of applicable design constraint will be briefly examined: hardware-imposed, C-Bus-imposed and function-imposed. Despite these restrictions there is still almost total freedom in the details of internal module design, maintaining flexibility and facilitating the replacements of older units as technological improvements permit. The following discussion is necessarily, however, incomplete and must draw heavily on experience of the Phase I implementation of the subnet.

4. DESIGN CONSTRAINTS SPECIFIED BY MMPU

4.1 Functional constraints

The functional constraints imposed on a module are dictated by the operational requirements of its role within the system. Although the Shell Model Processor could legitimately be regarded as a 'calculator' it would be misleading to restrict a discussion of its target problem set to the Lanczos iteration for, as indicated earlier, not only is this set currently more extensive (e.g. calculation of density matrix elements, expectation value of quantum observables, etc.), but in addition there is the need, as already emphasised, for inherent functional flexibility. An MCM, for example, must be capable of performing a large and, indeed, still incompletely specified list of widely differing tasks.

For these reasons it is important to construct a system which is configured to run a range of software packages, including future user-generated programs. This kind of freedom is only realistically available if an appropriate independent operating system is installed to provide a user/machine interface. The MMPU is capable of providing hardware support for operating systems ranging from the centralised to the distributed, as desired by the user. For example, the Supervisor Module could be programmed to exercise tight control over all system activities in a strictly hierarchical manner, or to intervene only when asked for assistance by an MCM.

In the case of the Shell Model Processor, since the users are liable to be themselves experienced program-

mers, and since the range of applications is liable to be relatively restricted, the operating system can be fairly unsophisticated. As envisaged at present (current software packages for the MMPU have only very limited operating system support), it will consist essentially of a supervisory executive running in a multiprogrammed environment on the Supervisor Module, overseeing a series of distributed local *kernels*, each physically resident on one of the slave modules. User processes will be assigned by the executive, arbitrarily or by user specification, to given modules, where they will run under the control of the local kernels. The operating system will handle all interprocess, and hence all interprocessor, communication, task scheduling and resource management.

User programs may be written directly in assembler, or in a high-level language provided with an appropriate library of system call procedures (the authors have done this for Pascal). In either case, operating system functions are ultimately accessed via software-generated exceptions, following a predefined protocol (e.g. in the present rudimentary system, calls are made by means of the 68000 TRAP no. 15 instruction). Many hardware resources, including the subnet and local peripherals (co-processors, I/O lines, etc.) are only available to a processor running in system mode, so that, for example, one user process wishing to pass data to another must trap to the local kernel. System processes can, of course, access the hardware directly. During a Shell Model iteration the Supervisor Module functions mainly as a watchdog, responding to interrupts generated by other modules in need of central services (in normal Shell Model processing such interrupts are typically initiated by error conditions). It is also, of course, responsible for overall coordination of the system as an iteration is scheduled or terminated, and for providing a user interface to the operator.

Functionally, each *active module* (i.e. each module capable of running a user process) will be identified to the operating system as either an MCM (general-purpose) or a special-purpose unit. Unassigned processes will only be run on MCMs, but at initiation time the operator can declare that a newly installed process is to run on a specified module.

Since modules may vary widely in their internal topology, and may indeed support several microprocessors, it will clearly be necessary to define software interfaces governing communication between the local kernel and the external operating system, consisting of other local kernels and the supervisory executive. Once this is done, internal kernel design can be tailored to suit the architectural requirements of any given module.

4.2 C-Bus constraints

The overall processor-memory description of any module must conform to the constraints imposed by the C-Bus addressing structure. Since C-Bus supports a 32-bit address bus, a processor with C-Bus master capability, when in operating system mode, will view the physical system as a 4 Gbyte block, certain regions of which may be restricted from access either by Supervisor-level protection, or by target module memory management. Of this total physical address space, each active module is assigned 128 Mbytes which are internally accessible to on-board processors without the use of C-Bus.

Up to 20 active modules may reside within an MMPU, so that a total of 2.5 Gbytes of the system space are reserved for their use. The remaining 1.5 Gbytes are divided in any appropriate manner between modules such as CMMs or other dedicated units. The 128 Mbyte block of the system address space allocated to an active module, called its *Primary Module Map* (PMM), does not necessarily contain all addressable on-board devices. It is also permissible for processors to use locations which may be switched out of the PMM or, indeed, which are inaccessible to it by direct random-access operations. There is no constraint on the number of processors which may reside within a module. If there are several, they may be organised in any desired manner, for example hierarchically, functionally or with co-equal access to on-board resources (see Section 5).

4.3 Hardware constraints

At the hardware level the only significant constraints are that each module should satisfy the electrical loading and signal protocols specified for each bus interface which it supports. Every module is interfaced to C-Bus but only MCMs and CMMs to CMA-Bus, only MCMs and peripheral interface modules to I-Bus and only active modules to COM-Bus (Fig. 4). Although an MCM must interface to the 4 system buses, only C-Bus can act as an extension of the processor's local bus. The CMA-Bus, I-Bus and COM-Bus interfaces are specially designed *pre-fetch buffers* (*PFBs*) which can conduct memory cycles independently of, and in parallel with, the on-board MPUs.

Also, there is a practical requirement for some degree of low-level software compatibility between modules. This implies a need to link the MMPU architecture to a microprocessor architecture which essentially combines currently available high performance with projected upward-compatible 32-bit machines. The authors have selected Motorola's M68000 family as, in their view, providing the optimal mix of these qualities.[14] The MMPU as presently implemented is configured to support the recently announced MC68020 microprocessor,[13] but the prototype modules which are already installed are based on the proven MC68000 and MC68010 MPUs.

5. MCM DESIGNS

To indicate the practical realisation of the concepts discussed above, it might be helpful to give some indication of the nature of the hardware which has been designed for the Shell Model Processor project. The MCM is not only the major determining factor in fixing the limits of real system performance, but it is a paradigm which can be used as a basis for the design of other active modules, and its internal architecture might be expected to be particularly instructive. A number of MCM designs (Figs 6, 7, 8) have been studied seriously. These are monoboard processing elements of increasing computational power and can perform well over a wide range of applications. However, they are tailored to tackle calculations of the type arising in the theory of the

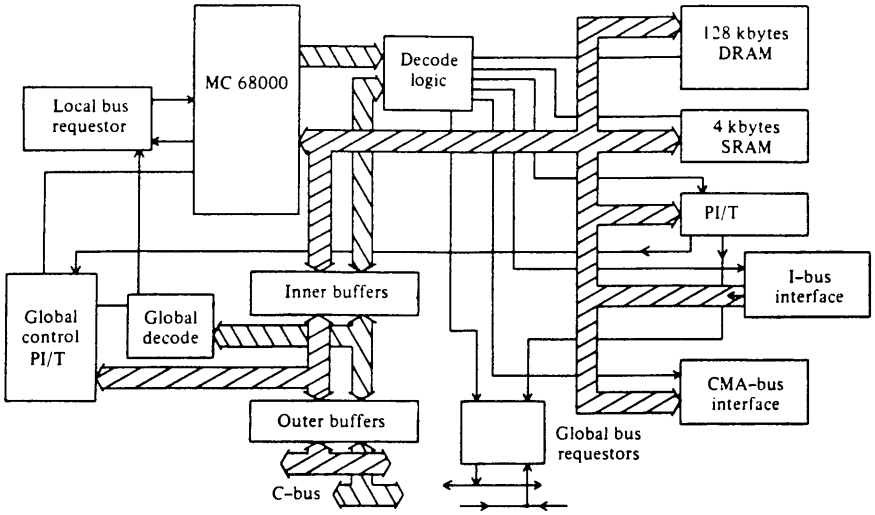


Figure 6. MCMI block diagram

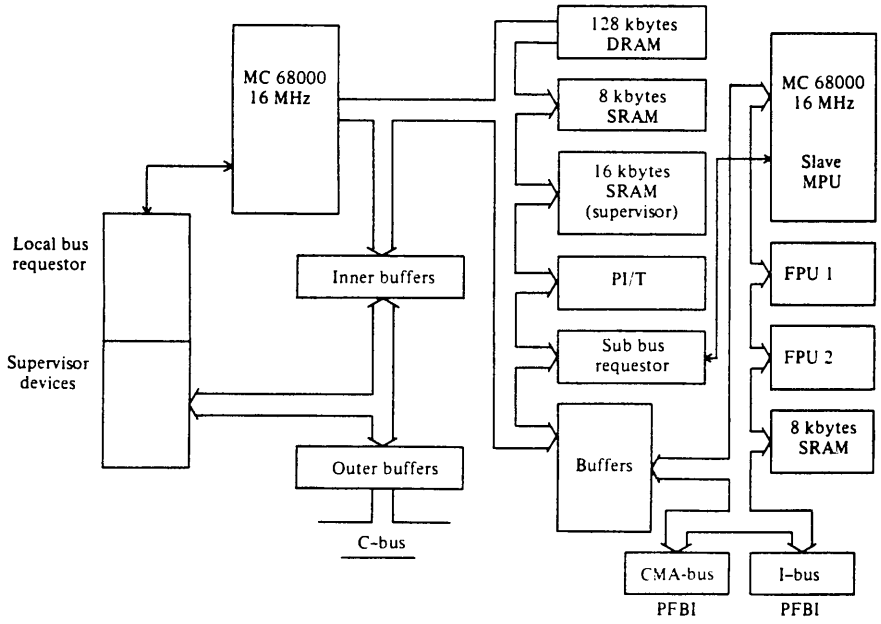


Figure 7. MCMII design

Nuclear Shell Model, and performance figures quoted must be treated accordingly.

MCMI (Fig. 6), built as part of an early feasibility study (1982), was designed rather to test system concepts than for optimal performance. The local bus topology is simple and supports only one processor, an 8 MHz MC68000, but all on-board devices are dual-port with respect to C-Bus. As with all its successors there is no on-board firmware, and all system code is loaded by the SM at initiation time into protected areas of RAM. This gives a tremendous amount of inherent flexibility, allowing dynamic tailoring of a module kernel and assisting enormously in its development and testing.

The MCMII design (Fig. 7), now operationally tested, is intended to act as an advanced prototype capable of providing processing power adequate for extensions of the calculations to higher nuclear shells. The module is hierarchically organised around a single *master* processor, an enhanced-performance MC68000 running at 16 MHz (a steady 1–2 MIPs capability). An 8 Kbyte block of very fast static RAM allows the 16 MHz processor to execute a memory access (read or write) in 250 ns (no wait states) and is intended to hold time-critical program sections and frequently accessed variables. The master MPU is also provided with 128 Kbytes or 512 Kbytes of local bulk memory which runs with 4 wait states (375 ns cycle time). A second 16 MHz 68000 acts as a slave on a local sub-bus to which are directly interfaced the I-Bus and CMA-Bus

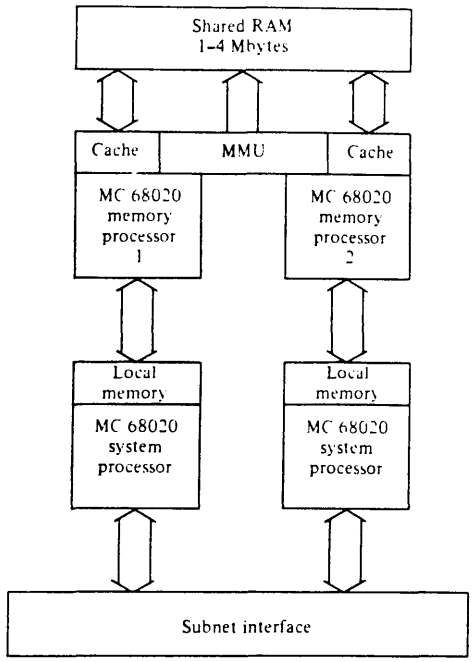


Figure 8. Design of proposed Phase II MCM (MCMII)

pre-fetch buffers together with another 8 Kbytes of fast dual-port memory, which can be used to pass data and commands between the two microprocessors. The slave also controls two National Semiconductor NS16081 Floating Point Units (FPUs), which are accessed as 16-bit peripherals and provide the arithmetic capability required by the Shell Model application. During Shell Model processing the slave handles all interaction with CMA-Bus and I-Bus as well as performing, with the aid of the FPUs, all arithmetic operations. As a guide, if MCMI performance is normalised to 1, then that of MCMII is approximately 9 during a matrix generating iteration in a Shell Model calculation.

On the basis of recent complete iterations on real nuclear data, the authors estimate that, with two MCMII modules in place, performance is approximately half that attainable on an IBM 360/195 mainframe using conventional Shell Model programming techniques.[4] Further, within the defined limits of the subnet, performance should increase almost linearly with the number of similar MCMs installed.

The Phase II MCM, now in the design stage, will be a powerful tightly coupled monoboard multiprocessor based on four MC68020 MPUs (Fig. 8), each equipped with a 'write-through' 8 Kbyte set-associative cache. In this design, the processors are paired, each pair consisting of a 'memory' processor with access to local bulk memory and a 'system' processor responsible for control of the subnet interface. The local bulk memory (1–4 Mbytes) is shared and divided into 1 Kbyte page-frames which may be dynamically designated cacheable or non-cacheable. When a task running on one of the processors attempts an access to shared memory the cache is checked while, concurrently, a local memory management unit (MMU) performs any address translations and checks access rights. If an access violation is detected the cycle is suspended or aborted; otherwise a request is issued to the on-board arbitration and a local shared-memory cycle is initiated. The MMU informs the cache whether or not the requested address falls in a cacheable page: if it does, the cache automatically stores the data as the processor reads it; if it does not, no such store may proceed. Thus only data in cacheable pages may be cached, avoiding the problem of cached data going 'stale' due to multiprocessor activity.

The proposed MMU will support demand-paged virtual memory and facilitate intertask protection in a much more general multiprogrammed multiprocessor environment. For the Shell Model application, the design of Fig. 8 is expected to yield a performance of approximately 30 on the above scale.

6. CONCLUSIONS

As outlined above, the MMPU designed for the Shell Model Processor project employs the latest 16/32-bit microprocessor technology to implement a small but powerful and flexible multiple CPU system. By emphasising modularity and linking the development to a particular microprocessor family, technological enhancement may be achieved without loss of user software compatibility. The MMPU global structures are designed to perform well above their currently projected load and it is hoped that, with scope for the integration of very-high-performance general-purpose processing elements and, indeed, of optimised dedicated processor modules where necessary, the range of applicability of the system will be significantly extended in the future.

Acknowledgements

The authors would like to thank Dr R. R. Whitehead of the Theoretical Nuclear Structure Group at Glasgow University for his assistance.

REFERENCES

1. E. T. Fathi and M. Krieger. Multiple microprocessor systems: what, why, and when. *IEEE Computer* (1983).
2. I. Barron, P. Cavill and D. May. Transputer does 10 or more MIPs even when not used in parallel. *Electronics* (17 Nov. 1983).
3. R. B. Pearson, J. L. Richardson and D. Toussaint, Special purpose processors in theoretical physics. *Communications of the ACM* **28** (4) (1985).
4. R. R. Whitehead, A. Watt, B. J. Cole and I. Morrison, *Computational Methods for Shell Model Calculations.* Advances in Nuclear Physics, vol. 9. Plenum Press, London (1977).
5. J. L. Baer, *Computer Systems Architecture.* Pitman, London (1980).
6. W. A. Wulf and C. G. Bell, C.mmp - A multi-miniprocessor. *AFIPS Conference Proceedings* 41, AFIPS Press (1972).
7. E. L. Dagless, M. D. Edwards and J. T. Proudfoot, The shared memories in the CYBA-M multi-microprocessor. *Proceedings of IEE*, E 301 (1983).

8. C. L. Seitz, The cosmic cube. *Communications of the ACM* **28** (1) (1985).
9. R. J. Swan, S. H. Fuller and D. P. Siewiorek, Cm* - A modular multi-microprocessor. *AFIPS Conference Proceedings* 46, AFIPS Press (1977).
10. P. Borrill and J. Theus, An advanced communications protocol for the proposed IEEE 896 Futurebus. *IEEE Micro* (1984).
11. Fastbus. A modular high-speed data acquisition system for high energy physics and other applications. US-NIM Committee DOE/ER-0189 (1982).
12. *VMEbus Specification Manual*, Rev. B. Motorola, Mostek, Signetics (1982).
13. *MC68020*, User's Manual. Motorola (1984).
14. E. Stritter and J. Gunter, A microprocessor architecture for a changing world: the Motorola 68000. *Computer* (1979).

Announcements

10–14 MAY 1987

APL 87, The International APL Conference on APL computer programming language, is to be held at the Fairmont Hotel, Dallas, Texas, USA. It is sponsored by the Special Interest Group of the Association of Computing Machinery and the Southwest APL Users' Group.

For further information please contact: APL 87 Registrar, 440 Northlake Shopping Center, suite 210, Dallas, TX 75238, U.S.A.

1–4 SEPTEMBER 1987

13th International Conference on Very Large Data Bases, Brighton, England, U.K.

VLDB Conferences are a forum and focus for identifying and encouraging research, development, and the novel applications of database management systems and techniques. The Thirteenth VLDB Conference will bring together researchers and practitioners to exchange ideas and advance the subject. Papers of up to 5000 words in length and of high quality are invited on any aspect of the subject but particularly on the topics listed. All submitted papers will be read and carefully evaluated by the Programme Committee.

Programme

The programme will include an exhibition, six tutorials by eminent speakers which are specially oriented towards the needs of industry, and a high standard of refereed papers. The topics covered include: Data Models; Design Methods and Tools; Distributed Databases; Query Optimisation; Concurrency Control; Database Machines; Performance Issues; Security; Knowledge Base Representation; Multi-media Databases; Implementation Techniques; Object-Oriented Models; The role of logics.

Social Programme

There will be an extensive social programme including a civic reception, traditional English events, a conference dinner, sightseeing tours and 'weekend breaks' in London.

For further information and registration forms please contact:
Miss Christine Edginton, Conference Manager, BISL Conference Department, The British Computer Society, 13 Mansfield Street, London W1M 0BP (44-1-637 0471; Telex 262284).

7–11 SEPTEMBER 1987

People and Computers HCI '87
The third annual conference of the BCS Human–Computer Interaction Specialist Group will be held at Exeter University, Devon, England from Tuesday 8 September to Friday 11 September 1987. The conference will be preceded by a day of tutorials on Monday 7 September.

The goals of the conference will again be: (i) to represent the current state of HCI, (ii) to increase communication between people working in the different disciplines of HCI and (iii) to discuss the future of HCI.

The conference has been planned in the knowledge that there is to be an international conference on a similar theme (Interact '87) in Germany the previous week. HCI '87 is designed to complement Interact '87. Many people who work in HCI in the U.K. will not be able to attend a conference held outside the U.K. Furthermore, the type of papers presented at the two conferences are likely to be of a different type. The papers in HCI '87 will be of a substantial length and will deal in detail with specific topics within HCI. In fact, HCI '87 plans to take advantage of the coincidence of Interact '87 by inviting to the U.K. international speakers, particularly from the U.S.A. and Japan, who will be in Europe at the beginning of September. There will also be workshops during HCI '87 that will report and discuss in detail issues raised, but perhaps not answered, during Interact '87. We hope that many of those who attend Interact '87 will also attend HCI '87 and play a major participatory role in making HCI '87 the success it has been in previous years.

For further details contact:

HCI '87 Conference, B.I.S.L., 13 Mansfield Street, London W1M 0BP. Telephone: (01) 637 0471.

8–11 SEPTEMBER 1987

IFIP TC 8 Conference on Governmental and Municipal Information Systems will be held in Budapest, Hungary.

For further information please contact:

IFIP TC 8 Conference Secretariat, John von Neumann Society for Computing Sciences, Budapest 5, P.O.B. 240 H-1360, Hungary. Telephone: 361 329-390. Telex: 22 5369.

"The Recursion Method and its Applications",
eds D.G. Pettifor & D.L. Weaire, Springer-Verlag Berlin, 1985, p165.

A Dedicated Lanczos Computer for Nuclear Structure Calculations

L.M. MacKenzie, D. Berry, A.M. MacLeod and R.R. Whitehead

Department of Natural Philosophy, The University, Glasgow G12 8QQ, Scotland

Abstract

Using a combination of the occupation number representation and the Lanczos method, nuclear shell-model calculations can be cast in a form which is suitable for parallel computation. An attempt to design and construct the prototype of a suitable machine is described.

1 Introduction

This talk is about an attempt to design and build a dedicated computer for use in nuclear structure calculations. There is, of course, nothing new in the idea of dedicated computers - some people think that Stonehenge was one, and the Greeks certainly had them (the *antikythera* mechanism) as did the Arabs who invented the planispheric astrolabe. Mention of such devices is not completely irrelevant to the main topic of this conference; the original need for the development of rational approximation and continued fractions arose in connection with the gearing of planetaria and similar problems.

The thing that is relatively new, however, is the ease with which one can construct analogue computers out of digital bits and pieces. In effect, a modern analogue computer uses streams of digital numbers instead of electric currents or the rotation of a wheel as the analogue quantity.

The main requirement to be satisfied before a dedicated computer can be envisaged is that the calculations to be done must be cast in such a form that each step is as computationally well matched to the machinery as possible. Other speakers have already described how the matching or *mapping* is done in the case of lattice calculations using distributed array processors. A less obvious but more striking illustration is provided by the Fast Fourier Transform. In signal processing, where there is a natural desire and need to work in frequency space, progress was slow until the Fast Fourier Transform was introduced. Almost immediately thereafter people were making dedicated on-line Fourier Transformers and the subject leapt ahead.

In the following sections we will discuss the nuclear shell model problem and describe the first attempt to build a computer whose structure matches as closely as possible the physics involved.

2 The Nuclear Shell Model

We use the expression "shell model" to refer to microscopic treatments of nuclear phenomen in which the elementary constituents are protons and neutrons. There are other kinds of nuclear models, but all of these must

ultimately be referred back to the shell model just as the shell model must ultimately be referred back to the quark structure of the nucleons.

The essence of the shell model is that each nucleon is confined in a potential well produced by its interactions with all of the other nucleons. This well is often taken to be of the form of a three-dimensional harmonic oscillator as shown in Fig. 1. The ordering of and spacings between the various shells, os, op, isod, etc. account reasonably well for some of the gross properties of nuclei and may be used as the foundation for configuration mixing studies.

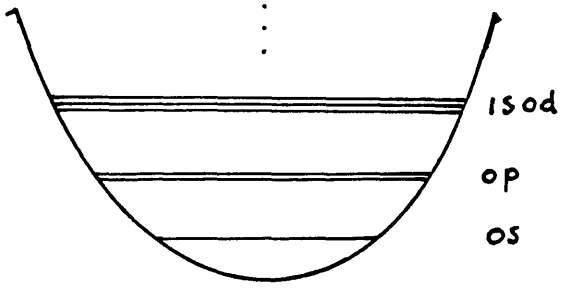


Figure 1 Schematic representation of the single-particle levels in a harmonic oscillator well

In the most usual approximation only one major shell is actively involved in the configuration mixing. The computational problem is therefore to set up the Hamiltonian matrix evaluated between the states of the active configuration and then to diagonalise it. Both eigenvalues and eigenvectors are required, the latter to enable the calculation of transition rates and expectation value of various measurable quanities. Traditionally, that is since the mid 1930's, the basis states involved have been specified by means of group theory and the necessary matrix elements evaluated using Racah algebra and the formalism of fractional parentage. Such methods are very far from being matched in the sense described above.

The Lanczos method was first used in shell-model calculations in 1968 by SEBE and NACHAMKIN [1] and by WHITEHEAD [2]. Sebe and Nachamkin used it as a matrix diagonaliser but with the idea in mind that a well chosen initial state would result in rapid convergence. Whitehead used it to calculate the tri-diagonal matrix directly from the two-body Hamiltonian without the intermediate step of constructing the full secular matrix. In both cases the basis states were specified group theoretically. A little later it was realised [3,4] that the standard formalism was an encumbrance and that the full power of the Lanczos method could be brought to bear if the basis states and the Hamiltonian were specified in the occupation number representation:

$$|i> = a^+_{i_1} a^+_{i_2} \ldots a^+_{i_n} |0>$$

$$\text{and } H = \sum_{\alpha\beta\gamma\delta} V_{\alpha\beta\gamma\delta} a^+_\alpha a^+_\beta a_\delta a_\gamma$$

where $|0>$ represents the innert filled shells, the a's and a^+'s are fermion destruction and creation operators and the $V_{\alpha\beta\gamma\delta}$ are the two-body matrix elements that define H (there is, of course, also a one-body interaction, but it is computationally advantageous to combine it with the two-body part). The operation of multiplying a vector by H could now be performed using simple bit manipulations in the computer. For example, the state $|i>$ can be represented by a string of 0's and 1's, the 1's representing the presence of the creation operators. When H operates on $|i>$ each term in the sum results in a pair of 1's being removed and a new pair inserted.

The general organisation of such a calculation is illustrated in Fig. 2. The current vector is specified by a list of amplitudes for the basis states. Each basis state is operated on in turn by the Hamiltonian as outlined above and for each turn in H a new basis state results and the product of the initial amplitude A and the V involved is accumulated in the final amplitude vector B. The process as described is simply a matrix multiplication, but one in which the matrix is stored indirectly in a highly condensed form. There is certainly scope for parallel computation since a number of initial basis states could be handled simultaneously. Unlike some of the applications described

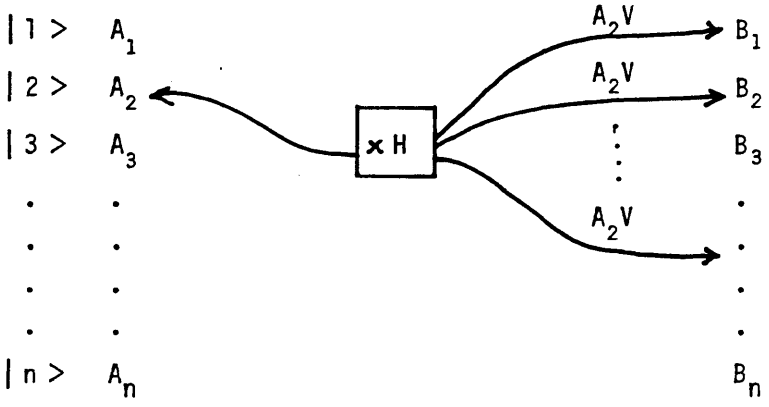


Figure 2

at this conference, though, it is the operation of multiplying a basis state by H rather than the arithmetic, the multiplication and accumulation of the A's and V's, that dominates the calculation. This is therefore not a suitable application for a single-instruction-multiple-data array processor.

3 The Prototype Machine

The advantages for shell-model work of a dedicated machine are:

(i) Low cost

(ii) Total access

(iii) Great computational power

The prototype machine to be described costs less than £10,000, will run reliably for long periods and has a performance comparable to that obtainable with an IBM 360/195. It is a quarter-scale version of the "production" machine, which will be capable of performing calculations that simply cannot be done on foreseeable commercial computers. It is nevertheless experimental in the sense that the final design is by no means fixed and the prototype is intended as a testbed for future developments rather than as a finished

machine.

The logical structure of the machine is shown in Fig. 3. The Matrix Format Generator performs the operations of creation and destruction and produces information about which A and which V (see Fig. 2) are to be multiplied and where the result is to be stored. This is passed to the Multiple Microprocessors Unit which performs the arithmetic, extracting the necessary data from and inserting the results in the Central Memory.

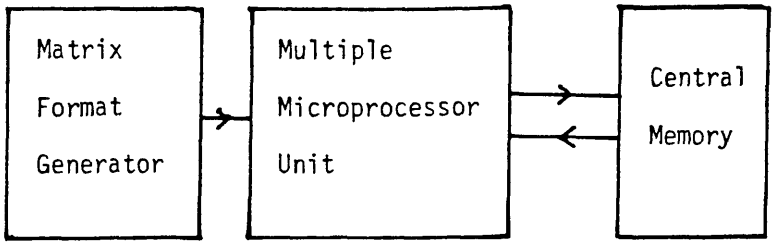


Figure 3 Logical structure of prototype machine

The Matrix Format Generator is shown schematically in Fig. 4. The Primary Generator constructs a basis state $|\, i >$ represented by a string of 32 0's or 1's (the production version will have 128). This string is fed, in parallel, to the Secondary Generator where it acts as a "seed" stimulating the production of all the other basis states that have non-zero Hamiltonian matrix elements with the seed state. In the present version this is achieved by means of a system of self-addressing tables in which each 8-bit byte of the seed state is used as the address in a table at which a suitable target byte is to be found. This new byte is used in the same way until the original seed byte is again encountered signalling exhaustion of the possibilities.

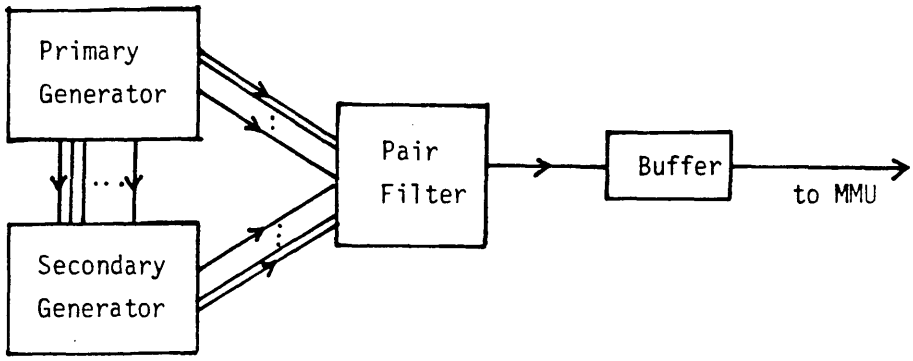


Figure 4 The matrix format generator

Owing to the conservation of additive quantum numbers such as the third components of angular momentum and isospins the Secondary Generator cannot be designed so as to produce *only* those basis states which have non-zero matrix elements with the seed state. It actually produces more states than it should. The function of the Pair Filter is to eliminate the redundant states and to extract the creation and destruction operators needed to convert the seed state into the target state. The indices of these

operators specify which V is to be used later.

The Secondary Generator and Pair Filter are constructed from very fast Emitter Coupled Logic components running at a clock rate of more than 100MHz. The output from the Pair Filter is buffered to even out the rate of presentation to the Multiple Microprocessor Unit.

The design of the Matrix Format Generator was conditioned to a great extent by the relatively high cost of memory when the project began. The present design avoids the necessity to store the full list of basis states, which would have been very expensive in the projected 132-bit machine.

The output from the Matrix Format Generator, consisting of the index numbers of the initial and final basis states and the two-body matrix element indices, passes to the Multiple Microprocessor Unit. This consists of a set of identical microcomputers arranged so that whichever one is not busy accepts the next input and performs the necessary operations (see Fig. 5).

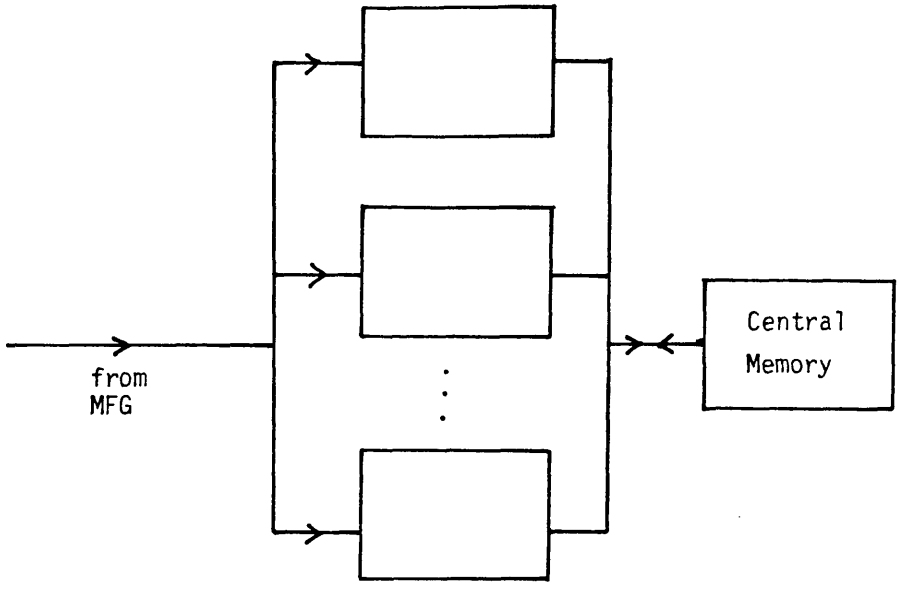


Figure 5

The tasks of extracting the relevant V and A, multiplying them together and storing the result cannot be accomplished by a single microprocessor without slowing down the Matrix Format Generator. The type of parallelism employed here is therefore one of overlapping operations in a series of asynchronous autonomous processors.

The prototype machine as described does not yet exploit all the possibilities for parallelism. For example one could have two or more MFG's each working on different sections of the basis.

The machine was originally designed around 8-bit microprocessors for the sake of cheapness. It was however designed to be "upward compatible" with newer 16 and 32 bit microprocessors of the same (Motorola) family. These

are very much faster and some have hardware floating point arithmetic. As a result of these advances we now have designs for MMU modules one or two of which will easily be able to keep up with the present MFG. This means that the MFG should now probably be redesigned. The cost of memory has also come down dramatically and this may also have a bearing on future developments.

Acknowledgments

We are indebted to the Motorola Company for assistance in many aspects of this work. R.R.W. acknowledges the tenure of an SERC Senior Fellowship during the course of the work.

References

1. T. Sebe and J. Nachamkin Ann. Phys. (NY) 51 (1969) 100

2. R.R. Whitehead 1969 Unpublished report

3. R.R. Whitehead Nucl. Phys. A 182 (1972) 290

4. R.R. Whitehead, A. Watt, B.J. Cole and I. Morrison Adv. in Nucl. Phys. Vol. 9 Eds. Baranger and Voyt (Plenum Press, 1977)